Semantic Foundations for Heterogeneous Systems

by

Roberto Passerone

Laurea (Politecnico di Torino) 1994
M. S. (University of California, Berkeley) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Spring 2004

The dissertation of Roberto Passerone is approved:

Chair	Date
	Date
	Date
	Date

University of California, Berkeley

Spring 2004

Semantic Foundations for Heterogeneous Systems

Abstract

Semantic Foundations for Heterogeneous Systems

by

Roberto Passerone
Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto L. Sangiovanni-Vincentelli, Chair

The ability to incorporate increasingly sophisticated functionality makes the design of electronic embedded systems complex. Many factors, beside the traditional considerations of cost and performance, contribute to making the design and the implementation of embedded systems a challenging task. The inevitable interactions of an embedded system with the physical world require that its parts be described by multiple formalisms of heterogeneous nature. Because these formalisms evolved in isolation, system integration becomes particularly problematic. In addition, the computation, often distributed across the infrastructure, is frequently controlled by intricate communication mechanisms. This, and other safety concerns, demand a higher degree of confidence in the correctness of the design that imposes a limit on design productivity.

The key to addressing the complexity problem and to achieve substantial productivity gains is a rigorous design methodology that is based on the effective use of decomposition and multiple levels of abstraction. Decomposition relies on models that describe the effect of hierarchically composing different concurrent parts of the system. An abstraction is the relationship between two representations of the same system that expose different levels of detail. To maximize their benefit, these techniques require a semantic foundation that provides the ability to formally describe and relate a wide range of concurrency models. This Dissertation proposes one such semantic foundation in the form of an algebraic framework called Agent Algebra.

Agent Algebra is a formal framework that can be used to uniformly present and reason about the characteristics and the properties of the different models of computation used in a design, and about their relationships. This is accomplished by defining an algebra that consists of a set of denotations, called *agents*, for the elements of a model, and of the main operations that the model provides to compose and to manipulate agents. Different models of computation are constructed as distinct instances of the algebra. However, the framework takes advantage of the common algebraic structure to derive results that apply to all models in the framework, and to relate different models using structure-preserving maps.

Relationships between different models of computation are described in this Dissertation as conservative approximations and their inverses. A conservative approximation consists of two abstractions that provide different views of an agent in the form of an over- and a under-approximation. When used in combination, the two mappings are capable of preserving refinement verification results from a more abstract to a more concrete model, with the guarantee of no false positives. Conservative approximations and their inverses are also used as a generic tool to construct a correspondence between two models. Because this correspondence makes the correlation between an abstraction and the corresponding refinement precise, conservative approximations are useful tools to study the interaction of agents that belong to heterogeneous models. A detailed comparison also reveals the necessary and sufficient conditions that must be satisfied for the well established notions of abstract interpretations and Galois connections (in fact, for a pair thereof) to form a conservative approximation. Conservative approximations are illustrated by several examples of formalization of models of computation of interest in the design of embedded systems.

While the framework of Agent Algebra is general enough to encompass a variety of models of computation, the common structure is sufficient to prove interesting results that apply to all models. In particular, this Dissertation focuses on the problem of characterizing the specification of a component of a system given the global specification for the system and the context surrounding the component. This technique, called Local Specification Synthesis, can be applied to a solve synthesis and optimization problems in a number of different application areas. The results include sufficient conditions to be met by the definitions of system composition and system refinement for constructing such characterizations. The local specification synthesis technique is also demonstrated through its application to the problem of protocol conversion.

Professor Alberto L. Sangiovanni-Vincentelli
Dissertation Committee Chair

To the ones I love

Acknowledgments

I get to put my name on the cover of this dissertation, but many are the people who contributed in part to the work and who supported me during these years. Without them, it wouldn't have been nearly as much fun.

First, I would like to thank Prof. Alberto Sangiovanni-Vincentelli for the privilege of having him as my research advisor during my career as a student at UC Berkeley. He is the one "responsible" for getting me started on this project and for getting me excited about it during all these years. While giving me a lot of freedom in shaping the research direction, he was relentless in keeping it on track. Alberto taught me what research is all about.

A special thank goes to Dr. Jerry Burch. His influence and contributions were critical in many situations and are spread throughout this work. Jerry showed me how to do research and how to look at problems from different perspectives. Our long discussions over the phone, at times intense, and always stimulating, have had a profound impact on the way I approach a problem, both in and outside research. His style is for me the model to be followed in any future endeavor.

I would like to thank Prof. Jan Rabaey and Prof. Ted Slaman for serving as members of my Dissertation Committee. I am grateful to Jan for getting me involved with his group at the BWRC, and for being a constant reminder that, in the end, we need "things that work". I am grateful to Ted for introducing me to the mysteries of logic and recursive functions, and for our delightful discussions over lunch.

I would like to thank all the staff at UC Berkeley, and in particular Flora Oviedo, Jennifer Stone, Lorie Brofferio, Mary Byrnes and Ruth Gjerde. If the students can get anything accomplished it is because of their support.

During my career as a graduate student I had the opportunity to meet some of the finest people there are. I would like to thank all the past and present members of Cadence's VCC group and of the Cadence Laboratories in Berkeley with whom I have shared these years of professional life; I am indebted to all the students and the staff of the Berkeley Wireless Research Center for the many pleasant hours spent at the center and for the exciting retreats; I want to thank the members and the staff of the GSRC for promoting such interesting research and for organizing thought-provoking meetings and workshops; and last but not least I am especially grateful to all the occupants of the old 550 Cory Hall and to my office mates in 545 Cory Hall. In particular I would like to extend a special thank to Alessandra Nardi, Alessandro Pinto, Alex Kondratyev, Claudio Pinello, Doug Densmore, Elaine Cheong, Farinaz Koushanfar, Felice Balarin, Fernando De Bernardinis, Gerald

Wang, Grant Martin, Guang Yang, Harry Hsieh, Huifang Qin, Jörn Janneck, Jerry Burch, Julio Silva, Luca Carloni, Luca Daniel, Luciano Lavagno, Luigi Palopoli, Marco Sgroi, Marlene Wan, Massimo Baleani, Max Chiodo, Mike Sheets, Mukul Prasad, Naji Ghazal, Paolo Giusto, Peggy Laramie, Perry Alexander, Rong Chen, Rupak Majumdar, Subarna Sinha, Suet-Fei Li, Trevor Meyerowitz, Vandana Prabhu, Yanmei Li and Yoshi Watanabe. It is the calibre of these people and the warmth of their friendship that makes this a great place to live.

But foremost I owe the very chance of coming to Berkeley in the first place to the support of my wife Barbara. I was fortunate to have her by my side during these years. The accomplishments of this work are hers as much as mine. And life has taken a whole new meaning with the arrival of our son Davide. It is with his smile in our hearts that we bring this work to a close. Barbara and Davide are indispensable ingredients of my life.

Contents

List of Figures

List of Definitions

Chapter 1

Introduction

Embedded systems are electronic devices that function in the context of a real environment, by sensing and reacting to a set of stimuli. Embedded systems are pervasive and very diverse. One extreme is microscopic devices [1, 77, 94] powered by ambient energy in their environment, that are able to sense numerous fields, position, velocity, and acceleration, and to communicate with appropriate and sometimes substantial bandwidth in the near area. One the other extreme are larger, more powerful systems within an infrastructure driven by the continued improvements in storage and memory density, processing capability, and system-wide interconnects. Applications are also diverse, ranging from control-dominated systems, such as those found in the automotive and aerospace industry; to data-intensive systems, such as set-top boxes and entertainment devices [3, 24, 35, 80]; to life-critical systems such as active prostheses and medical devices [97, 88].

Because of such diversity, currently deployed design methodologies for embedded systems are often based on *ad hoc* techniques that lack formal foundations and hence are likely to provide little if any guarantee of satisfying a given set of constraints and specifications without resorting to extensive simulation or tests on prototypes. However, in the face of growing complexity, cost and safety constraints, this approach will have to yield to more rigorous methods [55]. These methods will most likely include several common traits. In fact, despite their diversity, the challenges that designers of embedded systems in different application areas must face are often similar. In all cases, *concurrency*, the simultaneous execution of several elements in a system, and design constraints must be considered as first class citizens at all levels of abstraction and in both hardware and software. In addition, complexity in the design not only arises from the size of the system, but it also emerges from its *heterogeneous* nature, that is from the fact that in complex designs that interact with the real world, different parts are more appropriately captured using different models and different techniques. For example, the model of the software application that runs on a distributed collection of nodes in a network is often concerned only with the initial and final state of the behavior of a reaction. In contrast, the particular sequence of actions of the reaction could be relevant to the design of one instance of a node. Likewise, the notation employed in reasoning about the a resource management subsystem is often incompatible with the handling of real time deadlines, typical of communication protocols. These subsystems are not, however, necessarily decoupled. In fact, applications in such distributed embedded systems will likely not be centered within a single device, but stretched over several, forming a path through the infrastructure. Consequently, the ability of the system designer to specify, manage, and verify the functionality and performance of *concurrent behaviors*, within *and* across heterogeneous boundaries, is essential.

We informally refer to the notation and the rules that are used to specify and verify the elements of a system and their collective behavior as a *model of computation* [37, 38, 61]. The objective of this work is to provide a formal framework to uniformly present and reason about the characteristics and the properties of the different models of computation used in a design, and about their relationships. We accomplish this by defining an algebra that consists of the set of the denotations, called the *agents*, of the elements of a model and of the main operations that can be performed to compose agents and obtain a new agent. Different models of computation are still constructed as distinct algebras in our framework. However, we can take advantage of the common algebraic structure to derive results that apply to all models in the framework, and to relate different models using structure-preserving maps. Abstraction and refinement relationships between and within the relevant models of computation in embedded systems design, and the techniques that take advantage of these relationships, are the focus of this work.

Modern design methodologies are turning to abstraction techniques to reduce the complexity of designing a system. In addition, design reuse in all its shapes and forms is of paramount importance. Together, abstraction, refinement and design reuse are the basis of the concept of *platform-based design* [43, 20, 82]. A *platform* consists of a set of library elements, or resources, that can be assembled and interconnected according to predetermined rules to form a platform *instance*. One step in a platform-based design flow involves mapping a specification onto different platform instances, and evaluating its performance. By employing existing components and interconnection resources, reuse in a platform-based design flow shifts the functional verification problem from the verification of the individual elements to the verification of their interaction [79, 85, 81, 16]. In addition, by exporting an abstracted view of the parameters of the model, the user of a platform is able to estimate the relevant performance metrics and verify that they satisfy the design constraints. The mapping and estimation step is then repeated at increasingly lower levels of abstraction in order to come to a complete implementation.

Platform-based design is a methodology that can be applied to various application domains [25, 52]. The Metropolis project is a software infrastructure and a design methodology for heterogeneous embedded systems that supports platform-based design by exploiting refinement through different levels of abstraction that are tuned to each application area [7]. For this reason, Metropolis is centered around a meta-model of computation [6] that is a set of primitives that can be used to construct several different models of computation that can all be used in a particular design. We develop our work in the context of the Metropolis project. The long term objective of the work presented here is to lay the foundations for providing a denotational semantics for the meta-model. To reach that objective we begin by studying several of the models of computation of interest, and by studying how relationships between these models can be established. Moreover, we propose a formalization of the design methodology that makes precise the relationships between the elements of the different platforms.

We begin this introduction by informally presenting our interpretation of certain concepts, such as model of computation and levels of abstraction, that are at the basis of our approach. While doing so, we also delimit the scope of this dissertation, and discuss the principles that influenced the development of our framework. We then motivate our efforts by presenting an example of a heterogeneous embedded system that includes a simple formalization of the semantic domain and the operators of a model of computation suitable for describing objects in continuous time. The example is followed by an extensive discussion and comparison with related work in this area. We conclude this chapter with a short summary of the main contributions of this dissertation and with an annotated outline of the work.

1.1 Models of Computation and Semantic Domains

In our terminology, a model of computation is a distinctive paradigm for computation, communication, and synchronization of agents (we use “agent” as a generic term that includes software processes, hardware circuits and physical components, and abstractions thereof). For example, the Mealy machine model of computation [51] is a paradigm where data is communicated via signals and all agents operate in lockstep. The Kahn Process Network model [53, 54] is a paradigm where data carrying tokens provide communication and agents operate asynchronously with each other (but coordinate their computation by passing and receiving tokens). Different paradigms can give quite different views of the nature of computation and communication. In a large system, different subsystems can often be more naturally designed and understood using different models of computation.

The notion of a model of computation is related to, but different from, the concept of a semantic domain for modeling agents. A semantic domain is a set of mathematical objects used to model agents. For a given model of computation, there is often a most natural semantic domain. For example, Kahn processes are naturally represented by functions over streams of values. In the Mealy machine model, agents are naturally represented by labeled graphs interpreted as state machines.

However, for a given model of computation there is more than one semantic domain that can be used to model agents. For example, a Kahn process can also be modeled by a state machine that effectively simulates its behavior. Such a semantic domain is less natural for Kahn Process Networks than stream functions, but it may have advantages for certain types of analyses, such as finding relationships between the Kahn process model of computation and the Mealy machine model of computation. Our interpretation of these terms highlights the distinction between a model of computation and a semantic domain. We use the term model of computation more broadly to include computation paradigms that may not fit within any of the semantic domains we consider.

We interpret the term "model of computation" slightly differently than others. There, the meaning of the term is based on designating one or more unifying semantic domains. A unifying semantic domain is a (possibly parameterized) semantic domain that can be used to represent a variety of different computation paradigms. Examples of unifying semantic domains include the Tagged Signal Model [62], the operational semantics underlying the SystemC language [44, 90] and the abstract semantics underlying the Ptolemy II simulator [28]. In this context, a model of computation is a way of encoding a computation paradigm in one of the unifying semantic domains. With this interpretation, it is common to distinguish different models of computations in terms of the traits of the encoding: firing rules that control when different agents do computation, communication protocols, etc. For example, in Ptolemy II, models of computation (also known as computation *domains*) are distinguished by differences in scheduling policies and communication protocols.

There is an important trade-off when constructing a unifying semantic domain. The unifying semantic domain can be used more broadly if it unifies a large number of models of computation. However, the more models of computation that are unified, the less natural the unifying semantic domain is likely to be for any particular model of computation. We want the users of our framework to be able to make their own trade-offs in this regard, rather than be required to conform to a particular choice made by us. In fact, it is not our goal to construct a single unifying semantic domain, or even a parameterized class of unifying semantic domains. Instead, we wish to construct a formal framework that simplifies the construction and comparison of different semantic domains, including semantic domains that can be used to unify specific, restricted classes of other semantic domains. Our aim therefore differs from that of the Ptolemy II project where the provision of a simulator leads to a notion of composition between different models that is fixed in the definition of the domain directors, resulting in a single specific unifying domain; there, a different notion of interaction requires redefining the rules of execution. To do so, we have created a mathematical framework in which to express semantic domains in a form that is close to their natural formulation (i.e., the form that is most convenient for a given domain), and yet structured enough to give us results that apply regardless of the particular domain in question.

1.2 Levels of Abstraction

An important factor in the design of heterogeneous systems is the ability to flexibly use different levels of abstraction. Different abstractions provide a different trade-off in terms of expressive power, accuracy and ability to support automated analysis, synthesis and verification. Different abstractions are often employed for different parts of a design (by way of different models of computation, for instance). Even each individual piece of the design undergoes changes in the level of abstraction during the design process, as the model is refined towards a representation that is closer to the final implementation. Different levels of detail are also used to perform different kinds of analysis: for example, a high level functional verification versus a very detailed electromagnetic interference analysis.

Abstraction may come in many forms. For example, most models include ways to talk about the evolution of the behavior of a system in time. How the notion of time is abstracted by the model is one of the fundamental aspects that characterizes its expressive power. For instance, models of computation that are intended to closely reflect physical phenomena usually employ a notion of time based on a continuous, totally ordered metric space. It is possible to use this notion of time to describe more “idealized” systems, such as systems that transition only at specified intervals. The continuous nature of the space however introduces irrelevant details that makes the representation more cumbersome to use. A discrete space, in this case, is more appropriate. Likewise, a software application is often not concerned with the “distance” between the occurrence of events. In that case, the space need not employ a metric. In general a partially ordered, or even a preordered set is used to represent the notion of time. We refer the reader to the Tagged Signal Model of Lee and Sangiovanni-Vincentelli for an excellent treatment of this subject [62].

A related form of abstraction has to do with the concurrency model. In general, a model is concurrent if agents are capable of simultaneously executing over time (hence, different notions of time support different notions of concurrency). The way the agents synchronize during the execution distinguishes the different concurrency models. The terminology used in the literature to describe concurrency models is varied, and often used inconsistently across communities. In hardware design, the most common synchronization schemes are the *synchronous* and the *asynchronous* models. In a synchronous model, *all* agents in a system execute in lockstep, by exchanging data and simultaneously advancing their behavior [10]. This synchronization scheme is typical of systems whose agents share the same global notion of time. Conversely, in an interleaved asynchronous model, the agents take turns in executing, and advance their behavior one at a time [34]. This model is more appropriate for systems where the lockstep execution is not practical, or systems whose agents have a local, rather than global, notion of time. Most models of computation that are used in practice employ some variation on these basic schemes.

Models of systems that include software components that execute on processors are usually based on an asynchronous scheme, to account for the unpredictability of their execution time. For this reason, hardware/software co-design methodologies are often based on the combination of the two models in what is known as Globally Asynchronous Locally Synchronous (GALS) systems [5]. Here, subsystems execute synchronously, while their global interaction occurs through an asynchronous model. This model thus combines the analysis techniques that can be applied to synchronous models with the flexibility afforded by the asynchronous model. A similar scheme can also be employed in purely software-based systems [21]. Here, however, the distinction between synchronous and asynchronous has to do with the communication paradigm, rather than with the timing model. The communication is synchronous if the process that initiates it awaits the completion of the remote procedure call by transferring the flow of control. In contrast, the communication is asynchronous if the process retains the flow of control and proceeds immediately without waiting while the request is serviced by the remote agent [70].

The data exchanged during the interaction of agents may take different forms. The most common means of interaction are either *action-based* or *value-based*. In an action-based scheme of interaction changes in the environment are propagated through the system during its execution. This usually indicates the occurrence of events that the system must react to, and is typical of control-dominated applications. The event can be associated with a value. However, the occurrence, rather than the value, is the most important piece of information carried by the event. In a value-base scheme, on the other hand, the value of some quantity is continuously made available to the rest of the system. This is the case, for instance, in data-dominated applications that process a continuous stream of data [63]. These models can also be combined to take advantage of their strengths, at the expense of additional complexity [32].

Another important abstraction technique consists of restricting the visibility of the internal operations of an agent. This is an operation that alters the scope of signals and values, and is employed by virtually all design languages and models of computation of interest. By hiding the internal structure, an agent is effectively encapsulated and "protected" from the influence of the environment. This mechanism is therefore able to *localize* the effects of certain behaviors, thus making the analysis of large systems easier. Because this abstraction technique is fundamental to the construction of well behaved models, we include scoping as one of the basic operators of the models in our framework.

1.3 Refinement Verification and Local Specification Synthesis

Related to the concept of levels of abstraction is the ability in a model of computation to verify the correctness of a design relative to a specification. Several methods for verifying concurrent systems are based on checking for *language containment* or related properties. In the simplest form of language containment-based verification, each agent is modeled by a formal language of finite (or possibly infinite) sequences. If agent p is a specification and p' is an implementation, then p' is said to satisfy p if the language of p' is a subset the language of p. The idea is that each sequence, sometimes called a *trace*, represents a behavior; an implementation satisfies a specification if and only if all the possible behaviors of the implementation are also possible behaviors of the specification. Indeed this relationship between "implementation" and "specification" is a manifestation of a hierarchy between models, whereby "specifications" are at a higher level of abstraction than "implementations". The fact that a lower-level model is an "implementation" of another higher-level model is verified by "behavior containment". Thus we need a formal way of describing behavior and containment to be able to establish this relationship. Also, we like to think of the relationships "implementation-specification" as indeed the implementation being a "refinement" of the specification. Hence we may qualify refinement as the relationships between a higher-level model and a lower one, while specification and implementation may relate more properly to the model used to "enter" the design process and implementation as the one with which we "exit" the design process.

Our work in the framework is indeed inspired by relationships between models of this sort. Hence, our definitions and theorems will proceed from a definition of agents and structural properties of models towards the notion of “approximations” as a way of capturing the behavior containment idea. Because of their properties, these relationships are called *conservative approximations*. Once we have established this key relationship, it is possible to compare and combine models by finding a common ground where behavior representations are consistent. Intuitively, we may find different ways of approximating models and consequently compositions and comparisons are dependent on the approximations. This has actually been observed in applications when heterogeneous models of computation are used for different parts of a design. The different parts of the design have, of course, to interact and they eventually do so in the final implementation, but the way in which we march towards implementation depends on our assumptions about the way the two models communicate. These assumptions more often than not are implicit and may be imposed by the tools designers use, leading to sub-optimal and even incorrect implementations. Therefore, different models of computation are related in our framework by a set of approximations through a common refinement, thus clearly establishing the assumptions regarding their interaction.

Operators of composition, scoping and instantiation in a model of computation together make it possible to describe an implementation and its specification as a *hierarchy* of components. Ideally, we would like to take advantage of the modularity afforded by the hierarchical representation to simplify the task of refinement verification, by decomposing a large problem into a set of smaller problems that are collectively simpler to solve. This idea is depicted in figure 1.1. There, a specification p' is decomposed as the composition of two agents q'_1 and q'_2. Similarly, the implementation p is decomposed into two agents q_1 and q_2. If the model of computation supports compositional verification, then verifying that q_1 implements q'_1 and that q_2 implements q'_2 is sufficient to conclude that p implements p'. This technique can be applied when the operators are *monotonic* relative to the refinement relationship. The issue of monotonicity, which we extend to the case of partial operators, is fundamental in our work and is the basis of many of the general results that hold in our framework. It is also a distinguishing factor with respect to other approaches to agent modeling [30]. In particular, we insist on a notion of monotonicity, which we call $\top$-*monotonicity*, that is consistent with the interpretation of the refinement relationship as substitutability. These concepts are fully developed in section 2.4.

A related problem is that of the synthesis of a local specification, depicted in figure 1.2. Here, we are given a global specification p' and a partial implementation, called a *context*, that consists of the composition of several agents, such as q_1 and q_2. The implementation is only partially

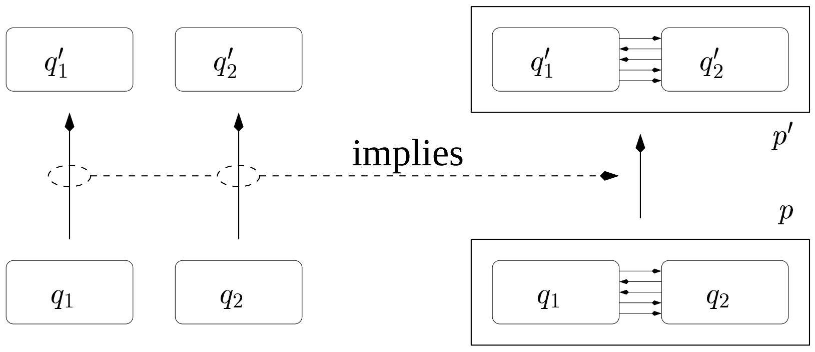

Figure 1.1: Compositional verification

specified, and is completed by inserting an additional agent q to be composed with the rest of the context. The problem consists of finding a local specification q' for q, such that if q implements q', then the full implementation p implements the global specification p'.

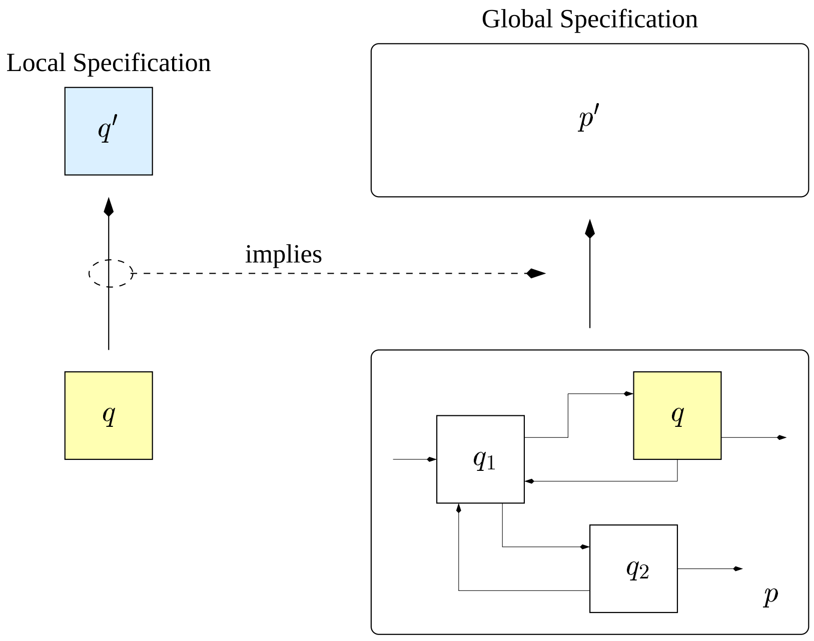

Figure 1.2: Local Specification Synthesis

The problem of local specification synthesis is very general and can be applied to a variety of situations. One area of application is for example that of *supervisory control synthesis* [4]. Here a plant is used as the context, and a control relation as the global specification. The problem consists of deriving the appropriate control law to be applied in order for the plant to follow the specification. *Engineering Changes* is another area, where modifications must be applied to part of a system in order for the entire system to satisfy a new specification. This procedure is also known as *rectification*. Note that the same rectification procedure could be used to optimize a design. Here, however, the global specification is unchanged, while the local specification represents all the possible admissible implementation of an individual component of the system, thus exposing its full flexibility [13].

We address and solve the problem of local specification synthesis in our framework. Unlike the similar problems described above, our solution is independent of the particular model of computation, since it is based on the properties of the framework, instead of some particular feature of a specific model. This gives us the additional advantage of exposing the conditions under which this technique can be applied.

1.3.1 Compatibility and Protocol Conversion

We have argued above that complexity issues can be addressed using a methodology that promotes the reuse of existing components, also known as *Intellectual Property*, or IPs.[1] However, the correct deployment of these blocks when the IPs have been developed by different groups inside the same company, or by different companies, is notoriously difficult. Unforeseen interactions often make the behavior of the resulting design unpredictable.

Design rules have been proposed that try to alleviate the problem by forcing the designers to be precise about the behavior of the individual components and to verify this behavior under a number of assumptions about the environment in which they have to operate. While this is certainly a step in the right direction, it is by no means sufficient to guarantee correctness: extensive simulation and prototyping are still needed on the compositions. Several methods have been proposed for hardware and software components that encapsulate the IPs so that their behavior is protected from the interaction with other components. Interfaces are then used to ensure the compatibility between components. Roughly speaking, two interfaces are compatible if they “fit together” as they are.

In this work we formally define compatibility as a consequence of a refinement order imposed on the agents. The order is interpreted as a relation of substitutability, called a *conformance order*, and is represented in terms of a set of agents, called a *conformance set*. The conformance set also induces the notion of compatibility in the model. Since our framework encompasses many different models of computation, we are able to represent many forms of interfaces. Simple in-

[1]The term “Intellectual Property” is used to highlight the intangible nature of virtual components which essentially consist of a set of property rights, rather than of a physical entity.

terfaces, typically specified in the type system of a system description language, may describe the types of values that are exchanged between the components. More expressive interfaces, typically specified informally in design documents, may describe the protocol for the component interaction [34, 74, 87, 29, 30, 17]. All of these can be used in our framework, and will be presented by ways of examples in this dissertation.

However, when components are taken from legacy systems or from third-party vendors, interface protocols are unlikely to be compatible. This does not mean though that components cannot be combined together: approaches have been proposed that construct a converter among incompatible communication protocols. In [74], we proposed to define a protocol as a formal language (a set of strings from an alphabet) and to use automata to finitely represent such languages. The problem of converting one protocol into another was then addressed by considering their conjunction as the product of the corresponding automata and by removing the states and transitions that led to a violation of one of the two protocols. While the algorithm was effective in the examples that were tried, it lacked a more formal and mathematically sound interpretation. In particular this made it difficult to understand and analyse its limitations and properties. The techniques developed in this work provide the formal basis to resolve those issues.

Informally, two interfaces are adaptable if they can be made to fit together by communicating through a third component, the adapter. If interfaces specify only value types, then adapters are simply type converters. However, if interfaces specify interaction protocols, then adapters are protocol converters. Here, we cast the problem of protocol conversion as an instance of a local specification synthesis. In this case, the context is represented by two different protocols that we wish to connect. The specification simply asserts the properties that we want to be true of the communication mechanism, such as no loss of data, and in order delivery. The local specification then corresponds to a converter between the two protocols. In this way we provide a general formalization and a uniform solution for the protocol conversion problem of [74].

The converter may need state to re-arrange the communication between the original interfaces, in order to ensure compatibility[2]. A novel aspect of our approach is that the protocol converter is synthesized from a specification that says which re-arrangements are appropriate in a given communication context. For instance, it is possible to specify that the converter can change the timing of messages, but not their order, using an n-bounded buffer, or that some messages may

[2]Hence the notion of protocol converter can be seen as a special case of the notion of behavior adapter introduced by Sgroi et al. [81] to characterize a modeling approach for communication-based design that is the basis of the Metropolis framework [6].

be duplicated.

1.4 Scope and Principles

The main objective of our work is to provide a mathematical framework that can be used to reason about and relate many different models of computation and forms of abstractions. A typical model consists of several components. Some syntax is used to describe the structure and the functionality of the design. The syntax includes operations that allows the designer to construct the structure of the design from smaller pieces. A semantic function is used to map the elements of the syntax to elements of the semantic domain, where an equivalent set of functions and relations is defined to parallel the ones of the syntax. The semantic function is typically such that the operations on the syntax are preserved across its application to the semantic domain. In our work, we concentrate on the semantic domain and on the relations and functions that are defined on the domain. In particular, we emphasize the relationships that can be constructed between different semantic domains, and how these relationships affect the functions defined on the domain. This work is therefore independent of the specific syntaxes and semantic functions employed. Likewise, we concentrate on a formulation that is convenient for reasoning about the properties of the domain. As a result, we do not emphasize finite representations or executable models. This and other aspects are deferred for future work.

Section 1.2 above outlined the importance of using different abstraction mechanisms in a design. The key to flexibly using abstractions is a framework that does not force too much detail in the models, but at the same time allows one to express the relevant details easily. There is therefore a trade-off between two goals: making the framework general, and providing structure to simplify constructing models and understanding their properties. While our notion of Agent Algebra is quite general, we have formalized several assumptions that must be satisfied by our domains of agents. These include assumptions about the monotonicity of certain operators. In the case of trace-based algebras, we use specific constructions that build process models (and mappings between them) from models of individual behaviors (and their mappings). These assumptions allow us to prove many generic theorems that apply to all semantic domains in our framework. In our experience, having these theorems greatly simplifies constructing new semantic domains that have the desired properties and relationships. Thus, while generality allows us to encompass a wide variety of different models and abstraction techniques, including different models of time and models of concurrency, structure allows us to prove results that apply to all the models constructed in the framework, and gives us mathematical “tools” that help build new models from existing ones. One objective of our work is therefore to provide a trade-off between generality and structure that works well in most situations. Because of its structure, our framework is capable of more than just classifying existing models of computation.

The ability to exploit structure to derive general results in the framework is an important aspect of our work. The structure that we impose is in fact sufficient to study approximations between models (see section 2.6), and to derive necessary and sufficient conditions for applying such techniques as refinement verification (section 3.3) and local specification synthesis (section 3.4). In the latter case, we are also able to derive an algebraic formulation of the solution that is independent of the particular model of computation in question. This is a strong result that unifies the different approaches to deriving implementation flexibility, as explained in section 1.3 and subsection 1.8.11. It must be pointed out, however, that our solution is purely algebraic and makes no assumption with regard to the implementation of the operators in general, and with the complexity of computing the expression in particular. Nonetheless, our formulation guarantees the correctness of the solution whenever a finite representation of the model and of the operators is available. Having done the theoretical work upfront thus allows the designer of the model to concentrate on improving the efficiency of the implementation. This often requires tuning the model to account for particular situations where the computation may be easier by taking advantage of additional assumptions. In this case, the conditions that we provide for the correctness of the solution can help the designer more quickly identify the changes that must be applied to a model in order to achieve higher efficiency.

For every semantic domain we require that certain functions be defined to formalize concepts such as composition, scoping and instantiation. The specific definition of these functions depends upon the particular model of computation being considered. Nevertheless, we do require that certain assumptions, in the form of axioms that formalize the intuitive interpretation of the operators, be satisfied. A model of computation fits in our framework if and only if it satisfies the axioms. Thus, we employ an *axiomatic* approach, as opposed to a *constructive* one. In a constructive approach, a model possesses certain properties because of the way it is constructed. This could be advantageous in certain situations, especially because one need not verify that the properties or assumptions are true when an object is constructed according to the rules. The axiomatic approach however provides us more flexibility in constructing different models, and clearly highlights the conditions under which the techniques presented in this work can be applied. The axiomatic and the constructive approach may also be combined, as proposed in chapter 4.

The properties that we require of the basic operators of composition, scoping and instantiation of agents are inspired by the *circuit algebra* proposed by Dill [34]. Other properties and operators are also possible. In particular, we do not present a complete treatment of the sequential composition operator, and do not investigate in detail the distinction between partial and complete behaviors [12]. Our choice of operators works well for most of the models of computation in use for embedded systems, and it simplifies the presentation of the theory. In particular, the axioms ensure that each operators performs one, and only one function. This separation of concerns is enforced throughout our work. For example, the parallel composition operator is limited to combining the behaviors of two agents, without altering the visibility of their internal signals. If one wishes to hide its internal structure, the composition operator must be explicitly followed by a scoping, or projection operation. This is unlike other models, like CCS [67, 68], that combine the operation of composition with that of hiding (in fact, removing) internal transitions.

Similarly, the communication in a parallel composition typically occurs by equating signals (or other distinguished features) that share the same name and that belong to different agents. This is especially true for our trace-based models (see chapter 4). This is unlike other frameworks that use an explicit interconnection operator to specify the topology of the system [30]. The explicit operator however essentially combines the instantiation of an agent with its interconnection, and we therefore do not consider it fundamental. The choice of such a simple communication mechanism is deliberate, and is a consequence of the communication-based design paradigm. In fact, if the model requires a more complex interaction, additional agents can be used to model the presence of a communication medium through which the interaction takes place. For example, an explicit interconnection can easily be simulated by introducing an additional agent that works as an identity while enforcing the required topology. Doing so allows us to use the full modeling power of agents to describe interactions that could potentially involve complicated protocols. This paradigm is also consistent with the Metropolis project. Furthermore, we have found that by describing the models at their natural level of abstraction, the simple form of composition is sufficient in most circumstances.

Our models typically include an order on the agents to represent a relation of substitutability or refinement. As discussed above, the operators of the algebra are required to be monotonic relative to the order on the agents. This condition is, in fact, fundamental to the application of compositional methods. To apply these methods, we consider different alternative definitions that extend the notion of monotonicity to functions which, like the operators of our algebras, are not total. We then adopt a notion of monotonicity, called $\top$-monotonicity, that is consistent with the interpretation of the order as a relation of substitutability. The ramifications of this choice can be observed throughout our work. In particular, we derive the compositionality principle that is consistent with our interpretation of the order.

The notion of substitutability is also formalized as a relation that involves evaluating the effects of the agents under different contexts. We call this relation a *conformance order*. Under certain conditions, and when the operators are monotonic, it is possible to restrict the number of contexts that must be considered to determine the conformance relation. In particular, we are interested in a characterization of the conformance order that relies on a simple parallel composition of each agent with another agent, called its *mirror*. Intuitively, the mirror of an agent represents its worst possible environment in relation to the conformance order. The existence of mirrors in a model of computation allows us to apply several different techniques, from refinement verification to local specification synthesis. It is for this reason that we study specific constructions, such as trace-based agent algebras, that guarantee the existence of a mirror function.

1.5 Major Results

The major contributions of this work are listed below.

- *Agent Algebras*, which provide general and powerful tools to construct agent models.
- Particular examples of agent algebras for common models of computation used in the design of embedded systems.
- A set of sufficient conditions for the normalization of expressions involving agents and the operators of the algebra.
- An extended notion of monotonicity for partial operators (in particular, the notion of $\top$-monotonicity) and its consequences on compositional methods.
- A complete characterization of the relationships between the notion of a conservative approximation and that of Galois connection and abstract interpretation.
- The use of conservative approximation to construct hierarchy of models and to formalize the concept of platform-based design.
- Particular conservative approximations in the form of the axialities of a relation between elements of a model of computation.
- The formalization of the concept of *mirror function* and its item complete and general characterization and construction in terms of conformance orders and compatibility.

- Particular definitions of mirror functions for trace-based models.
- A characterization of the relationships between mirror functions between an algebra and its subalgebras.
- A general solution of the local specification synthesis problem.
- An application of the local specification synthesis technique to solve a protocol conversion problem.

1.6 Motivating Example

So far, we have discussed the considerations that influenced our framework for formally modeling heterogeneous systems. Now we can give an informal overview of the framework before describing it formally in the remaining chapters. We do so by presenting an example that motivates the requirement for our framework to support multiple models of computation during the design process. Our exposition in this introduction and in the rest of this dissertation will focus on the definition of natural semantic domains, and their representation in our framework, for the set of models of computation used in the example.

The example, shown in figure 1.3, is an abstracted version of the PicoRadio project [77], developed at the Berkeley Wireless Research Center. A PicoRadio is a node in a network that exchanges information with its neighboring nodes. Depending on the application, a PicoRadio may function as the intercom end of a communication system, or as a controller for a set of sensors and actuators. Whatever its function is, the PicoRadio must include several subsystems, as shown in figure 1.3. Since communication with neighboring nodes occurs on a wireless link, a Radio Frequency (RF) subsystem is used to interface the design to the channel. Demodulation and decoding is done at the baseband level, after conversion from the high transmission frequency. The data streams obtained from the baseband is interpreted by a protocol stack, which feeds the application that ultimately interfaces with the user.

The design of such systems is complex, not so much in terms of their size, but because of the very stringent constraints on power and because of the intrinsic interactive nature of the nodes. Together, they call for a new design methodology and indeed, developing the new methodology was the primary task during the design of the first version of the PicoRadio ([26]). Because power concerns are best attacked at the algorithmic level, new protocols are being devised whose primary

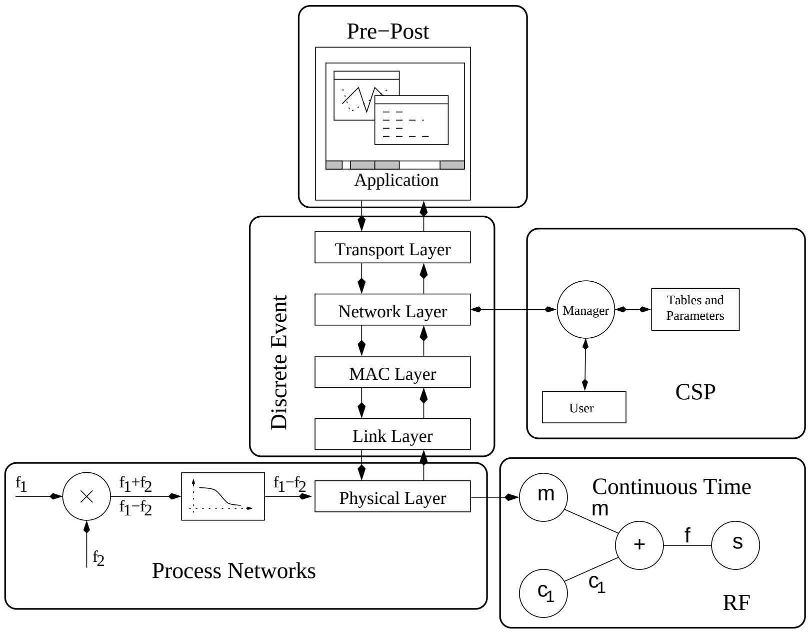

Figure 1.3: Full system

purpose is to maximize the up-time of the system. Consequently, the interaction between the different subsystems becomes critical.

Each subsystem must be described in some model of computation in order to properly verify its function through simulation and verification. Ideally, for each subsystem, we would like to use the model that is best suited for the particular task. Hence, the design flow often includes several different tools and models that offer characteristics appropriate to the specific subsystem being considered. In practice, however, the segmentation of the design process that results makes the interaction between different subsystems and the consequences of the design choices difficult to analyze. Typically, the solution to the problem involves simplifying the interfaces between subsystems by assuming certain timing behaviors. However, this not only may not be possible in certain situations, but it also amounts to working at a lower level of abstraction where the benefits of an application specific model could be diminished or lost.

The interaction between different models of computation can be understood when the description of the models is embedded in the same unifying framework. Agent algebra is one such framework. In this introduction we present the basic concepts and definitions by way of an example, i.e., the formalization of a semantic domain suitable for the representation of behaviors in a model of computation that supports continuous time. In particular, we will employ the techniques and notation introduced in chapter 4, which constitute a particular class of agent algebras. We first present a formalization that can be considered *natural* for the domain of application, and then show how the same can be cast in terms of a general set of definitions.

1.7 An Example Agent Algebra

This section presents a simple formalization of a model of computation that relies on equations to express the relationships between the quantities that occur in the model. This is only *one specific* example of several possible models of computation that fit in our framework. In particular, to make our presentation more intuitive, we construct the model using the trace-based technique described in chapter 4 instead of the fully general agent algebra introduced in chapter 2. Other examples using both techniques will be presented in the rest of this work.

More specifically, we are interested in a model of computation where the quantities (variables) are functions over the set of reals. By convention, it is assumed that the set of reals represents time, and we talk about functions over time. Consequently, the equations we are interested in are relations on functions over time, and we denote the independent variable with the letter t. This model is therefore particularly indicated as a representation of the continuous time component of the system shown in figure 1.3.

Consider the following equation:

$$x = 3t. \tag{1.1}$$

This is an equation in the unknown x. Traditionally, the interpretation of the equation is done in terms of the set of possible solutions. In our case, the set consists of functions that are associated to the variable x. A function $x : \mathbb{R}^{/} \to \mathbb{R}$ is a solution of the equation if, when substituted for the unknown, the resulting relation is true. The notation $\mathbb{R}^{/}$ denotes the set of non-negative reals (we use non-negative reals because we assume there exists an initial point in time). In this particular case there is only one solution

$$x(t) = 3t.$$

In our framework we want to make the interpretation in terms of the set of solutions more precise. More specifically, we would like to define a collection of mathematical objects that represent the set of solutions of an equation. A semantic function shall then be used to associate the correct solution with an equation.

In the first place, we must associate with each agent the alphabet $A \subseteq \mathcal{A}$ of the variables it uses, where $\mathcal{A}$ represents the set of all possible variable names. An agent is also characterized by a *signature*, which we denote with the symbol γ. The structure of the signature depends on the particular model of computation, and it uses the symbols in the alphabet to model the visible *interface* of the agent. For the equation in the example above, the alphabet A consists of the names of the variables that appear in the equation: $A = \{ x \}$. Note in particular that t is not included in the alphabet, because of its special role as an independent variable. Note also that the equation simply describes a condition for a function to be a solution. Therefore, when constructing a model for an agent represented by continuous time equations, we do not specify the direction of the signals (input or output), but simply associate a set of signals to each agent. Hence, the signature for agents in the continuous time model of computation simply consists of the set of symbols A: $\gamma = A$.

Consider again equation 1.1. As mentioned, we interpret the equation (agent) as the set of its possible solutions. In turn, we may interpret each solution as one possible *behavior* of the agent. In the specific case of functions over time, an individual execution is a set of functions, one for each unknown in the equation (a singleton in our example, since there is only one dependent variable). An agent is a set of sets of functions.

We define behaviors in the framework of agent algebra for the continuous time model to be a close formalization of the natural interpretation of a solution. However, we should make the relationship between the solution and the variables precise. In what follows, and to be consistent with the terminology that will be introduced in chapter 4, we will refer to a behavior as a *trace*. Because the definition of a trace must be independent of the particular agent, it must take the alphabet as a parameter. In the case of the continuous time model of computation, we must assign a function over the reals to each of the symbols in the alphabet. For our example, we could use traces of the form

$$A \to (\mathbb{R}^{\not-} \to V)$$

where the set V is the range of the functions. In our case we have $V = \mathbb{R}$. A trace thus contains both the solution, and the association of each of the functions in the solution to the variables that appear in the equation. A trace could therefore be expressed as a function $f : A \to (\mathbb{R}^{\not-} \to V)$. Note that the domain of this function is the set of symbols in the alphabet. For each symbol, the function f associates a function over the independent variable t. For the example above, the (only) solution can be expressed as the function

$$f(x) = \lambda t\ [3t].$$

Note that there might be several possible valid definitions of a trace. For example, the reader may find it more convenient to define a trace as a function that associates to each moment in time the values of the functions. The traces thus become functions of the form

$$\mathbb{R}^{\not{}} \rightarrow (A \rightarrow V).$$

The choice of alternative, isomorphic, definitions is often a matter of convenience in defining the operations that we discuss below, or it might reflect the desire to highlight certain aspects of the behavior.

In this particular example, the equation admits only one solution. More generally, equations may have several solutions. Consider for example the modified equation

$$x = 3t + x_0.$$

In this case the solution varies according to the values of the parameter x_0. We say that a function is a solution to the equation if there exists a value of the parameter such that the equation is satisfied. The solutions thus form a set. Since traces are individual solutions, agents must therefore be represented using sets of traces. More formally we can write the denotation of the equation above as

$$P = \{\, f : A \rightarrow (\mathbb{R}^{\not{}} \rightarrow V) : \exists x_0[f(x) = \lambda t[3t + x_0]\}.$$

Hence, a model of an agent, which we call a *trace structure*, consists of the signature γ and of a set P of traces. We usually denote the trace structure as the pair $p = (\gamma, P)$.

Systems of equations do not present any additional problem. Consider for example the system

$$\begin{aligned} x &= 3t + x_0, \\ y &= 4t + y_0. \end{aligned}$$

In this case we define the alphabet as the set $A = \{\, x, y\}$ and the signature as $\gamma = A$. However, the definition of a trace and of a trace structure is unchanged. A trace is again a function from the alphabet to the set of functions on time, and a trace structure is the signature together with a set of traces. What changes is the set of traces for this particular agent, which is now expressed as the set

$$P = \{f : A \rightarrow (\mathbb{R}^{\not{}} \rightarrow V) : \exists x_0, y_0[f(x) = \lambda t[3t + x_0] \wedge \; [f(y) = \lambda t[4t + y_0]\}.$$

Systems of equations can have two interpretations. On the one hand, they represent an agent whose constraints on the variables are expressed by different equations. On the other hand, they may represent the interaction of different agents, each represented by disjoint subsets of the equations in the system. The two views can be reconciled by interpreting the agent as a whole as the result of the interaction of the individual agents. We discuss the details of this interpretation after we introduce the relevant operations on traces.

Once we have established the notion of a trace and of a trace structure, the complexity of the equation doesn't really matter. In fact, we are not interested in *solving* the equation, but in providing a structured semantic domain for its *interpretation*. In particular, interpreting differential equations is no more complex than interpreting the simple linear equations shown above. As an example, consider the following differential equation

$$\frac{d^2 s}{dt^2} + g^2 s = 0.$$

This is a homogeneous second order differential equation in the variables g and s. The solutions of this equation describe an oscillatory behavior. In fact, this equation might be used to model an oscillator that generates a signal (for example a voltage) that we denote by the symbol s, whose frequency is controlled by another signal, denoted by the symbol g. Solutions to this equation are in the form of pairs of functions $s : \mathbb{R}^{\not+} \rightarrow V$ and $g : \mathbb{R}^{\not+} \rightarrow V$.

In general, solutions to differential equations depend on arbitrary parameters, whose value can be fixed by providing appropriate initial conditions. For instance we might require that

$$\begin{aligned} s(0) &= 1, \\ \frac{ds}{dt}(0) &= 0. \end{aligned}$$

Given the initial condition, one possible solution to this equation is the following pair of functions:

$$\begin{aligned} s(t) &= \cos(10t), \\ g(t) &= 10, \end{aligned}$$

which represents a constant oscillation with frequency 10 radiants per second. An agent, the denotation of the differential equation, is a set of individual executions, i.e., the set of all possible solutions. In our example, the trace structure has alphabet $A = \{ s, g \}$ and signature $\gamma = A$. The trace structure $p = (\gamma, P)$ is such that P is the set of traces that satisfy the equation. Note that in the definition above the trace structure doesn't include an initial condition: this is intentional, as we want the trace structure to model all possible solutions. Initial and boundary conditions, if any, arise implicitly as a result of the interaction (parallel composition) of different trace structures.

1.7.1 Operations on Behaviors and Agents

To complete our overview we define the operations on individual behaviors and on agents. These operations are defined to support common tasks used in design, like that of scoping, instantiation and composition of agents.

The *projection* operation removes from a trace all information related to certain signals. In our example of functions over time, this corresponds to retaining only the functions of interest (for instance s) in the solution, and dropping the others (g is our case). If $B \subseteq A$ is the set of signals that we want to retain, we define the projection as a restriction on the domain of the functions that characterize a trace x. Formally we write:

$$\mathit{proj}(B)(x) = \lambda t \in \mathbb{R}^{/}\, \lambda a \in B[x(t,a)],$$

where the λ notation introduces a function of the named variable, as usual. Projection on trace structures (agents) can be seen as the natural extension to sets of the corresponding operation of projection on individual traces. When applied to agents, the operation of projection corresponds to that of hiding internal variables in the equation. Note that the constraints imposed by the equation on the variables are retained, but their effect is only visible from outside through the remaining signals. In other words, the scope of the hidden variables is limited to the equation they belong to.

The *renaming* operation changes the names of the visible elements of the agent. If r is a renaming function, we define renaming on traces as the corresponding operation on the signals in the signature. Formally:

$$\mathit{rename}(r)(x) = \lambda t \in \mathbb{R}^{/}.\lambda a \in A.x(t, r(a)).$$

For functions over time, this corresponds to a substitution of variables. Substitution, however, must be done carefully to avoid changing the underlying meaning of the equation. The restriction that r be a bijection avoids conflicts of names that could potentially change the behavior of the agents. As for projection, renaming of trace structures can be seen as the natural extension to sets of the corresponding operation on individual traces. When applied to a trace structure, the effect is that of a renaming of the variables in the corresponding differential equation. This process corresponds to that of *instantiation* of a master agent into its instances.

Projection and renaming, seen as operators for scoping and instantiation, are common operations that are meaningful to all models of computation. For trace structures, they are always defined as the natural extension to sets of the corresponding operations on traces. The combination of the set of traces x for all alphabets A, and of the operations *proj* and *rename* has the structure of an algebra. We call this algebra a *trace algebra*, and we usually denote it with the symbol $\mathcal{C}$.

Similarly to trace algebra, the combination of all trace structures p and the operations of projection and renaming on trace structures form an algebra, that we call *trace structure algebra*. In addition to the operation on traces, a trace structure algebra includes the operation of *parallel composition* of agents. A system of equations is an example of a parallel composition in our model of computation based on continuous time. Here, each equation is interpreted as a single agent. The system is also interpreted as an agent, the one that is obtained by composing the individual agents. An example of a system of equations is the following:

$$\begin{aligned}
\frac{d^2s}{dt^2} + g^2 s &= 0 \\
\frac{d^2m}{dt^2} + I_1^2 m &= 0 \\
g &= m + I_2 \\
I_2 &= 3; \\
I_1 &= 2.
\end{aligned}$$

In the natural semantic domain, the agent that corresponds to the system of equations is made of collections of functions that are solutions to *all* equations. Intuitively, this corresponds to having the agents associated to each equation run concurrently by sharing the common signals.

We can easily formalize this notion in the framework of trace algebra. Let $p_1 = (\gamma_1, P_1)$ and $p_2 = (\gamma_2, P_2)$ be two trace structures, and denote with $p = p_1 \parallel p_2$ their parallel composition. Clearly, to model this composition, the signature of p must include the signals of both p_1 and p_2. Hence:

$$\begin{aligned}
A &= A_1 \cup A_2, \\
\gamma &= \gamma_1 \cup \gamma_2.
\end{aligned}$$

The set of traces P of p must be such that each trace belongs to both p_1 and p_2. However the traces must first be converted from one alphabet to another. This can be achieved by first extending the set of traces P_1 and P_2 to P_1^e and P_2^e, respectively, which are sets of traces over the alphabet A such that

$$\begin{aligned}
P_1^e &= \{x : \mathit{proj}(A_1)(x) \in P_1\} \\
P_2^e &= \{x : \mathit{proj}(A_2)(x) \in P_2\}.
\end{aligned}$$

The traces in P_1^e clearly satisfy the system of equations for p_1 (the additional functions are simply ignored), but do not necessarily satisfy that for p_2. Likewise, the traces in P_2^e satisfy the equation for p_2 but do not necessarily satisfy that for p_1. The parallel composition is the set of those traces that satisfy both,

$$P = P_1^e \cap P_2^e.$$

Given this definition, it is straightforward to show that parallel composition corresponds to the usual operation of taking the intersection of the solutions of two different equations. Consider again the system of differential equations in equation 1.2. This system can be represented as the parallel composition of 5 trace structures[3], as shown in figure 1.4, where the rounds represent trace structures, and the connections represent shared signals (functions over time). The signature of the parallel composition is

$$A = \{ I_1, m, I_2, g, s\}.$$

Each trace structure imposes its constraints to the overall solution. For example, if x is a trace with alphabet A, then the trace structure for I_2 requires that *proj*$(\{ I_2\})(x)$ be the function identically equal to 3.

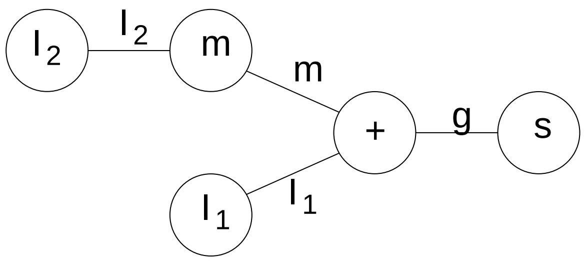

Figure 1.4: Parallel composition of agents

The definition of parallel composition of agents shown above for the continuous time model of computation can be generalized in a straightforward way in our framework. Parallel composition corresponds to the concurrent execution of two agents. As discussed above, the parallel composition $p = p_1 \parallel p_2$ is a set of traces in the union A of the alphabets of p_1 and p_2 that is "compatible" with the restrictions imposed by the agents being composed. We can formalize the notion of compatibility by requiring that if x is a trace of p with alphabet A, then its projection

[3]Parallel composition turns out to be associative and communative, therefore we can talk about the operation of parallel composition of more than just 2 trace structures.

proj$(A_1)(x)$ on the alphabet of p_1 is in P_1, and the projection *proj*$(A_2)(x)$ on the alphabet of p_2 is in P_2. The set of traces in p must be maximal with respect to that property. It can be shown that the previous definition of parallel composition for the continuous time model is equivalent to this formulation.

The combination of the set of trace structures and the operations of projection, renaming and parallel composition of trace structures forms an algebra that we call *trace structure algebra*. In chapter 4, we will show that a trace structure algebra is a particular case of the more general agent algebra model introduced in chapter 2.

To summarize, the first step in defining a model of computation as a trace-based agent algebra is to construct a trace algebra. The trace algebra contains the universe of behaviors for the model of computation. The algebra also includes two operations on traces: *projection* and *renaming*. These operations intuitively correspond to encapsulation and instantiation, respectively.

The second step is to construct a trace structure algebra. Here each element of the algebra is a trace structure, which consists primarily of a set of traces from the trace algebra constructed in the first step. A trace structure algebra also includes three operations on trace structures: *parallel composition*, *projection* and *renaming*. Projection and renaming are simply the natural extension to sets of the corresponding operations on traces, while parallel composition is derived from the definition of projection on traces. Thus, constructing a trace algebra is the creative part of defining a model of computation. Constructing the corresponding trace structure algebra is much easier.

Equations and systems of equations can be naturally ordered in terms of their solutions. An equation E *implies* another equation E' if the solutions of E are also solutions of E'. This relation translates directly into a relation between trace structures. A trace structure $p = (\gamma, P)$ is *contained* in a trace structure $p' = (\gamma, P')$ if $P \subseteq P'$. This containment relation, which is sometimes called *refinement* in the model, can be applied to all models described as trace structure algebras.

The example of this section shows how to formalize the natural semantic domain of a model of computation based on continuous time and differential equations. It is worth noting how our representation of the agents is completely denotational. In addition, while our formalization is close to the natural semantic domain of traditional differential equations, the algebraic infrastructure introduces additional concepts such as hierarchy, instantiation and scoping in a natural way. For instance, the trace structures that correspond to the oscillators could be viewed as instantiations of a primitive component obtained by a renaming operation. Also, the frequency modulator that results from the parallel composition outlined above could be used as a primitive component: to that end, it is enough to hide the internal signals $\{m, I_2, g\}$ through a projection operation. This fact, and the fact that algebraic equations on functions of real time are a subset of differential equations, makes it possible to construct the trace structure corresponding to a differential equation incrementally, by composing its pieces. In particular, trace structures corresponding to integrators are used to model the relationships between a variable and its derivative, while more conventional trace structures model the application of algebraic operations, like addition, multiplication, exponentiation and so on. This model is widely used to describe DSP applications, and is often referred to as *signal-flow model* [86].

In the rest of this work we will make these notion precise, and we will present the formalization of several other examples of models of computation. In particular, while distinguishing between agents and their individual behaviors is convenient for constructing new models, our main results are based on a less structured agent model, called agent algebra. Trace structure algebra will be shown to be a particular class of agent algebra. Relationships between different models of computation can be obtained as functions that map concrete agents to abstract agents, and vice versa. To be useful, these functions must preserve certain relationships between agents, including, in particular, the containment relationship. In addition, we will show how to characterize the containment relationship in terms of the operators of the algebra, and how to take advantage of this characterization to derive techniques in the area of refinement verification and synthesis.

1.8 Related Work

1.8.1 Algebraic Approaches

Many are the approaches to modeling that use algebraic techniques. The most notable are certainly process algebras CCS and CSP, originally proposed by Milner [67, 68, 69] and by Hoare [50], respectively. There, atomic actions are combined with process variables to form expressions that represent more complex processes. The algebra includes such operators as sequencing, union (representing choice), parallel composition, projection and recursion. A process algebra expression can be interpreted in terms of a labeled transition system that consists of a set of states and a set of transitions that are labeled by the actions that occur in the expression. The operators of the algebra can be directly translated into operators that act upon the transition system. The expression, in fact, is an implicit representation of the transition relation.

There is a fundamental difference between process algebras and the algebras used in our framework. A process algebra is essentially a "language" that can be used to describe specific instances of agents. The definition of the operators also reflects this intent. In our framework, instead, although we also combine agents using algebraic expression, the operators of the algebra are in general uninterpreted. Our knowledge about them is limited to the requirements that we impose to formalize their intuitive meaning. In particular, an expression in our framework does not include a notion of an atomic action and therefore is *not* interpreted as a labeled transition system. By doing so, we are able to uniformly consider models that are not "naturally" expressed in terms of the specific operators of a process algebra, such as those used for hybrid systems and dataflow. The expression, in other words, merely represents the hierarchical structure of the system that is constructed by composing different instances of agents, according to rules that depend upon the particular model of computation that is being considered.

Similar remarks apply to the framework of Abstract State Machines (formerly known as Evolving Algebras) proposed by Gurevich [45, 46]. Abstract State Machines build on the concept of *mathematical structure* [36]. A mathematical structure is the combination of a carrier set A, together with the interpretation of a set of function and relation symbols Υ, called the *signature* of the structure. A structure with signature Υ can be transformed into another structure by modifying the interpretation of the function and relation symbols. These modifications are called *updates* in the terminology of Abstract State Machines. Given a signature Υ, the set of structures over Υ forms a space that is taken as the state space of a class of state machines.

The approach is abstract in the sense that the mathematical structures used as the state space, and the corresponding update operations, can be arbitrarily complex, or arbitrarily simple, thus making it possible to represent computations at different levels of abstraction. Nevertheless, the semantics of the operations is fixed and corresponds to modifying the values of the functions that interpret the symbols in the signature. To put it another way, Abstract State Machines is a parameterized state machine model, where states (the parameters) are first order mathematical structures. This makes Abstract State Machines extremely useful for documenting an algorithm at different levels of granularity and precision. Our emphasis, on the other hand, is on studying the relationships that can be established across these levels of abstraction. For this reason, our operators are only defined axiomatically, rather than constructively. If nothing else, Abstract State Machines could be viewed as one of the models that fits in our framework.

Mathematical structures are also at the basis of the algebraic specification of data structures proposed by Ehrig et al. [39, 40, 41]. The basic idea is to specify data types independently of any specific representation or programming language. In their work, a data type specification SPEC is a tuple (S, OP, E), where S is a set of sorts representing the domain of the data structure, OP is a set of constant and operation symbols that act on the data structure, and E is a set of equations or axioms that provide the description of the operations in a constructive (for equations) or axiomatic (for axioms) way. A SPEC-algebra A is composed of a domain for each sort in S that defines the elements of the data structure, and of an interpretation of the operations in OP (i.e., constants and functions on the domains) that satisfies the equations and the axioms in E. Algebraic specifications can be parameterized by a second algebraic specification, and a mechanism of parameter passing (based on pushouts in an appropriate category) is provided for its actualization. In addition, a wide variety of operators can be defined to transform and combine algebraic specifications. These include products and unions, extensions and restrictions and refinement and implementations.

Our use of the algebraic framework is similar. In our case, however, the algebra is fixed and includes only one sort (the set of agents in a model of computation), and the operators of projection, renaming and parallel composition. The axioms are also fixed, and formalize the intuitive meaning of the operators. The objectives, and the style of presentation, are therefore similar to those of algebraic specifications: to define axiomatically the intended properties of some objects, instead of defining them constructively, for example as a parameterized, but fixed, model (such as, for example, Abstract State Machines, mentioned earlier, or the Tagged Signal Model mentioned below). However, our work concentrates on one particular "data type", that of the models of computation for concurrent systems. Our focus is therefore on the relationships between the objects (instantiations) of the data type, that is the relationships between different models of computation, instead of the relationships between different kinds of data types.

One specific interpretation of an algebraic specification of a data structure is its *initial semantics*, defined as the quotient of the term generated structure relative to the congruence induced by the axioms and the equations imposed by the algebra. This quotient represents the most abstract interpretation of the data structure. We do not consider this problem in our work since it is not consistent with our aims and scope. The problem is however addressed by Dill for the class of circuit algebras [34]. There, he defines a particular interpretation of the circuit algebra, called a *circuit structure*, which he shows isomorphic to the free circuit algebra (its initial semantics). Since the framework of agent algebra is derived from that of circuit algebra, a similar result could be expected to hold in our framework, as well.

On a different level, we can view algebraic specifications of data structures as part of particular models that fit in our framework. In this case the agents themselves are composed of algebraic specifications of data structures, which are used as a language to describe their internal structure. In other words, algebraic specifications of data structures may form the basis for the description of agents. The framework of agent algebra, on the other hand, could be used as the basis for their interaction. Indeed, algebraic specification of data structures must be used in conjunction with a concurrency model in order to describe the *behavior* of a system, as exemplified by the language LOTOS [41] which is based on a process calculus derived from CCS [68]. The relations between the different data structures could be used as the basis for the construction of conservative approximations between different models, that is of functions that preserve the refinement relation in the models across the boundaries of the model of computation. We do not, however, address this particular case in this work.

1.8.2 Trace Theory

Much of our inspiration comes from the work of Dill [34] and Burch [12]. In this comparison we highlight the extensions and the additional results and insights that were obtained by generalizing their work.

Dill introduces a model where each execution is modeled by a sequence of actions, called a *trace* [34]. Agents in the model, called *trace structures*, consist of two sets of traces, representing valid and invalid executions, respectively. The trace structures, together with their operations, form a *circuit algebra* and satisfy certain basic properties that formalize their intuitive interpretation. Later, Burch generalized the model by considering abstract executions rather than a specific trace model [12]. The term “trace” is here retained to denote arbitrary objects that together with their operations satisfy the axioms of *trace algebra*. To simplify the presentation, trace structures are obtained simply as sets of traces, rather than two sets of traces. The generalized trace structures are again shown to form a circuit algebra, there called *concurrency algebra*.

In our work, we further generalize the approach of Burch, and consider directly an algebra of arbitrary agents, instead of arbitrary executions. Our contribution consists in part in rephrasing many of concepts introduced by Dill and Burch in a more general setting. Free of the constraints of a specific model of computation, we expose and derive the conditions for applying certain techniques, such as conformance and conservative approximations, and therefore provide insights into the way they operate.

In particular, we generalize the notion of *conformation ordering* [34] and parameterize the concept of *failure freedom*. Additionally, we derive necessary and sufficient conditions for the existence of a mirror function. These generalization allow us to state and solve the problem of local specification synthesis, in a form similar to that already suggested by Burch et al. [13]. Because of the more general setting, however, our result is independent of the particular model.

The concept of a conservative approximation in our framework is derived from the one introduced by Burch [12]. Here we decompose the definition to highlight and discuss its compositionality properties, and study its relationship with traditional notions of abstraction, such as Galois connections and abstract interpretations. In addition we further characterize the inverse of a conservative approximation, and use it to define interactions between different models of computation. We also extend trace structures to two sets, and derive a mirror function for the appropriate notion of conformance based on failure freedom. We use this derivation to study the problem of protocol conversion.

Dill and Burch distinguish in their work between partial and complete traces. In particular, Burch provides a set of additional axioms that formalizes the intuitive interpretation of the concatenation operator, which can be applied to partial traces. In our work we only informally consider this distinction in the examples, and reserve a detailed treatment, including a revision of the axioms, for our future work.

1.8.3 Tagged Signal Model

Several formal models have been proposed over the years (see e.g., [37]) to capture one or more aspects of computation as needed in embedded system design. Many models of computation can be encoded in the Tagged-Signal Model [62]. In the Tagged-Signal Model (TSM), a model of computation is constructed in a fixed way by considering a set of values V, and a set of tags T. The set of values represents the type of data that can be exchanged by objects in the model. The set of tags, on the other hand, carries an order relationship that is used in the model to encode the particular notion of time, or, more properly, of precedence. An *event*, that is the change of a value in the system, is represented by the pair $\langle t, v \rangle$, where $t \in T$ tags the "time" of the event, and $v \in V$ provides the new value. A *signal* is an arbitrary collection of events, i.e., a subset of $T \times V$. Since an order on events can be derived from the corresponding order on tags[4], a signal can be seen as the evolution in time of a set of values. Signals are organized in tuples, where each element of the tuple corresponds to a particular "port" of an object. A tuple of signals is therefore the behavior that can be collectively observed at a set of ports. Finally, a set of tuples of signals is called a *process*, and represents the possible behaviors of an agent of the model.

[4]Strictly speaking the order induced on events is a preorder, since the condition of antisymmetry cannot be guaranteed.

The operations of projection and renaming are defined on the tuples as expected, by removing a component of the tuple or by exchanging their position. The operation of parallel composition is obtained by an initial inverse projection of the processes to the common "alphabet" (strictly speaking the alphabet here is positional in the tuple), followed by an intersection of the signals.

The Tagged-Signal Model is similar to our trace-based agent algebras. In particular, the operations on agents are defined in almost exactly the same terms (although we use names for signal, instead of a positional notation). However, because we do not dictate the structure of a trace, but only its properties relative to the operators, our approach seems to be more general. In addition, the extra flexibility allows us to construct a representation that is closer to the "natural" semantic domain of the model.

We are not aware to date of a general theory that explains the relation between different models encoded in the Tagged-Signal Model. Nonetheless, the work of Benveniste et al. [11] can be seen as an attempt in that direction. In this work the authors study Globally Asynchronous Locally Synchronous systems by considering sets of tags with different ordering relationships. Morphisms on the sets of tags are used to relate the different models. The results include conditions on a correct deployment of synchronous systems over asynchronous communication channels.

Informally, we can see morphisms on tags as corresponding morphisms on sets of traces, or on agents in an appropriate agent algebra. It would be interesting to derive abstraction and refinement relationships between the tagged systems in a way similar to our conservative approximations. This is part of our future work.

1.8.4 Ptolemy II

The study of systems in which different parts are described using different models of computation (heterogeneous systems) is the central theme of the Ptolemy project [27, 28]. Our work shares the basic principles of providing flexible abstractions and an environment that supports a structured approach to heterogeneity. The approach, however, is quite different. In Ptolemy II each model of computation is described operationally in terms of a common executable interface. For each model, a "director" determines the order of activation of the agents (a.k.a. actors; for some models, the actors are always active and run in their own thread). Similarly, communication is defined in terms of a common interface. A model of computation, or *domain*, in Ptolemy II is a pair composed of a director together with an implementation of the communication interface, called a "receiver". The domain defines the scheduling of the actors, the communication scheme and the possible interactions with other models of computation. On the other hand, we base our framework on a denotational representation and de-emphasize executability. Instead, we are more concerned with studying the process of abstraction and refinement in abstract terms. For example, it is easy in our framework to model the non-deterministic behavior that emerges when an abstract model is embedded into a more detailed model. Any executable framework would require an upfront choice that would make the model deterministic, potentially hiding some aspects of the composition.

The approach to heterogeneity in Ptolemy II is strictly hierarchical. This implies that each node of the hierarchy contains exactly one domain, and that each actor interacts with the rest of the system using the specific communication mechanism selected by the domain for the hierarchy node it belongs to. Domains only interact at the boundary between two different levels of the hierarchy. In contrast, our approach to heterogeneity is based on explicitly refining heterogeneous models into a third model that is detailed enough to exactly represent the initial two. In addition, the framework also supports agents that use several interaction mechanisms at the same time and at the same level of hierarchy. The conjunction of these mechanisms can be seen as a new model of interaction. The proliferation of models that results is intentional. For this reason our framework must provide tools that make it easy to construct models of computation and to study their relationships. It is arguable that hierarchical heterogeneity is a more structured approach. Such structure may be desirable in a simulation context. However, we believe that in the context of formal models that support clean mixing of models of computation, the clearest descriptions sometimes require more flexibility than hierarchical heterogeneity. One example are the transducers of a hybrid model that translate from the digital to the analog domain, and vice versa. Interestingly, these cases are also handled by special transducers in the Ptolemy II framework, which appear to relax the requirement for strict hierarchical heterogeneity.

One of the innovative concepts in the design of the Ptolemy II infrastructure is the notion of *domain polymorphism* [64]. An actor is domain polymorphic if it can be used indifferently under several directors, and therefore models of computation. To check whether an actor can be used under a particular model, the authors set up a type system based on state machines, which is used to describe the the assumptions of each director and each actor relative to the abstract semantics (i.e., it describes the subset of sequences of actions in the abstract semantics that are admissible for a certain object).

We also introduce a similar notion. In our framework, an agent can be used in different models of computation if it has an exact representation is such models. The notion of abstraction in the form of a conservative approximation and its inverse provides us with the appropriate interpretation of an agent from one model in another model . An agent is polymorphic precisely when this interpretation is exact. This has the advantage of making the process of abstraction and refinement of an agent explicit.

1.8.5 Abstract Interpretations

Abstract interpretations are a widely used means of relating different domains of computation for the purpose of facilitating the analysis of a system [22, 23]. An abstract interpretation between two domains of computation consists of an abstraction function and of a concretization function that form a Galois connection. The distinguishing feature of an abstract interpretation is that the concretization of the evaluation of an expression using the operators of the abstract domain of computation is guaranteed to be an upper bound of the corresponding evaluation of the same expression using the operators of the concrete domain.

Our notion of conservative approximation is closely related to that of an abstract interpretation, and a detailed account of the similarities and differences is presented in subsection 2.7.1. In particular, the upper bound of a conservative approximation and the inverse of the conservative approximation form, in some cases, a Galois connection and/or an abstract interpretation. Conversely, the lower bound of a conservative approximation does not have an analogue in the theory of abstract interpretations. Nonetheless, in subsection 2.7.1 we show that the lower bound of a conservative approximation can be explained as the concretization map of another Galois connection, one that goes from the abstract to the concrete model. A conservative approximation is thus composed of *two pairs* of related functions, instead of just one, and are used in combination to derive stronger preservation results. In particular, by applying one pair to the implementation and the other to the specification, we are able to not only guarantee that certain properties are preserved from the abstract to the concrete domain, but also that a refinement verification result is preserved in the same direction. To our knowledge, for abstract interpretations a positive refinement verification result in the abstract domain implies a positive verification result in the concrete domain only if there is no loss of information when mapping the specification from the concrete domain to the abstract domain. Thus, conservative approximations allow non-trivial abstraction of both the implementation and the specification, while abstract interpretations only allow non-trivial abstraction of the implementation.

One example of how additional structure allows one to derive stronger results is the work presented by Loiseaux et al. [65] on property-preserving abstractions. The authors consider a specific model based on transition systems. In particular they show that the traditional notion of simulation based on a binary relation between the states of two transition systems (one concrete and one abstract) can be rephrased in terms of a Galois connection between the powerset of the state spaces. There is an isomorphism between the powerset of the set of states and the set of properties of a transition system. Therefore, the Galois connection, and by transitivity the simulation relation, can easily be seen as a property transformation. The detailed knowledge of the model allows the authors to not only state results in terms of property preservation of the functions involved, but also to determine the particular abstract transition system that exhibits the most properties of the concrete transition system under study.

The techniques presented in [65] are applicable, to some extent, to trace-based agent algebras. In fact, the correspondence between a simulation relation and a Galois connection is one instance of an *axiality*, that is of the one-to-one correspondence that exists between a binary relation over two sets and a Galois connection between their powersets [42]. As proposed already in [12] (although not exactly in these terms), a conservative approximation is constructed as an axiality, where a relation between the set of traces, i.e., the trace algebra homomorphism, is converted into a Galois connection on the powersets of traces, i.e., the upper bound of the conservative approximation on trace structures with its inverse. The homomorphism is a particular kind of relation on traces, one that is sufficient to guarantee the compositionality properties of the conservative approximation. The lower bound of the conservative approximation is also derived as an axiality, this time from the inverse relation on traces, i.e., the inverse homomorphism. It is easy to show that this axiality is related to the forward relation by a double complementation of the sets involved, as the formulation proposed in [12] already suggests.

The homomorphism, similarly to the relation on the sets of states, defines a notion of "simulation relation" between sets of traces, or trace structures. Given a trace algebra homomorphism, it would be interesting to address the problem of finding the most faithful approximation of a trace structure, as described in [65]. Our framework provides the additional flexibility of leaving the definition of what is an admissible agent to the designer of the model of computation. The problem to be solved is that the simple extension of the homomorphism to sets of traces may not yield a trace structure in the abstract domain, since the trace structure algebra does not in general include *all* trace structures (despite being closed under the operations). The tightest conservative approximation proposed in [12] can be seen as one step in that direction, at least in terms of the lower bound. However, we do not address the general problem here, and we reserve it for our future work.

Note further the significant difference between a relation on states and a trace algebra homomorphism. In the first case, the axiality gives an abstraction that to an agent p associates a more abstract agent p', where p and p' are part of the *same* model of computation. In contrast, a trace algebra homomorphism, by relating two possibly different algebras, is able to change the level of abstraction of the model. The proper notion of homomorphism in trace algebra that doesn't change the level of abstraction would therefore be that of an endomorphism. Indeed, the projection and renaming operators on traces are trace algebra endomorphisms. Their axialities are the corresponding operations of projection and renaming on trace structures, together with the inverse projection and inverse renaming which are the adjoints function of the Galois connection (note, also, that renaming is typically an automorphism, i.e., an isomorphism of trace algebra that does not give rise to any abstraction). The reason why we consider these as basic operations of the model, rather than as abstractions, is that in this way we are able to construct expressions that describe the hierarchical structure of a system of components in an intuitive, and almost graphical sense.

The authors of [65] go even further, and derive compositionality rules for abstractions obtained using a different relation on states for each component. This translates into using different homomorphisms of trace algebra for each agent. This is a potentially very interesting technique, especially in the framework of trace-based agent algebras, where the different homomorphisms could translate different agents into yet different models of computation. In this way, it would be possible to "tune" the abstraction of each component of a system for a specific problem. The definition of compositionality must however be adapted to a new situation, one where the different agents being composed reside in different models. While far from solving the problem, we move in that direction by considering the issue of co-composition.

Similar results could be explored in the more general setting of agent algebra (as opposed to trace-based agent algebra). Since in this case we do not "look" inside the agents, our knowledge of the model is limited to the axioms of the algebra, and the condition of monotonicity. Nonetheless, we can represent properties of agents as the collection of agents that satisfies them, essentially relying again on powersets as done in the case of states and in the case of traces. We do not however attempt to formulate the problem here, and we reserve it for our future work.

As noted above, the axiality that corresponds to the inverse of a relation is useful to obtain a lower bound of a conservative approximation. Although in agent algebra we do not work with powersets (and with the corresponding notion of complementation) we can still use the mirror, when it exists, as a complementation operator. In section 3.5 we explore this fact in more details, and we derive the necessary and sufficient conditions that the concretization function of a Galois connection must satisfy in order for the upper and lower bound to form a conservative approximation.

1.8.6 Interface Theories

The agent algebra that we propose is similar in nature to the *block algebra* proposed by De Alfaro and Henzinger [30]. In this section we discuss the differences and similarities with our work.

Block algebras are mathematical structures that are used to model a system as a hierarchical interconnection of blocks. The algebra consists of a set of blocks and interconnections; a *composition operator* that returns the parallel composition of two blocks; a *connection operator* that composes blocks and interconnections to establish a relation between the ports of the blocks; and a binary relation, called *hierarchy*, that holds when a block F refines a more abstract block F'. The operators and the hierarchy relation must also satisfy certain properties that are consistent with their intuitive meaning.

Block algebras resemble agent algebras in several respects. Both are concerned with building a model hierarchically through the application of composition starting from some basic blocks. However, the operators that are used to construct the hierarchy work in different ways. In particular, connection are formed in our framework by identifying signals with the same name, rather than through an explicit object (the interconnection) as in block algebras. This choice reflects our focus on the semantic domain, rather than in the language used to specify agents: if an interconnection operator is needed (because, for example, it is convenient to specify a system), it can easily be obtained by syntactic transformations, or, more generically, by using a separate agent as described in section 1.4. Moreover, block algebras lack a notion of scoping, which is in our opinion essential in hierarchical designs to hide the internal details of a composite. Observe also that in agent algebras *hierarchy* and *refinement* are two distinct concepts. Refinement is a relation between agents that establishes when one agent can be substituted for another. We instead reserve the term *hierarchy* to denote the containment relation between a composite and its parts. In the remainder of this section we further discuss the differences between the notion of refinement in block algebras and the notion of refinement in agent algebras.

As in agent algebras, blocks can be very general objects. In block algebras, however, blocks are further classified into two classes: *components* and *interfaces*. Informally components are descriptions of agents that say what an agent does. Conversely, interfaces are descriptions of agents that say the expectations that the agent has about its environment. This distinction is based upon the observation that physical components do something in any possible environment, whether they behave well or misbehave. In contrast, interfaces describe for each block the environments that can correctly work with the block. On the other side, both components (through their behaviors) and interfaces provide guarantees to the environment regarding their possible actions.

Assumptions and guarantees of components and interfaces are related to the refinement relation in the algebra. If a block F refines a block F', then F may only make weaker assumptions on its environment than F'. At the same time F should make stronger guarantees than F'. This is consistent with the view of refinement as substitutability. If a more refined block is to replace a more abstract one in any environment, then the more refined block shall not assume properties about the environment that are not assumed by the abstract block. Because components make no assumptions about their environment, abstraction simply weakens the guarantees, and refinement strengthen them. Conversely, interfaces also exhibit a contravariant strengthening and weakening of the environment assumptions.

The distinction between components and interfaces has, according to the authors, an impact on the notion of compositionality (i.e., the relationships between the composition operators and the refinement relation). Component algebras are block algebras that support *compositional abstraction*. This implies that if f and g are two compatible components (i.e., their composition is defined), and if f refines f', then also f' and g are compatible. This is possible because "for components, abstraction is weakening and compatibility is made more likely by weakening". Interface algebras, on the other hand, are block algebras that support *compositional refinement*. In this case, if interfaces f' and g are compatible, and if f refines f', then also f and g are compatible. This is because "for interfaces, refinement weakens input assumptions and thus can make compatibility more likely".

Agent algebras do not distinguish between components and interfaces. Likewise, the "direction" of compositionality in agent algebras is independent of the nature of the agents, and always agrees with the notion of compositional refinement. There are several reasons for this.

In the first place, as already discussed, the notion of refinement in agent algebras always represents substitutability. Later, we argue that compositional refinement is the only notion that is compatible with this interpretation. Second observe that, according to their informal definition, components are just a special case of interfaces. They are interfaces that make no assumptions about their environment. Hence, any general result that holds for interfaces must hold for components, as well. In particular, given an interface algebra we can find a subset of the interfaces that are components (the interfaces that make no assumptions). If this set if closed under composition and connection, it forms a subalgebra. However, the component subalgebra of an interface algebra supports compositional refinement, and not compositional abstraction. This makes components a subclass of interfaces, rather than a distinct class.

The argument that for components abstraction makes compatibility more likely is difficult to explain. First, abstraction and refinement weaken and strengthen assumptions and guarantees in the same way for both components and interfaces, and should therefore lead to the same compositionality result. In the case of components, however, assumptions are held constant (and empty) and abstraction simply weakens the guarantees. However, it seems more intuitive that weaker guarantees should make compatibility *less* likely, because it is harder to prove the assumptions of the other components. In addition, because components make no assumptions, the guarantees can play no role in deciding compatibility, since any guarantee will satisfy the empty assumption. Weaker assumptions do make compatibility more likely, but in the case of components these are always empty. Other conditions must be verified in order for compositional abstraction to work.

In block algebras, components can be related to interfaces by a relation called *implementation*. This relation can be seen as a form of refinement. It is, however, a refinement that occurs *across* two different algebras. It is interesting to note that compositionality for the implementation relation follows the rule of compositional refinement. The implementation relation roughly corresponds to our conservative approximation, or to a Galois connection. A relation, instead of a function or pair of functions, may provide more flexibility. For example, a relation may express the fact that a certain component implements interfaces that are unrelated. With conservative approximation this would translate in different upper and lower bounds. A relation could therefore provide more information. We prefer to use functions because they provide additional results with the problem of refinement verification.

The distinction between interfaces and components seems to ultimately arise from the fact that components, by making no assumptions, are unable to constrain their environment. For this reason, components are often called *input-enabled*, or *receptive*. Receptiveness and environment constraints are not, however, mutually exclusive. The two notions coexist, and are particularly well-behaved, for example in the trace-based agent algebra model presented in section 5.2, derived from the two-set trace structures proposed by Dill [34] (or, in Negulescu's Process Spaces [71, 72]). In this model traces are classified as either *successes* or *failures*. In order for a system to be *failure-free*, the environment of each agent must not exercise the failure traces. Failure traces therefore represent the assumptions that an agent makes relative to its environment. However, the combination of failure and success traces makes the agents receptive. Note in addition that the operations of this algebra are monotonic and that they support compositional refinement (see the discussion in section 2.4). Hence, two-set trace structures are “components” (because they are receptive), but follow the compositionality principle of “interfaces”.

We believe that all of the interface models proposed in the framework of interface theories [29, 18, 19, 31] can be explained in these terms. For example, interface automata [29] can be explained almost exactly in terms of the prefix closed trace structures of Dill [34]. In particular, interface automata are always kept in canonical form, and their composition operator is an implementation of Dill’s autofailure manifestation and failure exclusion.

The distinction between interfaces and components seems therefore unnecessary. Or, to be more precise, the distinction between a component and its interface in our framework has to do with a difference in the level of abstraction, rather than with a difference in their nature.

1.8.7 Process Spaces

Process Spaces [71, 72] is a very general class of concurrency models, and it compares quite closely to our trace-based agent models, described in chapter 4. Given a set of executions $\mathcal{E}$, a Process Space $\mathcal{S}_{\mathcal{E}}$ consists of the set of all the processes (X, Y), where X and Y are subsets of $\mathcal{E}$ such that $X \cup Y = \mathcal{E}$. The sets of executions X and Y of a process are not necessarily disjoint, and they represent the assumptions (Y) and the guarantees (X) of the process with respect to its environment. This interpretation of the execution sets is enforced by the refinement relation, which is defined as $q \sqsubseteq p$ if and only if $X_q \subseteq X_p$ and $Y_q \supseteq Y_p$ (note that in [71, 72] the notation for the order is reversed, so that the above case is written $p \sqsubseteq q$).

In trace-based agent algebras executions are called traces, and processes are called trace structures. In particular, our two-set trace structures express conditions that are similar to the assumptions and guarantees of processes by distinguishing between *success* and *failure* traces. The similarities between the two models are not coincidental, since both frameworks extend the trace theory presented in [34]. The generalization to abstract executions that Process Spaces claim to be their distinguishing aspect is in fact already present in trace structure algebra [12], the framework that we take as our basis for generalization. Nevertheless, several differences distinguish trace structures and processes. For example, in our generalization we do not require the elimination of connectivity restrictions or of references to inputs and outputs, and instead deal with partial operators directly. Indeed, the notion of monotonicity for partial functions that we introduce in section 2.4 is used as the basis of our models throughout this work, and sheds light on the requirements that must be met by a model to exhibit the necessary properties to apply compositional methods.

Process Spaces introduce a notion of conformance (called “testing”) and mirror (called “reflection”) which closely reflect the corresponding notions of [34] and [96]. Similarly to [34], these notions are fixed and are derived from a concept of robustness of processes that amounts to not making any assumptions (i.e., for process (X, Y), $Y = \mathcal{E}$). Because of their construction, Process Spaces always admit conformance orders and mirrors regardless of the set of executions. In this work we introduce these notions in the framework of agent algebra. Agent algebra is a further generalization of our trace-based models which are here presented as a special case. In particular, agent algebras give up entirely the notion of an execution, and talk about agents in abstract terms. Because of that, we use a parameterized notion of conformance that can be customized to the particular instance of agent algebra. Depending on the robustness criterion, which is embodied by a set of agents G (called a conformance set), mirrors may or may not exist for a particular model. With agent algebra we derive an exact characterization of the existence of a mirror function in terms of the conformance order and certain greatest elements in the sets of agents. We also derive a general solution to the local specification synthesis problem, which is also solved, however limited to the parallel composition operator, in process spaces (called the “design inequality”).

Process Spaces do not have such operators as projection and renaming (although they are considered as process abstractions), which we instead take as fundamental operators of our algebra. In addition Process Spaces enjoy a lattice structure which, because of the additional generality, is not necessarily present in agent algebra. Therefore, Process Spaces are endowed with additional meet and join operators. We introduce these operators only where they are necessary, i.e., when characterizing mirror functions, and do not develop a general theory. Trace-based agent algebras could be augmented with these operators. This extension is left as future work.

The notion of process abstraction in Process Spaces is related to the notion of conservative approximation. Process abstractions are again defined as axialities of a relation and its inverse on abstract executions, as discussed in subsection 1.8.5. In Process Spaces, a process abstraction is classified as optimistic or pessimistic according to whether it preserves a robustness verification (in our terminology, whether an agent is in G or not) from the concrete to the abstract or from the abstract to the concrete model. These two kinds of abstractions can be used in combination to preserve the verification result both ways. In that case, the two models are essentially isomorphic since there is effectively no loss information.

Optimistic and pessimistic process abstractions roughly correspond to the upper and lower bound of conservative approximations. However, our use of the two bounds is very different. In particular we only deal with refinement verification, since robustness verification would not make sense with our parameterized notion. More importantly, our use of the upper and lower bound is significantly different, since we apply them in combination (the lower bound for the specification, the upper bound for the implementation) without sacrificing the abstraction.

Like our examples of trace-based agent algebras, Process Spaces can be seen as a special case of agent algebra as well. In fact, all of our trace-based models could be constructed in general by providing for each agent a classification of the set of executions (or traces) into a set of distinguished classes $\mathcal{I}$. For example, $\mathcal{I}$ could distinguish between the successes and the failures of our two-set trace structures. Operations on agents could then be derived from corresponding operations on the elements of $\mathcal{I}$. Both our trace structures models and Process Spaces could be seen as special cases of this more general technique. We defer this generalization for our future work.

1.8.8 Category Theoretic Approaches

Category theory [66, 9, 75] is a particular form of algebraic approach in which elements are partitioned into *objects* and *morphisms*, i.e., into the objects under study and their relationships. Similarly to our agent algebras, it is possible to construct different categories that correspond to different models of computation.

The work that is more closely related to ours is due to Winskel et al. [83, 95, 84]. In their formalism, each model of computation is turned into a category where the objects are the agents, and the morphisms represent a refinement relationship based on *simulations* between the agents. The authors study a variety of different models that are obtained by selecting arbitrary combinations of three parameters: behavior vs. system (e.g., traces vs. state machines), interleaving vs. non-interleaving (e.g., state machines vs. event structures) and linear vs. branching time. The common operations in a model are derived as universal constructions in the category. Given a set of objects and morphisms in a category, a universal construction consists of a new set of objects and morphisms that enjoy a particular universal property, that is a property that is true *for all* the elements (usually morphisms) of a certain set, relative to the original objects and morphisms. In particular, the authors describe how to derive projections and renamings by lifting corresponding morphisms on the alphabets to the category of the model, and how to use products and coproducts (sums) to model parallel composition and choice. These constructions are the same across the different models. However, for each model they give rise to different operations in accordance to the difference between the structure of the objects and the morphisms that relate them. The constructions are shown to “implement” the usual notions of operations on agents that are typical of the models being considered.

A fundamental aspect addressed in this work, and which is also fundamental in our work, is the development of relationships between models. This is accomplished by relating the categories corresponding to different models of computation by means of functors, which are homomorphisms of categories that preserve morphisms and their compositions. When categories represent models of computation, functors establish connections between the models in a way similar to abstraction maps and semantic functions. In particular, when the morphisms in the category are interpreted as refinement, functors become essentially monotonic functions between the models, since preserving morphisms is equivalent to preserving the refinement relationship.

In [84], the authors thoroughly study the relationships between the eight different models of concurrency above by relating the corresponding categories through functors. In addition, these functors are shown to be components of *reflections* or *co-reflections*. These are particular kinds of adjoints, which are pairs of functors that go in opposite directions and enjoy properties that are similar to the order preservation of the abstraction and concretization maps of a Galois connection. When the morphisms are interpreted as refinement, reflections and co-reflections generalize the concept of Galois connection to preorders. In addition, reflections and co-reflections enjoy additional preservation properties relative to the universal constructions in the category, and by transitivity to the operators of the model. These properties are similar to those that are required of the abstraction function of an abstract interpretation, or those sufficient to make a conservative approximation compositional.

There are certainly many similarities between our work and the category theoretic approach. In particular, the idea of generalizing the construction of the operators by extending corresponding, but simpler operations on a different domain is at the basis of the relationship between trace algebras and trace structure algebras [12], which we here revisit. The operators of the trace structure algebra are in essence the “lifting” of the corresponding operators on the trace algebra, by either a simple extension to sets or by a more complex definition as for parallel composition (which involves, in fact, a sort of restricted product). It wouldn’t be surprising if this technique could be explained exactly in the terms presented in [95]. In our work, however, we are concerned with a more restricted set of operators (for example, we do not include choice in our basic set).

Our approach with agent algebra differs, since the agents, the operators on the agents, and the order relationship can all be defined independently, as long as they satisfy the required properties, including $\top$-monotonicity. In other words, we take an axiomatic approach, instead of a constructive approach. This is simply a different point of view, one that allows us to use the axioms as basic building blocks in our proofs.

The similarities also extend to how we establish relationships between different models, or algebras. In fact, the relationships between categories based on adjoints are similar in nature to the abstractions and refinements obtained by abstract interpretations.[5] However, as described above for abstract interpretations, we use independent upper and lower bounds for the implementation and the specification in order to derive a stronger result in terms of preservation of the refinement relation, and avoidance of false positive verification results. Indeed, we require two Galois connections, instead of one, to determine a single conservative approximation. In the work presented in [84], this translates in two adjoints per pair of categories.

We also believe that the aim of our work is different. In [84], the authors are mostly concerned with the classification of different models of concurrency, and they certainly go to great length to establish certain particular relationships between certain particular models (and, in doing so, they also propose a new, richer, model of concurrency). Our focus goes beyond the classification, as we strive to find techniques that can be applied to most models that fit in our framework. The refinement verification technique based on conservative approximations, and the local specification synthesis based on mirrors are two examples of tools that apply to a variety of models. While these questions are not raised by Winskel et al., it is true that we could have addressed them in a categorical setting. The choice of the "language" of presentation is one that is often made upfront, and is often resolved in favor of the ones that the authors are most comfortable with. In this case, one should ask whether category theory could shed more light, for example by exposing more properties, or save some of the work, by means of ready to use results. We believe that this is not the case for agent algebras and that a simpler language, based on sets and relations, is sufficient for our aims. For example, the specialization of the results on adjoints in category theory to Galois connections is all we need to talk about conservative approximations. We also believe that the specialization of the language is ultimately more intuitive. It is true that the categorical approach would, for example, provide the extension to preorders (that we do not present here) essentially for free. However, such an extension is straightforward for anyone familiar with preorders, by taking the appropriate notions up to equivalence. Aspects of category theory may however prove useful for other extensions. As a form of generalized homomorphism, functors could for example be used to provide a more powerful notion of abstraction on agents executions, as discussed in section 6.2.

[5]It would certainly be interesting to see the authors' own account of the similarities and differences between using adjoints and an approach based on abstract interpretations.

1.8.9 Rosetta

The ability to define domains of agents for different models of computation is also a central concept of the Rosetta language [56, 57, 58, 59]. In Rosetta, a model of computation is represented as a domain that is described declaratively as a set of assertions in a higher order logic. The definition first declares the objects of the discourse, such as for example the variable that represents time, or power, or a transition relation, thus representing the presence of state. The assertions then axiomatically determine the interpretation of these quantities in the properties that they must satisfy.

Different domains can be obtained by extending a definition in a way similar to the subtyping relation of a type system. The extended domain inherits all the assertions (the terms) of original domain, and adds additional constraints on top of them. Domains that are obtained this way are automatically related by an abstraction/refinement relationship. Domains that are unrelated can still be compared by constructing functions, called interactions, that (sometimes partially) express the consequences of the properties and quantities of one domain onto another. This process is particularly useful for expressing and keeping track of constraints during the refinement of the design.

In contrast to Rosetta we are not concerned with the definition of a language. In fact, we define a domain directly as a collection of elements of a set, not as the model of a theory. In this sense, the approach taken by Rosetta seems more general. As already discussed, however, the restrictions that we impose on our models allow us to prove additional results that help us create and compare the models. In particular, the interactions between different domains in Rosetta are essentially unconstrained. In our case we are interested in proving facts about these interactions, and therefore require that our abstraction maps satisfy certain properties that have mostly to do with preservation of refinement verification. In particular, while the interactions in Rosetta are exact, we instead employ upper and lower approximations.

1.8.10 Hybrid Systems

In our framework we define a domain of agents that is suitable for describing the behavior of systems that have both continuous and discrete components. The term hybrid is often used to denote these systems. Many are the models that have been proposed to represent the behavior of hybrid systems. Most of them share the same view of the behavior as composed of a sequence of steps; each step is either a continuous evolution (a flow) or a discrete change (a jump). Different models vary in the way they represent the sequence. One example is the Masaccio model proposed by Henzinger et al. [47, 49].

In Masaccio the representation is based on components that communicate with other components through variables and locations. Variables are used to exchange data, while locations are used to transfer the flow of control. During an execution the flow of control transitions from one location to another according to a state diagram that is obtained by composing the components that constitute the system. Each transition in the state diagram models a jump or a flow of the system and constrains the input and output variables through a difference or differential equation. The underlying semantic model is based on sequences. The behavior of each component is characterized by a set of finite executions, each of them composed of an entry location and a sequence of steps that can be either jumps or flows. An exit location is optional. The equations associated with the transitions in the state diagram define the legal jumps and flows that can be taken during the sequence of steps.

The operation of composition in Masaccio comes in two flavors: parallel and serial. The parallel composition is defined on the semantic domain as the conjunction of the behaviors: each execution of the composition must also match an execution of the individual components. Roughly speaking, a sequence of steps s is an execution of the parallel composition $A||B$ iff the projection of s (obtained by restricting s to the variables of A or B) is an execution of A *and* B. This operation models the concurrent activities of two components. Conversely, serial composition is defined as the disjunction of the behaviors: each execution of the composition need only match the execution of one of the components. A sequence of steps s is an execution of the serial composition $A + B$ iff the projection of s (obtained by restricting s to the variables of A or B) is an execution of A *or* B. Despite its name, this operation doesn't serialize the behaviors of the two components; it is more like a choice operator. A further operation of *location hiding* is required to serialize executions.

To date we are not aware of a formal definition of parallel or serial composition for the state transition representation.

In our work we take an approach that is based solely on the semantic domain. Note in fact that the semantic model based on sequences and the representation based on a state transition system are easily decoupled. In our framework we talk about hybrid models in terms of the semantic domain only (which is based on functions of a real variable rather than sequences). This is a choice of emphasis: in Masaccio the semantic domain is used to describe the behavior of a system which is otherwise represented by a transition system. In our approach the semantic domain is the sole player and we emphasize results that abstract from the particular representation that is used. It's clear, on the other hand, that a concrete representation (like a state transition system) is extremely important in developing applications and tools that can generate or analyze an implementation of a system.

In this work we present several models for semantic domains. Masaccio compares to our more detailed model. In our approach we have decided to model the flows and the jumps using a single function of a real variable: flows are the continuous segments of the functions, while jumps are the points of discontinuity. This combined view of jumps and flows is possible in our framework because we are not constrained by a representation based on differential equations, and hence we do not require the function to be differentiable. Another difference is that different components are allowed to concurrently execute a jump and a flow, as long as the conditions imposed by the operation of parallel composition are satisfied.

Because in Masaccio the operations of composition are defined on the semantic domain and not on the representation it is easy to do a comparison with our framework. Parallel composition is virtually identical (both approaches use a projection operation). On the other hand we define serial composition in quite different terms: we introduce a notion of concatenation that is difficult to map to the sequence of steps that include serial composition and location hiding, which is contrary to our principle of not mixing different operators. This could simply be an artifact of the representation based on state transitions that requires the identification of the common points where the control can be transferred.

The concept of *refinement* in Masaccio is also based on the semantic domain. Masaccio extends the traditional concept of trace containment to a prefix relation on trace sets. In particular, a component A refines a component B either if the behavior of A (its set of executions) is contained in the behavior of B, or if the behaviors of A are suffixes of behaviors of B. In other words, B could be seen as the prefix of all legal behaviors.

In our framework we must distinguish between two notions of refinement. The first is a notion of refinement within a semantic domain: in our framework this notion is based on pure trace containment. We believe this notion of refinement is sufficient to model the case of sequential systems as well: it is enough to require that the specification include all possible continuations of a common prefix. The second notion of refinement that is present in our framework has to do with changes in the semantic domain. This notion is embodied in the concept of conservative approximation that relates models at one level of abstraction to models at a different level of abstraction. There is no counterpart of this notion in the Masaccio model.

1.8.11 Local Specification Synthesis

We have already described the problem of local specification synthesis in section 1.3. The literature on techniques to solve it is vast. Here we focus on two of the proposed techniques and highlight in particular the differences in the scope and aim relative to our work.

Larsen et al. solve the problem of synthesizing the local specification for a system of equations in a process algebra [60]. In order to represent the flexibility in the implementation, the authors introduce the Disjunctive Modal Transition System (DMTS). Unlike traditional labeled transition systems, the DMTS model includes two kinds of transitions: transitions that *may* exist and transitions that *must* exist. The transitions that must exist are grouped into sets, of which only one is required in the implementation. In other words, the DMTS is a transition system that admits several possible implementation in terms of traditional transition systems.

The system is solved constructively. Given a context and a specification, the authors construct a DMTS whose implementations include all and only the solution to the equation. To do so, the context is first translated from its original equational form into an operational form where a transition includes both the consumption of an event from the unknown component, and the production of an event. The transitions of the context and of the specification are then considered in pairs to deduce whether the implementation may or may not take certain actions. A transition is possible, but not required, in the solution whenever the context does not activate such transition. In that case, the behavior of the solution may be arbitrary afterwards. A transition is required whenever the context activates the transition, and the transition is used to match a corresponding transition in the specification. A transition is not allowed in the solution (thus it is neither possible, nor required) whenever the context activates it, and the transition is contrary to the specification.

The solution proposed by Larsen et al. has the advantage that it provides a direct way of computing the set of possible implementations. On the other hand it is specific to one model of computation (transition systems). In particular, the solution does not provide any insight as to why the technique works (despite the proof that it does work!). Conversely, our approach consists of working at a sufficiently high level of abstraction (above the model of computation) so that the conditions of applicability are exposed. Our solution is however not constructive, and is expressed in an algebraic form that may or may not be computable. It does however ensure that if the result is computable, then the solution is correct.

Yevtushenko et al. [98] present a formulation of the problem that is more closely related to ours. The local specification is obtained by solving abstract equations over languages under various kinds of composition operators. By working directly with languages, the solution can then be specialized to different kinds of representations, including automata and finite state machines.

In the formalism introduced by Yevtushenko et al., a language is a set of finite strings over a fixed alphabet. The particular notion of refinement proposed in this work corresponds to language containment: language P refines a language Q if and only if $P \subseteq Q$. If we denote with $\overline{P}$ the operation of complementation of the language P (i.e., $\overline{P}$ is the language that includes all the finite strings over the alphabet that are not in P), then the most general solution to the equation in the variable X

$$A \cdot X \subseteq C$$

is given by the formula

$$S = \overline{A \cdot \overline{C}}.$$

The language S is called the most general solution because a language P is a solution of the equation if and only if $P \subseteq S$. In the formulas above, the operator $\cdot$ can be replaced by different flavors of parallel composition, including synchronous and asynchronous composition. These operators are both constructed as a series of an expansion of the alphabet of the languages, followed by a restriction. For the synchronous composition, the expansion and the restriction do not alter the length of the strings of the languages to which they are applied. Conversely, expansion in the asynchronous composition inserts arbitrary substrings of additional symbols thus increasing the length of the sequence, while the restriction discards the unwanted symbols while shrinking the string.

The language equations are then specialized to various classes of automata, including finite automata and finite state machines. This provides an algorithmic way of solving the equation for restricted classes of languages (i.e., those that can be represented by the automaton). The problem in this case consists of proving certain closure properties that ensure that the solution can be expressed in the same finite representation as the elements of the equation. In particular, the authors consider the problem of receptiveness (there called I-progression) and prefix closure.

The solution obtained in our work is similar to that proposed by Yevtushenko. Our approach is however more general. In particular, we work with abstract behaviors (in fact, our most general formulation does not require behaviors at all) and do not require any particular form of the composition operator, as long as it satisfies certain assumptions. In other words, we take an axiomatic approach instead of a constructive one, as already noted in section 1.4. Therefore we need to prove our result only once, instead of once for every model that fits in our framework. In particular, our result applies whether we consider the synchronous or the asynchronous parallel composition operator.

In our work, we however only marginally consider the problem of finite representation, and therefore of the algorithmic solution of the equation. Indeed, questions of closure must be solved upfront in order for a model to fit in our framework. In the case of the asynchronous and of the synchronous operators above, these questions are also addressed by Dill [34], Wolf [96]. The problem of the synthesis of the local specification for the combinational case is also addressed by Burch et al. [13]. Since we generalize their work, we expect to be able to take full advantage of their characterizations.

1.9 Outline of the Dissertation

This dissertation is divided in four parts. The first two parts develop the basic theory underlying our algebraic framework, with simple examples that complement the theoretical presentation. The third part introduces trace-based agent algebras, and is mainly devoted to examples of different models of computation and their relationship. Finally, the fourth part further develops the theory of trace-based agent algebras, and presents an example of application of local specification synthesis to the problem of protocol conversion.

Chapter 2

Chapter 2 presents the basic framework of Agent Algebra with the definition of an agent algebra and of an ordered agent algebra. One of the main contributions of this chapter is a parameterized notion of monotonicity that applies to partial functions (definition 2.20). The codomain of the partial function is extended with an additional element d, and a total extension of the partial function is defined that maps to d the elements over which the original function was undefined. The position of the element d in the order of the codomain is particularly important. Different d's correspond to different notions of monotonicity, and give rise to different compositionality principles. We show that $\top$-monotonicity, obtained by placing the d element at the top of the codomain, is the only notion of monotonicity that is consistent with the interpretation of the order relationship as substitutability (theorem 2.24).

In this chapter we also introduce the notion of a conservative approximation, that we take from [12], to relate one domain of agents to another, more abstract, domain. A conservative approximation has two functions. The first, called the lower bound, is used to abstract agents that represent the specification of a design. The second, called the upper bound, is used to abstract agents that represent possible implementations of the specification. A conservative approximation is defined so that if the implementation satisfies the specification in the abstract domain, then the implementation satisfies the specification in the more detailed domain, as well. Here we further develop the theory related to the inverse of a conservative approximation, and find sufficient conditions to ensure that it is an embedding (theorem 2.72).

Another contribution of this chapter is a detailed study of the relationship between conservative approximations and the well established notions of Galois connections and abstract interpretations. In particular we give the necessary and sufficient conditions for a pair of Galois connections to form a conservative approximation (corollary 2.101). We also show that conservative approximations are in general more powerful than abstract interpretations. This study allows us to relate two different models through a common refinement (or a common abstraction), and leads to the definition of the notion of co-composition between agents that belong to different models. Our contributions also include a formalization of the process of platform-based design that makes explicit the relationships between function and architecture by way of a common semantic platform (subsection 2.8.5).

Chapter 3

Chapter 3 further develops the theory of agent algebras. Here we define what it means for an agent to *conform* to another agent in terms of all possible contexts. Given a set of agents G, an agent p conforms to an agent p' if substituting p for p' in any context keeps the evaluation of the context within G. The set G can therefore be seen as an initial partition for the conformance order that is then refined by the evaluation of the contexts. We then develop techniques to reduce the number of contexts that must be considered to check conformance, up to, in certain cases, a single composition context that is called mirror. These are generalizations of the corresponding definitions and results originally due to Dill [34]. In particular, the set G generalizes the notion of *failure freedom*. However our contribution here is not limited to just the generalization of these notions to abstract agents. By working above the level of a specific model, we are in fact able to determine the precise conditions that must be met for the existence of mirrors. In particular, we show that if the operators of the algebra are $\top$-monotonic, and if G is downward closed relative to the agent ordering, then the relative notion of conformance (with fewer contexts) implies the full notion of conformance (with all the contexts) (theorem 3.42). In addition, we introduce a notion of compatibility and provide a complete and general characterization of the existence of mirror functions and their construction in terms of conformance and compatibility (theorem 3.80). This is a particularly strong result. Specifically, it helps us understand how a model should be extended in case it is not already endowed with a mirror function. To that end, we also consider extensions of a mirror function that include a predicate or that applies to partitions that form subalgebras.

In this chapter we also present a general formulation of the local specification synthesis problem, that entails deriving a local specification for a component, given a global specification and the context of the component. This solution requires that expressions be transformed into a particular form that we call RCP normal form. We define the notion of expression equivalence and show that if an algebra satisfies certain sufficient conditions, then every expression is equivalent to an expression in RCP normal form (theorem 3.16). The concept of mirror, and the ability to transform expressions in a suitable normal form are used to solve the problem in closed form, one of the main contributions of this work (theorem 3.119).

Chapter 4

Chapter 4 introduces trace-based agent algebras, which are similar to the trace algebras and trace structure algebras introduced by Burch [12]. The signature, which was fixed for trace structure algebras, is here generalized using the more general notion of agent algebra and the construction of direct products. Here we also develop more complex models of concurrent systems, with particular attention to the models of computation used in todays heterogeneous embedded systems, both control and data-dominated (section 4.3). In particular we develop models that are useful for studying the behavior of hybrid systems. In addition we study relationships between these models in terms of conservative approximations. Our main original contribution here, besides the examples, is the derivation of conservative approximations between trace-based agent algebras in terms of the axialities of relationships between the traces (corollary 4.20). This is a generalization of the work of Burch [12], who proposes one specific form of conservative approximation induced by homomorphisms.

Chapter 5

In chapter 5 we prove the existence of a mirror function for trace-based agent algebras under a specific choice for the conformance ordering. We also extend the notion of a single-set trace structure to that of a two-set trace structure, again generalizing the work of Dill [34], in order to model successes and failures, and we derive a mirror function. We compare our results to those of Dill, and explain Dill's results and his canonical form in terms of a subalgebra of our more general model. Besides the generalization of successes and failures to abstract behaviors, our contribution includes necessary and sufficient conditions for the existence of canonical forms in a subalgebra (theorem 5.28). We conclude the chapter by showing an example of application of the local specification synthesis technique using a trace-based synchronous model to solve a protocol conversion problem.

Chapter 2

Agent Algebras

This chapter describes some very general methods for constructing different models of concurrent systems, and for proving relationships between these models. We introduce the notion of an *agent algebra* to formalize the model of a concurrent system. Agent algebras are a broad class of models of computation. They developed out of Burch's and our work on *concurrency algebra*, *trace algebra* and *trace structure algebra* [12, 14, 15], which builds on Dill's work on *circuit algebra* and *trace theory* [33]. Through trace structure algebra we have studied concepts, such as *conservative approximations*, that help clarify the relationships between different models of computation. Agent algebra provides a simpler formalism for describing and studying these concepts. The trade-off is that agent algebra is more abstract and provides less support for constructing models of computation.

An agent algebra is a simple abstract algebra with three operations: parallel composition, projection, and renaming. The three operations must satisfy certain axioms that formalize their intuitive interpretation. The domain (or carrier) of an agent algebra is intended to represent a set of processes, or *agents*. Any set can be the domain of an agent algebra if interpretations for parallel composition, projection and renaming that satisfy the axioms can be defined over the set. In this document, whenever we define an interpretation for these three operations, we always show that the interpretation forms an agent algebra, which gives evidence that the interpretation makes intuitive sense.

Agent algebras can be constructed from other agent algebras by the usual devices of direct product and sum. We introduce these construction in this chapter, and show that they indeed yield new agent algebras. Direct products will also be useful in the following chapters to construct hybrid models and to provide a sort of "signature specification" to a set of agents.

In verification and design-by-refinement methodologies a specification is a model of the design that embodies all the possible implementation options. Each implementation of a specification is said to *refine* the specification. In our framework, agent algebras may include a preorder on the agents that represents this refinement relationship. Proving that an implementation refines a specification is often a difficult task. Most techniques decompose the problem into smaller ones that are simpler to handle and that produce the desired result when combined. To make this approach feasible, the operations on the agents must be monotonic with respect to the refinement order. In this chapter we extend the notion of monotonic function to the case of partial functions, and show under what circumstances compositional verification techniques can be applied.

An even more convenient approach to the above verification consists of translating the problem into a different, more abstract semantic domain, where checking for refinement of a specification is presumably more efficient. A *conservative approximation* is a mapping of agents from one agent algebra to another, more abstract, algebra that serves that purpose. Thus, conservative approximations are a bridge between different models of computation.

Informally, a model is a conservative approximation of a second model when the following condition is satisfied: if an implementation satisfies a specification in the first model, then the implementation also satisfies the specification in the second model. Conservative approximations are useful when the second model is accurate but difficult to use in proofs or with automatic verification tools, and the first model is an abstraction that simplifies verification.

Conservative approximations represent the process of abstracting a specification to a less detailed semantic domain. Inverses of conservative approximations represent the opposite process of refinement. In this chapter we introduce the notion of the inverse of a conservative approximation, and relate our technique to the abstraction and concretization functions of Galois connections and abstract interpretations.

2.1 Preliminaries

The algebras we develop in this document have many characteristics in common. This section discusses several of those characteristics.

Each of the algebras has a domain D which contains all of the objects under study for the algebra. We borrow the term "domain" from the programming language semantics literature; algebraists call D a "carrier".

Associated with each element of D is a set A of *signals*, called an *alphabet*. Signals are used to model communication between elements of D. Typically signals serve as actions and/or state variables that are shared between elements of D, but this need not be the case. Associated with each algebra is a *master alphabet*. The alphabet of each agent must be a subset of the master alphabet. A master alphabet typically plays the role of $\mathcal{A}$ in the following definition.

Definition 2.1 (Alphabet). If $\mathcal{A}$ is a set, then A is an *alphabet over* $\mathcal{A}$ if and only if $A \subseteq \mathcal{A}$.

We often make use of functions that are *over* some domain or master alphabet.

Definition 2.2. Let S be an arbitrary set. A function f of arity n is *over* S if and only if $\mathit{dom}(f) \subseteq S^n$ and $\mathit{codom}(f) \subseteq S$.

Agent algebras use three operators over the domain of agents. They are defined as follows.

Definition 2.3 (Renaming). Let $\mathcal{A}$ and D be sets. A *renaming operator over master alphabet* $\mathcal{A}$ *and over domain* D (written *rename*) is a total function such that

1. the domain of *rename* is the set of bijections over $\mathcal{A}$, and
2. the codomain of *rename* is the set of partial functions from D to D.

Definition 2.4 (Projection). Let $\mathcal{A}$ and D be sets. A *projection operator over master alphabet* $\mathcal{A}$ *and over domain* D (written *proj*) is a total function such that

1. the domain of *proj* is the set of alphabets over $\mathcal{A}$, and
2. the codomain of *proj* is the set of partial functions from D to D.

Definition 2.5 (Parallel Composition). Let D be a set. A *parallel composition operator over domain* D (written a binary infix operator $\|$) is a partial function over D such that

1. the domain of $\|$ is $D \times D$, and
2. the codomain of $\|$ is D.

Intuitively, the renaming operator takes a renaming function over the master alphabet, and applies it to an element of the domain to obtain the corresponding renamed element according to the renaming function. Here the renaming function defines the desired correspondence between the signals in the alphabet (e.g., signal a is mapped to signal α, and b to β). The renaming operator, on the other hand, defines how the renaming function should be applied to agents. Similarly, the projection operator takes the set of signals that must be *retained* as a parameter. The operator must then define how the remaining signals should be removed from the agent to which it is applied. Parallel composition, on the other hand, does not take any alphabet as a parameter, and is simply a binary function over the domain of the algebra. Examples of the definition and use of these operators are found throughout this work.

The codomain of the operators above are partial functions, and can therefore be undefined for certain arguments. In the rest of this work, we often say that the operator itself is undefined, with the understanding that it is the resulting partial function that really is undefined at a certain argument. In formulas, we use the notation $\downarrow$ to indicate that a function is defined at a particular argument, and $\uparrow$ to indicate that it is undefined.

2.2 Agent Algebras

Informally, an agent algebra $\mathcal{Q}$ is composed of a domain D which contains the agents under study for the algebra, and of the following operations on agents: parallel composition, projection and renaming. The algebra also includes a master alphabet $\mathcal{A}$, and each agent is characterized by an alphabet A over $\mathcal{A}$. All of this is formalized in the following definitions. Throughout the document, equations are interpreted to imply that the left hand side of the equation is defined if and only if the right hand side is defined, unless stated otherwise.

Definition 2.6 (Agent Algebra). An *agent algebra* $\mathcal{Q}$ has a domain $\mathcal{Q}.D$ of *agents*, a *master alphabet* $\mathcal{Q}.\mathcal{A}$, and three operators: *renaming* (definition 2.3), *projection* (definition 2.4) *parallel composition* (definition 2.5), denoted by rename, proj and $\|$. The function $\mathcal{Q}.\alpha$ associates with each element of D an alphabet A over $\mathcal{A}$. For any p in $\mathcal{Q}.D$, we say that $\mathcal{Q}.\alpha(p)$ is the *alphabet of* p.

The operators of projection, rename and parallel composition must satisfy the axioms given below, where p and p' are elements of D, $A = \alpha(p)$, $A' = \alpha(p')$, B is an alphabet and r is a renaming function.

A1. If $\mathit{proj}(B)(p)$ is defined, then its alphabet is $B \cap A$.

A2. $\mathit{proj}(A)(p) = p$.

A3. If $\mathit{rename}(r)(p)$ is defined, then $A \subseteq \mathit{dom}(r)$ and $\alpha(\mathit{rename}(r)(p)) = r(A)$, where r is naturally extended to sets.

A4. $\mathit{rename}(\mathit{id}_A)(p) = p$.

A5. If $p \parallel p'$ is defined, then its alphabet is $A \cup A'$.

A6. Parallel composition is associative.

A7. Parallel composition is commutative.

The operators have an intuitive correspondence with those of most models of concurrent systems. The operation of renaming corresponds to the instantiation of an agent in a system. Note that since the renaming function is required to be a bijection, renaming is prevented from altering the structure of the agent interface, by for example "connecting" two signals together. Projection corresponds to hiding a set of signals. In fact, the projection operator is here used to *retain* the set of signals that comes as an argument, and hide the remaining signals in the agent. In that sense it corresponds to an operation of scoping. Finally, parallel composition corresponds to the concurrent "execution" of two agents. It is possible to define other operators. We prefer to work with a limited set and add operators only when they cannot be derived from existing ones. The three operators presented here are sufficient for the scope of this work.

A1 through A7 formalize the intuitive behavior of the operators and provide some general properties that we want to be true regardless of the model of computation. These properties, together with the ones required for normalization later in the following chapter, are at the basis of the results of this work. Specifically, A1 asserts that *proj* is effectively hiding the signals not in B from the agent, while A2 says that if all the signals of an agent are retained, then the agent is unchanged. A3 and A4 assert similar properties for the renaming operator, where the identity function on the alphabet results in a no-operation. Finally, A5 through A7 formalize the intuition that parallel composition must be associative and commutative, and requires that the alphabet of the result be obtained as the union of the original alphabets, thus ruling out the possibility of a simultaneous projection. It is important to keep a clear separation between composition and projection, or else the laws of the algebra would become entangled and more difficult to verify.

As described in the above definition, an agent in an agent algebra contains information about what its alphabet is. A simple example of an agent algebra $\mathcal{Q}$ can be constructed by having each agent be nothing more than its alphabet, as follows.

Example 2.7 (Alphabet Algebra). For this example, the master alphabet $\mathcal{Q}.\mathcal{A}$ is an arbitrary set of signal names. The domain $\mathcal{Q}.D$ of the algebra is the set of all subsets of $\mathcal{Q}.\mathcal{A}$. The alphabet of any p in $\mathcal{Q}.D$ is simply p itself. Thus, $\mathcal{Q}.\alpha$ is the identity function. If r is a bijection over $\mathcal{A}$, then $\mathit{rename}(r)(p)$ is defined whenever $p \subseteq \mathit{dom}(r)$, in which case $\mathit{rename}(r)(p)$

is $r(A)$ (where r is naturally extended to sets). If B is a subset of the master alphabet $\mathcal{A}$, then *proj*$(B)(p)$ is $B \cap p$. Finally, $p \parallel p'$ is $p \cup p'$. It is easy to show that A1 through A7 are satisfied.

On the opposite extreme from the previous example is an agent algebra where all the agents have an empty alphabet. Later, we will show how such an agent algebra can be useful by constructing more complex agent algebras in terms of simpler ones.

Example 2.8. This agent algebra can be used to model some quantitative property of an agent, such as maximum power dissipation. The master alphabet $\mathcal{Q}.\mathcal{A}$ is an arbitrary set of signal names. The domain $\mathcal{Q}.D$ of the algebra is the set of non-negative real numbers. For any p in $\mathcal{Q}.D$, the alphabet of p is the empty set. If r is a bijection over $\mathcal{A}$, then *rename*$(r)(p)$ is p. Similarly, if B is a subset of $\mathcal{A}$, then *proj*$(B)(p)$ is p. Finally, $p \parallel p'$ is $p + p'$. Again it is easy to show that the axioms are satisfied.

The agent algebra in example 2.8 illustrates a class of agent algebras which we call *non-alphabetic*, since the agents in the algebra have empty alphabets and *rename* and *proj* are identity functions. This class is formally defined as follows.

Definition 2.9. A *nonalphabetic agent algebra* $\mathcal{Q}$ is an agent algebra with the following properties for any p in $\mathcal{Q}.D$:

1. the alphabet of p is the empty set,
2. if r is a bijection over $\mathcal{Q}.\mathcal{A}$, then *rename*$(r)(p) = p$, and
3. if B is a subset of $\mathcal{Q}.\mathcal{A}$, then *proj*$(B)(p) = p$.

We can use agent algebras to describe the interface that agents expose to their environment, in terms of the input and output signals. The following definitions provide some examples. For all of the examples, it is straightforward to show that the axioms of agent algebras are satisfied. Also, for all algebras, the master alphabet $\mathcal{Q}.\mathcal{A}$ is an arbitrary set of signal names. In chapter 4 we will explore agent algebras that also include a notion of behavior.

Example 2.10 (IO Agent Algebra). Consider the IO agent algebra $\mathcal{Q}$ defined as follows:

- Agents are of the form $p = (I, O)$ where $I \subseteq \mathcal{Q}.\mathcal{A}$, $O \subseteq \mathcal{Q}.\mathcal{A}$ and $I \cap O = \emptyset$. The alphabet of p is $\alpha(p) = I \cup O$.

- *rename*$(r)(p)$ is defined whenever $\alpha(p) \subseteq$ *dom*(r). In that case *rename*$(r)(p) = (r(I), r(O))$, where r is naturally extended to sets.
- *proj*$(B)(p)$ is defined whenever $I \subseteq B$. In that case *proj*$(B)(p) = (I, O \cap B)$.
- $p_1 \parallel p_2$ is defined whenever $O_1 \cap O_2 = \emptyset$. In that case $p_1 \parallel p_2 = ((I_1 \cup I_2) - (O_1 \cup O_2), O_1 \cup O_2)$.

For each agent in this algebra we distinguish between the set of the input signals and the set of the output signals. Notice that parallel composition is defined only when the intersection of the output signals of the agents being composed is empty. In other words, for this algebra we require that each signal in the system be controlled by at most one agent. Notice also that it is impossible to hide input signals. This is required to avoid the case where a signal is not part of the interface of an agent, but it is also not controlled by any other agent (similarly to a floating wire).

The following example is based on the asynchronous trace structure algebra introduced by Dill [33]. Here, we extract from his definitions only the part that concerns the input and output interface of a trace structure. What we obtain is a slightly different notion of input and output algebra.

Example 2.11 (Dill's IO Agent Algebra). Consider the Dill's IO agent algebra $\mathcal{Q}$ defined as follows:

- Agents are of the form $p = (I, O)$ where $I \subseteq \mathcal{Q}.\mathcal{A}$, $O \subseteq \mathcal{Q}.\mathcal{A}$ and $I \cap O = \emptyset$. The alphabet of p is $\alpha(p) = I \cup O$.
- *rename*$(r)(p)$ is defined whenever $\alpha(p) =$ *dom*(r). In that case *rename*$(r)(p) = (r(I), r(O))$.
- *proj*$(B)(p)$ is defined whenever $B \subseteq \alpha(p)$ and $I \subseteq B$. In that case *proj*$(B)(p) = (I, O \cap B)$.
- $p_1 \parallel p_2$ is defined whenever $O_1 \cap O_2 = \emptyset$. In that case $p_1 \parallel p_2 = ((I_1 \cup I_2) - (O_1 \cup O_2), O_1 \cup O_2)$.

The definitions are similar to those in example 2.10, except that the operators of renaming and projection are less often defined. When defined, however, the operators coincide with those in example 2.10.

The above two examples are only concerned with the number and the names of the input and output signals. This is appropriate for models that use signals as pure events. Sometimes signals are associated to a set of values. Many models also include the ability to define a *type* for each signal, that restricts the set of possible values that the signal can take. The following example is a formalization of a valued and typed interface that builds upon example 2.10.

Example 2.12 (Typed IO Agent Algebra). In this example we extend the IO agent algebra described in example 2.10 to contain typing information. Let V be a set of values and 2^V be its powerset. The Typed IO agent algebra $\mathcal{Q}$ is defined as follows:

- Agents are of the form $p = f : \mathcal{Q}.\mathcal{A} \to S$ where

  $$S = \{ c_U \} \cup \{ (c_I, v) : v \subseteq V \} \cup \{ (c_O, v) : v \subseteq V \}.$$

  where c_U, c_I and c_O are constants that denote unused, input and output signals, respectively. The set v that is associated to an input or an output represents the range of values (i.e., the type) that the signal can assume. The alphabet of p is $\alpha(p) = \{ a \in \mathcal{Q}.\mathcal{A} : f(a) \neq c_U \}$. It is also convenient to define the set of inputs, outputs and unused signals as follows:

  $$\begin{aligned} \textit{inputs}(p) &= \{ a \in \mathcal{Q}.\mathcal{A} : f(a) \in \{ c_I \} \times 2^V \} \\ \textit{outputs}(p) &= \{ a \in \mathcal{Q}.\mathcal{A} : f(a) \in \{ c_O \} \times 2^V \} \\ \textit{unused}(p) &= \{ a \in \mathcal{Q}.\mathcal{A} : f(a) = c_U \} \end{aligned}$$

  To simplify the notation we denote the individual components of the function f by $f(a).c$ and $f(a).v$, respectively.

- $\textit{rename}(r)(p)$ is defined whenever $\alpha(p) \subseteq \textit{dom}(r)$. When defined, $\textit{rename}(r)(p) = g$ such that for all $a \in \mathcal{Q}.\mathcal{A}$

  $$g(a) = \begin{cases} f(r^{-1}(a)) & \text{if } r^{-1}(a) \text{ is defined and } r^{-1}(a) \in \alpha(p) \\ c_U & \text{otherwise} \end{cases}$$

- $\textit{proj}(B)(p)$ is always defined and $\textit{proj}(B)(p) = g$ such that for all $a \in \mathcal{Q}.\mathcal{A}$

  $$g(a) = \begin{cases} f(a) & \text{if } a \in \alpha(p) \cap B \\ c_U & \text{otherwise} \end{cases}$$

- $p \parallel p'$ is defined if and only if
  - $\mathit{outputs}(p) \cap \mathit{outputs}(p') = \emptyset$;
  - $f(a).v \subseteq f'(a).v$ whenever $f(a).c = c_O$ and $f'(a).c = c_I$.
  - $f'(a).v \subseteq f(a).v$ whenever $f'(a).c = c_O$ and $f(a).c = c_I$.

  When defined, $p \parallel p' = g$ is such that for all $a \in \mathcal{Q}.\mathcal{A}$

$$g(a) = \begin{cases} f(a) & \text{if } f(a).c = c_O \\ f'(a) & \text{if } f'(a).c = c_O \\ f(a) & \text{if } f'(a).c = c_U \\ f'(a) & \text{if } f(a).c = c_U \\ (c_I, f(a).v \cap f'(a).v) & \text{if } f(a).c = c_I \text{ and } f'(a).c = c_I \end{cases}$$

The definitions are again similar to those in example 2.10. However, the parallel composition operator is restricted to be defined only if the range of values of an output signal is *contained* in the range of values of the corresponding input signal. In addition, if a signal appears as an input in both agents, the range of values for that input in the composition corresponds to the intersection of the original ranges, so that only values consistent with both components can be used when composing with other agents.

2.3 Construction of Algebras

Standard algebraic constructions, such as products and sums, apply to agent algebras. These constructions are useful to build complex agent models out of simpler ones, which could be easier to define and analyze. When defining these construction, however, we must make sure that the axioms of the algebra are satisfied. In this section we define the most relevant constructions and prove that they satisfy the desired properties.

Definition 2.13 (Product). Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be agent algebras with the same master alphabet (i.e., $\mathcal{Q}_1.\mathcal{A} = \mathcal{Q}_2.\mathcal{A}$). The (cross) product of $\mathcal{Q}_1$ and $\mathcal{Q}_2$, written $\mathcal{Q}_1 \times \mathcal{Q}_2$, is the agent algebra $\mathcal{Q}$ such that

1. $\mathcal{Q}.\mathcal{A} = \mathcal{Q}_1.\mathcal{A} = \mathcal{Q}_2.\mathcal{A}$,
2. $\mathcal{Q}.D = \mathcal{Q}_1.D \times \mathcal{Q}_2.D$,

3. $\alpha(\langle p_1, p_2\rangle) = \alpha(p_1) \cup \alpha(p_2)$,

4. $\textit{rename}(r)(\langle p_1, p_2\rangle)$ is defined if and only if $\textit{rename}(r)(p_1)$ and $\textit{rename}(r)(p_2)$ are both defined. In that case,

$$\textit{rename}(r)(\langle p_1, p_2\rangle) = \langle \textit{rename}(r)(p_1), \textit{rename}(r)(p_2)\rangle.$$

5. $\textit{proj}(B)(\langle p_1, p_2\rangle)$ is defined if and only if $\textit{proj}(B)(p_1)$ and $\textit{proj}(B)(p_2)$ are both defined. In that case,

$$\textit{proj}(B)(\langle p_1, p_2\rangle) = \langle \textit{proj}(B)(p_1), \textit{proj}(B)(p_2)\rangle.$$

6. $\langle p_1, p_2\rangle \parallel \langle p'_1, p'_2\rangle$ is defined if and only if $p_1 \parallel p'_1$ and $p_2 \parallel p'_2$ are both defined. In that case,

$$\langle p_1, p_2\rangle \parallel \langle p'_1, p'_2\rangle = \langle p_1 \parallel p'_1, p_2 \parallel p'_2\rangle.$$

The domain of the product is formed by the set of all pairs of agents from the original algebras. As expected, the operators are defined component-wise, and are defined whenever they are defined on the individual components. It is easy to prove that the product of two agent algebras is again an agent algebra.

Theorem 2.14. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be agent algebras, and let $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$ be their product. Then $\mathcal{Q}$ is an agent algebra.

Proof: To prove the validity of the axioms simply apply the definitions and the basic commutative, distributive and associative properties of the operations involved. $\square$

Products of algebras are useful to combine in one single model the expressive power contained in two different models. The following example illustrates this point.

Example 2.15. Recall the agent algebras in example 2.7 and example 2.8, which have domains of $2^{\mathcal{A}}$ and the non-negative real numbers, respectively. The agents of their cross product are of the form $\langle A, w\rangle$, where A is an alphabet over $\mathcal{A}$, and w is a non-negative real. The cross product of these two agent algebras thus combines the information of the two individual algebras.

A construction similar to the product is the disjoint sum of two algebras.

Definition 2.16 (Disjoint Sum). Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be agent algebras with master alphabet $\mathcal{A}_1 = \mathcal{Q}_1.\mathcal{A}$ and $\mathcal{A}_2 = \mathcal{Q}_2.\mathcal{A}$, respectively. The disjoint sum of $\mathcal{Q}_1$ and $\mathcal{Q}_2$, written $\mathcal{Q}_1 \uplus \mathcal{Q}_2$, is the agent algebra $\mathcal{Q}$ such that

1. $\mathcal{Q}.\mathcal{A} = \mathcal{Q}_1.\mathcal{A} \uplus \mathcal{Q}_2.\mathcal{A}$,

2. $\mathcal{Q}.D = \mathcal{Q}_1.D \uplus \mathcal{Q}_2.D$,

3. $\alpha(p) = \begin{cases} \mathcal{Q}_1.\alpha(p) & \text{if } p \in \mathcal{Q}_1.D \\ \mathcal{Q}_2.\alpha(p) & \text{if } p \in \mathcal{Q}_2.D \end{cases}$

4. $\mathit{rename}(r)(p) = \begin{cases} \mathcal{Q}_1.\mathit{rename}(r)(p) & \text{if } p \in \mathcal{Q}_1.D \\ \mathcal{Q}_2.\mathit{rename}(r)(p) & \text{if } p \in \mathcal{Q}_2.D \end{cases}$

5. $\mathit{proj}(B)(p) = \begin{cases} \mathcal{Q}_1.\mathit{proj}(B)(p) & \text{if } p \in \mathcal{Q}_1.D \\ \mathcal{Q}_2.\mathit{proj}(B)(p) & \text{if } p \in \mathcal{Q}_2.D \end{cases}$

6. $p \parallel p' = \begin{cases} p \parallel p' & \text{if both } p \in \mathcal{Q}_1.D \text{ and } p' \in \mathcal{Q}_1.D \\ p \parallel p' & \text{if both } p \in \mathcal{Q}_2.D \text{ and } p' \in \mathcal{Q}_2.D \\ \uparrow & \text{otherwise} \end{cases}$

In a disjoint sum, the algebras being composed are simply placed side by side to form a new algebra. The agents of each algebra, however, have no interaction with the agents of the other algebra. For this reason the rest of this work will concentrate on products of algebras. Nonetheless, it is easy to show that the disjoint sum of agent algebras is again an agent algebra.

Theorem 2.17. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be agent algebras, and let $\mathcal{Q} = \mathcal{Q}_1 \uplus \mathcal{Q}_2$ be their disjoint sum. Then $\mathcal{Q}$ is an agent algebra.

If $\mathcal{Q}'$ is an agent algebra and $D \subseteq D'$ is a subset of the agents that is closed in D' under the application of the operators, then D can be used as the domain of a subalgebra $\mathcal{Q}$ of $\mathcal{Q}'$.

Definition 2.18 (Subalgebra). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be agent algebras over the same master alphabet $\mathcal{A}$. Then $\mathcal{Q}$ is called a *subalgebra* of $\mathcal{Q}'$, written $\mathcal{Q} \subseteq \mathcal{Q}'$, if and only if

1. $\mathcal{Q}.D \subseteq \mathcal{Q}'.D$

2. The operators of projection, renaming and parallel composition in $\mathcal{Q}$ are the restrictions to $\mathcal{Q}.D$ of the operators of $\mathcal{Q}'$.

Clearly, the above definition implies that $\mathcal{Q}.D$ is closed in $\mathcal{Q}'.D$ under the application of the operations of agent algebra. Conversely, every subset of $\mathcal{Q}'.D$ that is closed under the application of the operations is the domain of a subalgebra $\mathcal{Q}$ when the operators are the restriction to the subset of the corresponding operators in $\mathcal{Q}'$. It is easy to show that $\mathcal{Q}$ is indeed an agent algebra. The result follows from the fact that the axioms are valid in the substructure, since A1 to A7 are true of all agents in the superalgebra, and therefore must be true of all agents in the subalgebra. The following is an interesting example of this fact.

Theorem 2.19. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be agent algebras, and let $\mathcal{Q}' = \mathcal{Q}_1 \times \mathcal{Q}_2$ be their cross product. Consider the subset $S \subseteq \mathcal{Q}'.D$ such that for all agents $\langle p_1, p_2 \rangle \in S$, $\alpha(p_1) = \alpha(p_2)$. Then S is closed in $\mathcal{Q}'.D$ under the operations of projection, renaming and parallel composition.

Proof: Let $p = \langle p_1, p_2 \rangle$ and $q = \langle q_1, q_2 \rangle$ be elements of S. The proof is composed of the following cases.

- Assume $p \in S$ and assume $\mathit{proj}(B)(p)$ is defined in $\mathcal{Q}'$. Then,

$$p \in S$$

by hypothesis

$$\Leftrightarrow \quad \alpha(p_1) = \alpha(p_2)$$

by A1

$$\Rightarrow \quad \alpha(\mathit{proj}(B)(p_1)) = \alpha(\mathit{proj}(B)(p_2))$$

$$\Rightarrow \quad \mathit{proj}(B)(p) = \langle \mathit{proj}(B)(p_1), \mathit{proj}(B)(p_2) \rangle \in S$$

- Assume $p \in S$ and assume $\mathit{rename}(r)(p)$ is defined in $\mathcal{Q}'$. Then,

$$p \in S$$

$$\Leftrightarrow \quad \alpha(p_1) = \alpha(p_2)$$

by A3

$$\Rightarrow \quad \alpha(\mathit{rename}(r)(p_1)) = \alpha(\mathit{rename}(r)(p_2))$$

$$\Rightarrow \quad \mathit{rename}(r)(p) = \langle \mathit{rename}(r)(p_1), \mathit{rename}(r)(p_2) \rangle \in S$$

- Assume $p \in S$, $q \in S$, and assume $p \parallel q$ is defined in $\mathcal{Q}'$. Then,

$$\begin{aligned}
& p \in S \wedge q \in S \\
& \quad \Leftrightarrow \quad \alpha(p_1) = \alpha(p_2) \wedge \alpha(q_1) = \alpha(q_2) \\
& \quad\ \ \text{by A5} \\
& \quad \Rightarrow \quad \alpha(p_1 \parallel q_1) = \alpha(p_2 \parallel q_2) \\
& \quad \Rightarrow \quad p \parallel q = \langle p_1 \parallel q_1, p_2 \parallel q_2 \rangle \in S
\end{aligned}$$

□

Since S is closed, the algebra $\mathcal{Q}$ that has the set S (the pairs of agents that have the same alphabet) as the domain, and the restriction to S of the operators of $\mathcal{Q}'$ as the operators, is a subalgebra of $\mathcal{Q}_1 \times \mathcal{Q}_2$.

2.4 Ordered Agent Algebras

To study the concepts of refinement and conservative approximations, we can add a preorder or a partial order to an agent algebra. The result is called a *preordered agent algebra* or a *partially ordered agent algebra*, respectively.

We require that the functions in an ordered agent algebra be monotonic relative to the ordering. However, since these are partial functions, this requires generalizing monotonicity to partial functions. The following definition gives two different generalizations. Later we discuss which of these best suits our needs.

Definition 2.20 ($\top$-$\bot$-monotonic). Let D_1 and D_2 be preordered sets. Let f be a partial function from D_1 to D_2. Let

$$\begin{aligned}
D_2^{\top} &= D_2 \cup \{\top\} \\
D_2^{\bot} &= D_2 \cup \{\bot\},
\end{aligned}$$

where $\top$ and $\bot$ are not elements of D_2. The preorders over $D_2^{\top}$ and $D_2^{\bot}$ are the extensions of the preorder over D_2 such that

$$p_2 \preceq \top \wedge \top \not\preceq p_2$$

and

$$p_2 \not\preceq \bot \wedge \bot \preceq p_2,$$

respectively, for every p_2 in D_2. Let $f_\top$ and $f_\bot$ be the total functions from D_1 to $D_2^\top$ and $D_2^\bot$, respectively, such that for all p_1 in D_1

$$f_\top(p_1) = \begin{cases} f(p_1), & \text{if } f(p_1) \text{ is defined;} \\ \top, & \text{otherwise;} \end{cases}$$

$$f_\bot(p_1) = \begin{cases} f(p_1), & \text{if } f(p_1) \text{ is defined;} \\ \bot, & \text{otherwise.} \end{cases}$$

We say the function f is $\top$-*monotonic* if and only if $f_\top$ is monotonic. Analogously, the function f is $\bot$-*monotonic* if and only if $f_\bot$ is monotonic.

In this work we always interpret the formula $p \preceq p'$ to mean intuitively that p can be substituted for p' in any context. Let f be a partial function and assume $f(p)$ is undefined. Then intuitively, $f(p)$ cannot be substituted for any other agent in D_2, except for another undefined expression. This is always the case if f is $\top$-monotonic. In that case, in fact, $f(p) = \top$ and $f(p) \preceq f(p')$ together imply that $f(p') = \top$, i.e., $f(p')$ is also undefined. A $\top$-monotonic function is therefore consistent with our interpretation of the order. Thus, an undefined expression should be treated as a maximal element relative to the ordering. Therefore, we require that functions in ordered agent algebras be $\top$-monotonic.

Definition 2.21 (Ordered Agent Algebra). A *preordered (partially ordered) agent algebra* is an agent algebra $\mathcal{Q}$ with a preorder (partial order) $\mathcal{Q}.\preceq$ such that for all alphabets B over $\mathcal{Q}.\mathcal{A}$ and all bijections r over $\mathcal{Q}.\mathcal{A}$, the partial functions $\mathcal{Q}.\mathit{rename}(r)$, $\mathcal{Q}.\mathit{proj}(B)$ and $\mathcal{Q}.\|$ are $\top$-monotonic. The preorder (partial order) $\mathcal{Q}.\preceq$ is called the *agent order* of $\mathcal{Q}$.

Our most general definition uses preorders, rather than partial orders, because a relation of substitutability cannot in general be required to be antisymmetric. As usual, however, preorders induce a natural equivalence relation on the underlying set and a natural partial order on the equivalence classes.

Definition 2.22 (Order Equivalence). Let $\mathcal{Q}$ be a preordered agent algebra. We define the relation "$\approx$" to be the equivalence relation induced by the preorder "$\preceq$". That is

$$p \approx q \Leftrightarrow p \preceq q \wedge q \preceq p.$$

If $p \approx q$ we say that p and q are order equivalent. Order equivalence and equality are the same if the agent algebra is partially ordered.

Corollary 2.23. If $\mathcal{Q}$ be a partially ordered agent algebra, then

$$p \approx q \Leftrightarrow p = q.$$

The parallel composition operator is the basis of compositional methods for both design and verification. Monotonicity is required for these methods to work correctly. Henzinger et al. [30] propose to distinguish between *interface* and *component* algebras. Corollary 2.25 below shows that because parallel composition is $\top$-monotonic in an ordered agent algebra, it supports an inference rule identical to the "compositional design" rule for interface algebras. Similarly, component algebras have a "compositional verification" rule that corresponds to $\perp$-monotonic functions. This suggests that the ordering of a component algebra cannot be interpreted as indicating substitutability.

Theorem 2.24. Let f be a $\top$-monotonic partial function. If $p \preceq p'$ and $f(p')$ is defined, then $f(p)$ is defined and $f(p) \preceq f(p')$.

Proof: Let $f_\top$ be as described in definition 2.20. Assume $p \preceq p'$ and $f(p')$ is defined. To show by contradiction that $f(p)$ is defined, start by assuming otherwise. Then, $f_\top(p)$ is equal to $\top$ and $f_\top(p')$ is not. This leads to a contradiction since $p \preceq p'$ and $f_\top$ is monotonic. Also, since $f(p)$ and $f(p')$ are defined, it follows easily from the monotonicity of $f_\top$ that $f(p) \preceq f(p')$. □

Corollary 2.25. Let $\|$ be the composition function of a preordered agent algebra. If $p_1 \preceq p_1'$, $p_2 \preceq p_2'$ and $p_1' \| p_2'$ is defined, then $p_1 \| p_2$ is defined and $p_1 \| p_2 \preceq p_1' \| p_2'$.

Proof: Since $\|$ is $\top$-monotonic by the definition of a preordered agent algebra (definition 2.21), this is simply specializing theorem 2.24 to a binary function. □

The rest of this section is devoted to examples. For each example we derive necessary conditions that the order must satisfy in order for the operators to be $\top$-monotonic. We then choose a particular order, and show that the operators are in fact $\top$-monotonic relative to the order.

Example 2.26 (Alphabet Algebra). Consider the alphabet agent algebra $\mathcal{Q}$ described in example 2.7. The condition of $\top$-monotonicity imposes restrictions on the kind of orders that can be employed in the algebra. In this particular case, the order must be such that $p \preceq p'$ only if $p \subseteq p'$.

Theorem 2.27. Let $\preceq$ be an order for $\mathcal{Q}$ (example 2.7) such that *rename*, *proj* and $\|$ are $\top$-monotonic. Then $p \preceq p'$ only if $p \subseteq p'$.

Proof: Let $p \preceq p'$ and let r be a renaming function such that $p' = \mathit{dom}(r)$. Then clearly $\mathit{rename}(r)(p')$ is defined. Since *rename* is $\top$-monotonic, also $\mathit{rename}(r)(p)$ is defined. Therefore $p \subseteq \mathit{dom}(r) = p'$. □

The above result only provides a necessary condition on the order so that the operators are $\top$-monotonic. Any particular choice of order must still be shown to make the operators $\top$-monotonic. Consider, for instance, the order $\preceq$ that corresponds exactly to $\subseteq$, that is $p \preceq p'$ if and only if $p \subseteq p'$. Then

Theorem 2.28. The operators *rename*, *proj* and $\|$ are $\top$-monotonic with respect to $\subseteq$.

Proof: Let $p \subseteq p'$.

- Assume $\mathit{rename}(r)(p')$ is defined. Then $p' \subseteq \mathit{dom}(r)$. Thus, since $p \subseteq p'$, also $p \subseteq \mathit{dom}(r)$, so that $\mathit{rename}(r)(p)$ is defined. In addition since r is a bijection and $p \subseteq p'$

$$\mathit{rename}(r)(p) = r(p) \subseteq r(p') = \mathit{rename}(r)(p')$$

- Let B be a subset of $\mathcal{A}$. Then $\mathit{proj}(B)(p')$ and $\mathit{proj}(B)(p)$ are both defined. In addition, since $p \subseteq p'$,

$$\mathit{proj}(B)(p) = p \cap B \subseteq p' \cap B = \mathit{proj}(B)(p').$$

- Let q be an agent. Then $p' \| q$ and $p \| q$ are both defined. In addition since $p \subseteq p'$

$$p \| q = p \cup q \subseteq p' \cup q = p' \| q.$$

□

Example 2.29 (IO Agent Algebra). Consider the IO agent algebra $\mathcal{Q}$ defined in example 2.10. The requirement that the functions be $\top$-monotonic places a corresponding requirement on the order that can be defined in the algebra.

Theorem 2.30. Let $\preceq$ be an order for $\mathcal{Q}$ (example 2.10) such that *rename*, *proj* and $\|$ are $\top$-monotonic. Then $p \preceq p'$ only if $I \subseteq I'$ and $O = O'$.

Proof: Let $p \preceq p'$.

- We first prove that $I \subseteq I'$. Since $I' \subseteq I'$, then *proj*$(I')(p')$ is defined. Since *proj* is $\top$-monotonic, then also *proj*$(I')(p)$ must be defined. Therefore it must be $I \subseteq I'$.

- We now prove that $O \subseteq O'$. Assume, by contradiction, that there exists $o \in O$ such that $o \notin O'$. Consider $q = (O', I' \cup \{o\})$. Then $p' \parallel q$ is defined because $O' \cap (I' \cup \{o\}) = \emptyset$ since by hypothesis $O' \cap I' = \emptyset$ and $o \notin O'$. Since $\parallel$ is $\top$-monotonic then also $p \parallel q$ must be defined. But then it must be $O \cap (I' \cup \{o\}) = \emptyset$, which implies $o \notin O$, a contradiction. Hence $O \subseteq O'$.

- Finally we prove that $O' \subseteq O$. Consider the agent $q = (O', I')$. Since by definition $O' \cap I' = \emptyset$, then $p' \parallel q$ is defined and

  $$p' \parallel q = ((I' \cup O') - (O' \cup I'), O' \cup I') = (\emptyset, O' \cup I').$$

  Since $\parallel$ is $\top$-monotonic, then also $p \parallel q$ is defined and

  $$p \parallel q = ((I \cup O') - (O \cup I'), O \cup I').$$

  Since $\parallel$ is $\top$-monotonic it must be $p \parallel q \preceq p' \parallel q$. Since *proj* is $\top$-monotonic, it must be

  $$(I \cup O') - (O \cup I') \subseteq \emptyset$$
  $$(I \cup O') - (O \cup I') = \emptyset$$
  $$(I \cup O') \subseteq (O \cup I')$$
  Since $I \subseteq I'$
  $$O' \subseteq (O \cup I')$$
  Since $O' \cap I' = \emptyset$
  $$O' \subseteq O.$$

□

The converse is not true. That is, it is not the case that if $\preceq$ is an order for $\mathcal{Q}$ such that $p \preceq p'$ only if $I \subseteq I'$ and $O = O'$, then the operators *rename*, *proj* and $\parallel$ are $\top$-monotonic.

For example, assume $\mathit{rename}(r)(p')$ is defined. Then we can show that $\mathit{rename}(r)(p)$ is defined. However, since we don't have sufficient conditions for the ordering, the hypothesis are insufficient to show that $\mathit{rename}(r)(p) \preceq \mathit{rename}(r)(p')$. Similarly for the other functions in the algebra.

For the purpose of this example we choose the order $\preceq$ so that $p \preceq p'$ if and only if $I \subseteq I'$ and $O = O'$.

Theorem 2.31. The functions *rename*, *proj* and $\|$ are $\top$-monotonic with respect to $\preceq$.

Proof: Let $p \preceq p'$.

- Assume $\mathit{rename}(r)(p')$ is defined. Then $A' \subseteq \mathit{dom}(r)$. By hypothesis, $A \subseteq A'$, so that $A \subseteq \mathit{dom}(r)$. Therefore $\mathit{rename}(r)(p)$ is defined. Since r is a bijection

  $$\begin{array}{rcl} I \subseteq I' & \Rightarrow & r(I) \subseteq r(I') \\ O = O' & \Rightarrow & r(O) = r(O') \end{array}$$

  Hence $\mathit{rename}(r)(p) \preceq \mathit{rename}(r)(p')$.

- Assume $\mathit{proj}(B)(p')$ is defined. Then $I' \subseteq B$. By hypothesis, $I \subseteq I'$, so that $I \subseteq B$. Therefore $\mathit{proj}(B)(p)$ is defined. In addition

  $$\begin{array}{rcl} I \subseteq I' & \Rightarrow & I \subseteq I' \\ O = O' & \Rightarrow & O \cap B = O' \cap B. \end{array}$$

  Hence $\mathit{proj}(B)(p) \preceq \mathit{proj}(B)(p')$.

- Assume $p' \| q$ is defined, where $q = (I_q, O_q)$. Then $O' \cap O_q = \emptyset$. By hypothesis, $O = O'$ so that $O \cap O_q = \emptyset$. Therefore $p \| q$ is defined. In addition

  $$\begin{array}{rcl} p' \| q & = & ((I' \cup I_q) - (O' \cup O_q), O' \cup O_q) \\ p \| q & = & ((I \cup I_q) - (O \cup O_q), O \cup O_q) \end{array}$$

  Clearly since $O = O'$

  $$O \cup O_q = O' \cup O_q.$$

Therefore, since $I \subseteq I'$

$$(I \cup I_q) - (O' \cup O_q) \subseteq (I' \cup I_q) - (O' \cup O_q).$$

Hence $p \parallel q \preceq p' \parallel q$.

$\square$

Example 2.32 (Dill's IO Agent Algebra). Consider now the Dill style IO agents described in example 2.11. Because the *rename* operator has a further restriction that the domain of the renaming function r be equal to the alphabet of the agent being renamed, the order that results in $\top$-monotonic function is completely determined. The following theorem proves this fact.

Theorem 2.33. Let $\preceq$ be an order for $\mathcal{Q}$ (example 2.11). Then *rename*, *proj* and $\parallel$ are $\top$-monotonic with respect to $\preceq$ if and only if for all agents p and p', $p \preceq p'$ if and only if $p = p'$.

Proof: For the forward direction, assume that $\preceq$ is an order such that the functions are $\top$-monotonic. Let $p = (I, O)$ and $p' = (I', O')$ be two agents. We must show that $p \preceq p'$ if and only if $p = p'$. Clearly, if $p = p'$, then $p \preceq p'$, since $\preceq$ is reflexive. Conversely, assume $p \preceq p'$. We must show that $p = p'$. To do so, we first show that $A = A'$. In fact, $\mathit{dom}(\mathit{id}_{A'}) = A' = \alpha(p')$ and therefore $\mathit{rename}(\mathit{id}_{A'})(p')$ is defined. Since *rename* is $\top$-monotonic, and since $p \preceq p'$, by theorem 2.24, also $\mathit{rename}(\mathit{id}_{A'})(p)$ is defined. Thus $A = \alpha(p) = \mathit{dom}(\mathit{id}_{A'}) = A'$.

We then show that $I \subseteq I'$. In fact, $I' \subseteq A'$ and $I' \subseteq I'$ imply that $\mathit{proj}(I')(p')$ is defined. Since *proj* is $\top$-monotonic, also $\mathit{proj}(I')(p)$ is defined. Thus $I' \subseteq A$ and $I \subseteq I'$.

Finally we show that $I = I'$ and $O = O'$. In fact, by definition of agent $O' \cap I' = \emptyset$ and therefore $p' \parallel (O', I')$ is defined. Thus, since $\parallel$ is $\top$-monotonic, also $p \parallel (O', I')$ is defined. Therefore $O \cap I' = \emptyset$. But since, by the above arguments, $O \cup I = O' \cup I'$, also $I' \subseteq I$, and thus $I = I'$. Therefore it must also be $O = O'$.

The reverse direction is trivial, since any function is $\top$-monotonic relative to the equality. $\square$

Thus for this example we must choose the order such that $p \preceq p'$ if and only if $I = I'$ and $O = O'$.

Example 2.34 (Typed IO Agent Algebra). Consider the Typed IO agent algebra $\mathcal{Q}$ defined in example 2.12. As for IO agents, $\top$-monotonicity restricts the set of orders that can be applied to the algebra.

Theorem 2.35. Let $\preceq$ be an order for $\mathcal{Q}$ (example 2.12) such that *rename*, *proj* and $\|$ are $\top$-monotonic. Then $p \preceq p'$ only if $\mathit{inputs}(p) \subseteq \mathit{inputs}(p')$ and $\mathit{outputs}(p) = \mathit{outputs}(p')$, and for all $a \in \mathcal{Q}.\mathcal{A}$, if $f(a).c = c_I$ then $f(a).v \supseteq f'(a).v$, and if $f(a).c = c_O$ then $f(a).v \subseteq f'(a).v$.

Proof: It is easy to adapt the proof of theorem 2.30 to show that $p \preceq p'$ only if

$$\begin{aligned} \mathit{inputs}(p) &\subseteq \mathit{inputs}(p') \\ \mathit{outputs}(p) &= \mathit{outputs}(p') \end{aligned}$$

To prove the rest of the theorem, let $p \preceq p'$ and let $q = f_q$ be the agent such that for all $a \in \mathcal{Q}.\mathcal{A}$

$$f_q(a) = \begin{cases} (c_O, v) & \text{if } f'(a) = (c_I, v) \\ (c_I, v) & \text{if } f'(a) = (c_O, v) \\ c_U & \text{otherwise} \end{cases}$$

so that $\mathit{inputs}(q) = \mathit{outputs}(p')$ and $\mathit{outputs}(q) = \mathit{inputs}(p')$. Then clearly $p' \| q$ is defined. Since $\|$ is $\top$-monotonic, $p \| q$ must also be defined. In fact, since $\mathit{outputs}(p) = \mathit{outputs}(p')$ we already know that $\mathit{outputs}(p) \cap \mathit{outputs}(q) = \emptyset$. Assume now that $a \in \mathcal{Q}.\mathcal{A}$ and $f(a).c = c_I$. Then also $f'(a).c = c_I$, and $f_q(a).c = c_O$. Hence, since $p \| q$ is defined, $f_q(a).v \subseteq f(a).v$. But $f_q(a).v = f'(a).v$, thus $f(a).v \supseteq f'(a).v$.

Similarly, assume that $a \in \mathcal{Q}.\mathcal{A}$ and $f(a).c = c_O$. Then also $f'(a).c = c_O$, and $f_q(a).c = c_I$. Hence, since $p \| q$ is defined, $f(a).v \subseteq f_q(a).v$. But $f_q(a).v = f'(a).v$, thus $f(a).v \subseteq f'(a).v$. □

Given this result, we choose to order the Typed IO agents so that $p \preceq p'$ if and only if $\mathit{inputs}(p) \subseteq \mathit{inputs}(p')$ and $\mathit{outputs}(p) = \mathit{outputs}(p')$, and for all $a \in \mathcal{Q}.\mathcal{A}$, if $a \in \mathit{inputs}(p)$ then $f(a).v \supseteq f'(a).v$, and if $a \in \mathit{outputs}(p)$ then $f(a).v \subseteq f'(a).v$.

Theorem 2.36. The operations of *rename*, *proj* and $\|$ are $\top$-monotonic with respect to $\preceq$.

Proof: The proof of this theorem is similar to the proof of theorem 2.31 and is left as an exercise. □

2.4.1 Construction of Algebras

In section 2.3 we have introduced several constructions used to create new algebras from existing ones. In this section we extend those constructions to include the agent order.

The order in the product is the usual point-wise extension.

Definition 2.37 (Product - Order). Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be ordered agent algebras with the same master alphabet. The product of $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$ is the ordered agent algebra defined as in definition 2.13 with the order such that

$$\langle p_1, p_2 \rangle \preceq \langle p_1', p_2' \rangle \Leftrightarrow p_1 \preceq p_1' \wedge p_2 \preceq p_2'.$$

Theorem 2.38. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be ordered agent algebras, and let $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$ be their product. Then $\mathcal{Q}$ is an ordered agent algebra.

Proof: We must show that the operators are $\top$-monotonic. Here we only show the case for projection, since the other cases are similar.

Let $\langle p_1, p_2 \rangle \preceq \langle p_1', p_2' \rangle$, and assume $\textit{proj}(B)(\langle p_1', p_2' \rangle)$ is defined. Then, by definition of product, both $\textit{proj}(B)(p_1')$ and $\textit{proj}(B)(p_2')$ are defined. By definition 2.37, since $\langle p_1, p_2 \rangle \preceq \langle p_1', p_2' \rangle$, also $p_1 \preceq p_1'$ and $p_2 \preceq p_2'$. Thus, since *proj* is $\top$-monotonic in $\mathcal{Q}_1$ and $\mathcal{Q}_2$, $\textit{proj}(B)(p_1)$ and $\textit{proj}(B)(p_2)$ are defined, and

$$\textit{proj}(B)(p_1) \preceq \textit{proj}(B)(p_1') \wedge \textit{proj}(B)(p_2) \preceq \textit{proj}(B)(p_2').$$

Therefore, by definition 2.37, also $\textit{proj}(B)(\langle p_1, p_2 \rangle)$ is defined and

$$\textit{proj}(B)(\langle p_1, p_2 \rangle) \preceq \textit{proj}(B)(\langle p_1', p_2' \rangle).$$

Hence *proj* is $\top$-monotonic in $\mathcal{Q}$. $\square$

The order in the disjoint sum corresponds to the orders in the components, and agents that do not belong to the same algebra are otherwise unrelated.

Definition 2.39 (Disjoint Sum - Order). Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be ordered agent algebras. The disjoint sum of $\mathcal{Q} = \mathcal{Q}_1 \uplus \mathcal{Q}_2$ is defined as in definition 2.16 with the order such that $p \preceq p'$ if and only if either

$$p \in \mathcal{Q}_1.D \wedge p' \in \mathcal{Q}_1.D \wedge p \preceq_{\mathcal{Q}_1} p',$$

or

$$p \in \mathcal{Q}_2.D \wedge p' \in \mathcal{Q}_2.D \wedge p \preceq_{\mathcal{Q}_2} p'.$$

The fact that the disjoint sum is an ordered agent algebra follows easily from the definitions, as stated in the next theorem.

Theorem 2.40. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be ordered agent algebras, and let $\mathcal{Q} = \mathcal{Q}_1 \uplus \mathcal{Q}_2$ be their product. Then $\mathcal{Q}$ is an ordered agent algebra.

Agents in a subalgebra are ordered exactly as in the superalgebra. Since the domain of the subalgebra is also closed under the application of the operators, it is not surprising that the subalgebra is again an ordered agent algebra.

Definition 2.41 (Subalgebra - Order). Let $\mathcal{Q}'$ be an ordered agent algebra. The agent algebra $\mathcal{Q}$ is a subalgebra of $\mathcal{Q}'$ if and only if

- $\mathcal{Q}$ is a subalgebra of $\mathcal{Q}'$ (definition 2.18), and
- for all $p, p' \in \mathcal{Q}.D$, $p \preceq_{\mathcal{Q}} p'$ if and only if $p \preceq_{\mathcal{Q}'} p'$.

Theorem 2.42. Let $\mathcal{Q}'$ be an ordered agent algebra and let $\mathcal{Q} \subseteq \mathcal{Q}'$. Then $\mathcal{Q}$ is an ordered agent algebra.

2.5 Agent Expressions

As is customary in the study of algebraic systems, we can define expressions in terms of the operators that are defined in an agent algebra. In this section we define agent expressions and define their semantics.

Definition 2.43 (Agent Expressions). Let V be a set of variables, and let $\mathcal{Q}$ be an agent algebra. The set of agent expressions over $\mathcal{Q}$ is the least set $\mathcal{E}$ satisfying the following conditions:

Constant If $p \in \mathcal{Q}.D$, then $p \in \mathcal{E}$.

Variable If $v \in V$, then $v \in \mathcal{E}$.

Projection If $E \in \mathcal{E}$ and B is an alphabet, then $\mathit{proj}(B)(E) \in \mathcal{E}$.

Renaming If $E \in \mathcal{E}$ and r is a renaming function, then $\mathit{rename}(r)(E) \in \mathcal{E}$.

Parallel Composition If $E_1 \in \mathcal{E}$ and $E_2 \in \mathcal{E}$, then $E_1 \parallel E_2 \in \mathcal{E}$.

We denote by $\mathit{sub}(E)$ the set of all subexpressions of E, including E.

Agent expressions have no binding constructs (e.g., quantifiers). Therefore every variable in an agent expression is free. The set of free variables of an agent expressions can be defined by induction on the structure of expressions as follows.

Definition 2.44 (Free variables). Let E be an agent expression over $\mathcal{Q}$. The set $\mathit{FV}(E)$ of free variables of E is

- If $E = p$ for some $p \in \mathcal{Q}.D$, then $\mathit{FV}(E) = \emptyset$.
- If $E = v$ for some $v \in V$ then $\mathit{FV}(E) = \{ v \}$.
- If $E = \mathit{proj}(B)(E_1)$ for some agent expression E_1 then $\mathit{FV}(E) = \mathit{FV}(E_1)$.
- If $E = \mathit{rename}(r)(E_1)$ for some agent expression E_1 then $\mathit{FV}(E) = \mathit{FV}(E_1)$.
- If $E = E_1 \parallel E_2$ for some agent expressions E_1 and E_2 then $\mathit{FV}(E) = \mathit{FV}(E_1) \cup \mathit{FV}(E_2)$.

We call an expression that has no free variables a *closed expression*.

Intuitively, an agent expression represents a particular agent in the underlying agent algebra once the variables have been given a value. Hence, to define the semantics of agent expressions we must first describe an assignment to the variables.

Definition 2.45 (Assignment). Let $\mathcal{Q}$ be an agent algebra and let V be a set of variables. An assignment of V on $\mathcal{Q}$ is a function $\sigma : V \mapsto \mathcal{Q}.D$.

The denotation $[\![E]\!]$ of an expression E is a function that takes an assignment σ and produces a particular agent in the agent algebra. Note however that since the operators in the agent algebra are partial functions, the denotation of an expression is also a partial function. The semantic function, the one that to each expression E associates the denotation $[\![E]\!]$ is, of course, a total function.

Definition 2.46 (Expression Evaluation). Let Σ be the set of all assignments. The denotation of agent expressions is given by the function $[\![-]\!] : \mathcal{E} \mapsto \Sigma \to \mathcal{Q}.D$ defined for each assignment $\sigma \in \Sigma$ by the following semantic equations:

- If $E = p$ for some $p \in \mathcal{Q}.D$, then $[\![E]\!]\sigma = p$.

- If $E = v$ for some $v \in V$ then $[\![E]\!]\sigma = \sigma(v)$.
- If $E = \mathit{proj}(B)(E_1)$ for some expression E_1 then $[\![E]\!]\sigma = \mathit{proj}(B)([\![E_1]\!]\sigma)$ if $[\![E_1]\!]\sigma$ is defined and $\mathit{proj}(B)([\![E_1]\!]\sigma)$ is defined. Otherwise $[\![E]\!]\sigma$ is undefined.
- If $E = \mathit{rename}(r)(E_1)$ for some expression E_1 then $[\![E]\!]\sigma = \mathit{rename}(r)([\![E_1]\!]\sigma)$ if $[\![E_1]\!]\sigma$ is defined and $\mathit{rename}(r)([\![E_1]\!]\sigma)$ is defined. Otherwise $[\![E]\!]\sigma$ is undefined.
- If $E = E_1 \parallel E_2$ for some expressions E_1 and E_2 then $[\![E]\!]\sigma = [\![E_1]\!]\sigma \parallel [\![E_2]\!]\sigma$ if both $[\![E_1]\!]\sigma$ and $[\![E_2]\!]\sigma$ are defined and $[\![E_1]\!]\sigma \parallel [\![E_2]\!]\sigma$ is defined. Otherwise $[\![E]\!]\sigma$ is undefined.

The following equivalent definition of expression evaluation highlights the fact that the semantic equations are syntax directed.

$$\begin{aligned}
[\![p]\!]\sigma &= p \\
[\![v]\!]\sigma &= \sigma(v) \\
[\![\mathit{proj}(B)(E)]\!]\sigma &= \begin{cases} \mathit{proj}(B)([\![E]\!]\sigma) & \text{if } [\![E]\!]\sigma\downarrow \text{ and } \mathit{proj}(B)([\![E]\!]\sigma)\downarrow \\ \uparrow & \text{otherwise} \end{cases} \\
[\![\mathit{rename}(r)(E)]\!]\sigma &= \begin{cases} \mathit{rename}(r)([\![E]\!]\sigma) & \text{if } [\![E]\!]\sigma\downarrow \text{ and } \mathit{rename}(r)([\![E]\!]\sigma)\downarrow \\ \uparrow & \text{otherwise} \end{cases} \\
[\![E_1 \parallel E_2]\!]\sigma &= \begin{cases} [\![E_1]\!]\sigma \parallel [\![E_2]\!]\sigma & \text{if } [\![E_1]\!]\sigma\downarrow, [\![E_2]\!]\sigma\downarrow \text{ and } [\![E_1]\!]\sigma \parallel [\![E_2]\!]\sigma\downarrow \\ \uparrow & \text{otherwise} \end{cases}
\end{aligned}$$

Since the semantic equations are syntax directed, the solution exists and is unique [78]. We extend the semantic function to sets of expressions and sets of assignments as follows.

Definition 2.47. Let $\mathcal{E}$ be a set of expressions and Σ' a set of assignments. We denote the possible evaluations of the expressions in $\mathcal{E}$ under the assignments in Σ as

$$[\![\mathcal{E}]\!]\Sigma' = \{ [\![E]\!]\sigma : E \in \mathcal{E} \text{ and } \sigma \in \Sigma' \}.$$

Clearly, the value of an agent expression depends only on the value assigned by the assignment σ to the free variables.

Lemma 2.48 (Coincidence Lemma). Let E be an expression, and let σ_1 and σ_2 be two assignments such that for all $v \in FV(E)$, $\sigma_1(v) = \sigma_2(v)$. Then

$$[\![E]\!]\sigma_1 = [\![E]\!]\sigma_2.$$

Since an agent expression involves only a finite number of free variables, we use the notation $E[v_1, \ldots, v_n]$ to denote that E has free variables $v_1, \ldots, v_n$. In that case, we use the notation $E[p_1, \ldots, p_n]$ for $[\![\, E \,]\!]\sigma$ where $\sigma(v_i) = p_i$ for $1 \leq i \leq n$. Note also that if an agent expression has no free variables its value does not depend on the assignment σ.

When an expression contains variables it is possible to substitute another expression for the variable. [1]

Definition 2.49 (Expression Substitution). Let $\mathcal{Q}$ be an agent algebra and let E and E' be a agent expressions. The agent expression $E'' = E[v/E']$ obtained by substituting E' for v in E is defined by induction on the structure of expressions as follows:

- If $E = p$ for some $p \in \mathcal{Q}.D$, then $E'' = p$.
- If $E = w$ for some $w \in V$, $w \neq v$ then $E'' = w$.
- If $E = v$ then $E'' = E'$.
- If $E = \mathit{proj}(B)(E_1)$ for some expression E_1 then $E'' = \mathit{proj}(B)(E_1[v/E'])$.
- If $E = \mathit{rename}(r)(E_1)$ for some expression E_1 then $E'' = \mathit{rename}(r)(E_1[v/E'])$.
- If $E = E_1 \parallel E_2$ for some expressions E_1 and E_2 then $E'' = E_1[v/E'] \parallel E_2[v/E']$.

Expression substitution differs from expression evaluation in that substitution is a syntactic operation that returns a new expression, while evaluation is a semantic operation that returns a value. The two are related by the following result.

Lemma 2.50 (Substitution Lemma). Let $E_1[v]$ and $E_2[v]$ be two expressions in the variable v. Then for all agents p

$$E_1[v/E_2[v]][p] = E_1[E_2[p]]$$

Proof: The result follows by induction on the structure of expressions. □

Expressions are defined over an agent algebra $\mathcal{Q}$ because agents from $\mathcal{Q}$ appear as constants in the expression. Sometimes it is necessary to translate one expression from one agent algebra $\mathcal{Q}$ to another agent algebra $\mathcal{Q}'$. Expressions can be so translated if there exists a function that maps each agent of $\mathcal{Q}$ to an agent of $\mathcal{Q}'$.

[1]While it is possible to define the simultaneous substitution of several expression for several variables, we limit the exposition to the single variable case to keep the notation simpler.

Definition 2.51 (Expression Translation). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be agent algebras and let E be a closed expression over $\mathcal{Q}$. Let H be a function from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$. The expression $E' = E[p/H(p)]$ is the expression over $\mathcal{Q}'$ formed from E by replacing every instance of each agent p in E with $H(p)$. Formally E' is defined by induction on the structure of expressions as follows:

$$\begin{aligned}
p[p/H(p)] &= H(p) \\
\mathit{proj}(B)(E_1)[p/H(p)] &= \mathit{proj}(B)(E_1[p/H(p)]) \\
\mathit{rename}(r)(E_1)[p/H(p)] &= \mathit{rename}(r)(E_1[p/H(p)]) \\
E_1 \parallel E_2[p/H(p)] &= E_1[p/H(p)] \parallel E_2[p/H(p)].
\end{aligned}$$

Note that in the notation above the symbol p does not represent a particular agent in the algebra, but is implicitly universally quantified as if it were a bound variable. Note also that expression translation differs from expression substitution (definition 2.49) in that expression translation replaces constants (agents) in one agent algebra with constants in another agent algebra, while leaving the structure of the expression unchanged. Therefore, while E is an expression over $\mathcal{Q}$, E' is an expression over $\mathcal{Q}'$. Notice also that we are only considering closed expressions.

2.6 Relationships between Agent Algebras

As discussed in the introduction, agent algebras do not exist in isolation. It is often convenient to use different models for different parts of the design. It is often *necessary* to use different models for different phases of the design. Conservative approximations represent the process of abstracting a specification in a less detailed semantic domain. Inverses of conservative approximations represent the opposite process of refinement.

As an example, consider an agent algebra $\mathcal{Q}$ that distinguishes between two events, a and b. The algebra has four agents that encode all the possible combinations of the occurrence of a and b as follows: ab says that both a and b have occurred, $a\overline{b}$ says that a has occurred and b has not occurred, occurred, $\overline{a}b$ says that b has occurred and a has not occurred, and $\overline{a}\overline{b}$ says that neither a nor b have occurred. We order the agents so that ab sits at the top, $\overline{a}\overline{b}$ at the bottom, and $a\overline{b}$ and $\overline{a}b$ are in the middle and mutually incomparable.

A more abstract agent algebra $\mathcal{Q}'$ does not distinguish between a and b, and can only represent the occurrence of one event c. Here we order c and $\overline{c}$ so that $\overline{c} \preceq c$. We build an abstraction

Ψ_u (the notation employed here will become clear later on) from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ as follows:

$$\begin{aligned} \Psi_u(ab) = \Psi_u(a\overline{b}) = \Psi_u(\overline{a}b) &= c, \\ \Psi_u(\overline{a}\overline{b}) &= \overline{c}. \end{aligned}$$

In other words, the abstraction is telling us that event c in the abstract domain represents the occurrence of either a, or b or both, and that $\overline{c}$ the absence of both a and b. The interpretation of c through the abstraction Ψ_u, denoted by c_u and $\overline{c_u}$, is therefore the following:

$$\begin{aligned} c_u &\Rightarrow a \vee b \vee ab = a \vee b, \\ \overline{c_u} &\Rightarrow \overline{a}\overline{b}. \end{aligned}$$

We wish to now construct an inverse, refinement map Ψ_{inv}. Given the above meaning of c and $\overline{c}$, we should clearly assign

$$\Psi_{inv}(\overline{c}) = \overline{a}\overline{b},$$

However, we are given a choice as to what to assign to $\Psi_{inv}(c)$. In fact, c determines an equivalence class in $\mathcal{Q}$, that is the set of the agents p such that $\Psi_u(p) = c$. In other words, if we are given $\Psi_u(p) = c$, we are unable to identify p uniquely.

Note however that Ψ_u is not the only possible abstraction. The function Ψ_u uses the abstract event as an upper bound, by choosing c to represent the possibility that a concrete event *has* occurred. Likewise, we may construct a lower bound Ψ_l, that is an abstraction that takes the abstract event to represent the possibility that a concrete event *has not* occurred. The definition of Ψ_l is as follows:

$$\begin{aligned} \Psi_l(ab) &= c \\ \Psi_l(a\overline{b}) = \Psi_l(\overline{a}b) = \Psi_l(\overline{a}\overline{b}) &= \overline{c} \end{aligned}$$

The interpretation of the abstract event is now different. In particular we have:

$$\begin{aligned} c_l &\Rightarrow ab \\ \overline{c_l} &\Rightarrow \overline{a} \vee \overline{b} \vee \overline{a}\overline{b} = \overline{a} \vee \overline{b}. \end{aligned}$$

Hence, for Ψ_l the inverse is uniquely determined only for c, and is such that

$$\Psi_{inv}(c) = ab.$$

If, for an agent p, $\Psi_l(p) = \Psi_u(p)$, then there is no ambiguity: either $\Psi_l(p) = \Psi_u(p) = c$, and therefore $p = ab$ (or, in other words, $\Psi_{inv}(c) = ab$), or $\Psi_l(p) = \Psi_u(p) = \overline{c}$, and therefore $p = \overline{a}\overline{b}$ (i.e., $\Psi_{inv}(\overline{c}) = \overline{a}\overline{b}$). The inverse, or refinement function, is therefore completely determined by the pair of functions Ψ_u and Ψ_l.

In the case that $\Psi_l(p) \neq \Psi_u(p)$ we however still have a choice. If $\Psi_u(p) = c$ and $\Psi_l(p) = \overline{c}$, then the interpretations of the abstractions give us

$$c_u \wedge \overline{c_l} = (a \vee b) \wedge (\overline{a} \vee \overline{b}),$$

i.e., either a or b occurs, but not both. This is consistent with both $a\overline{b}$ and with $\overline{a}b$, and hence we cannot determine p uniquely. This is to be expected, since we must have some loss of information by mapping a concrete into an abstract model.

Note the sense in which Ψ_u is an upper bound and Ψ_l is a lower bound. For all agents p, we have

$$p \preceq \Psi_{inv}(\Psi_u(p)),$$
$$\Psi_{inv}(\Psi_l(p)) \preceq p.$$

When $\Psi_u(p) = \Psi_l(p)$ then p is bounded from above and below by the same quantity, and is therefore determined exactly.

The above properties could be taken as the definition of a conservative approximation. However, unlike the simple example presented above, the inverse of an abstraction function is not necessarily always defined. This occurs, for example, when each of the models that we are relating are able to express information that is ignored by the other model. In that case, it is impossible to define the upper and the lower bound of a conservative approximation in terms of its inverse, since the inverse is not defined everywhere. We will therefore follow a different path. We first introduce conservative approximations by stating a preservation condition relative to the refinement order, without reference to the inverse function. We then later prove that when an inverse is defined, the above properties hold. Conversely, we show that if the properties hold and the abstractions are also monotonic, then we have a conservative approximation.

2.6.1 Conservative Approximations

A conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ is an ordered pair $\Psi = (\Psi_l, \Psi_u)$, where Ψ_l and Ψ_u are functions from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$. The first mapping is an upper bound of the agent relative to the order of the algebra: for instance, the abstract agent represents all of the possible behaviors of the agent in the more detailed domain, plus possibly some more. The second is a lower bound: the abstract agent represents only possible behaviors of the more detailed one, but possibly not all.

We define conservative approximations as abstractions that maintain a precise relationship between the orders in the two agent algebras.

Definition 2.52 (Conservative Approximation). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be ordered agent algebras, and let Ψ_l and Ψ_u be functions from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$. We say that $\Psi = (\Psi_l, \Psi_u)$ is a *conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$* if and only if for all agents p and q in $\mathcal{Q}.D$,

$$\Psi_u(p) \preceq \Psi_l(q) \Rightarrow p \preceq q.$$

Thus, when used in combination, the two mappings allow us to relate refinement verification results in the abstract domain to results in the more detailed domain. Hence, the verification can be done in $\mathcal{Q}'$, where it is presumably more efficient than in $\mathcal{Q}$. The conservative approximation guarantees that this will not lead to a false positive result, although false negatives are possible depending on how the approximation is chosen.

Usually a conservative approximation $\Psi = (\Psi_l, \Psi_u)$ has the additional property that $\Psi_l(p) \preceq \Psi_u(p)$ for all p, but this is not required. Also, having Ψ_l and Ψ_u be monotonic (relative to the ordering on agents) is common, but not required.

Example 2.53. Recall example 2.8, which described an agent algebra $\mathcal{Q}$ where each agent was simply a non-negative real number (representing, for example, maximum power dissipation). We extend $\mathcal{Q}$ to be an ordered agent algebra by defining $p \preceq p'$ if and only if p is less than or equal to p'. Let $\mathcal{Q}'$ be the analogous ordered agent algebra where each agent is a non-negative integer, rather than a real number. Then, $\Psi = (\Psi_l, \Psi_u)$ is a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$, where

$$\begin{aligned}\Psi_l(p) &= \lfloor p \rfloor \\ \Psi_u(p) &= \lceil p \rceil\end{aligned}$$

(i.e., the floor and the ceiling, respectively, of the real number p).

Example 2.53 above is typical: neither the floor function, nor the ceiling function, when used alone, would satisfy the requirements of a conservative approximation.

2.6.2 Inverses of Conservative Approximations

Let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. Let $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}'.D$ be such that $p' = \Psi_u(p)$. As we have discussed, p' represents a kind of upper bound on p. It is natural to ask whether there is an agent in $\mathcal{Q}.D$ that is represented exactly by p' rather than just being bounded by p'. If no agent in $\mathcal{Q}.D$ can be represented exactly, then Ψ is abstracting away too much information to be of much use. If every agent in $\mathcal{Q}.D$ can be represented exactly, then Ψ_l and Ψ_u are equal and are isomorphisms from $\mathcal{Q}$ to $\mathcal{Q}'$. These extreme cases illustrate that the amount of abstraction in Ψ is related to what agents p are represented exactly by $\Psi_u(p)$ and $\Psi_l(p)$.

To formalize what it means to be represented exactly in this context, we define the inverse of the conservative approximation Ψ. Normal notions of the inverse of a function are not adequate for this purpose, since Ψ is a pair of functions. We handle this by only considering those agents $p \in \mathcal{Q}.D$ for which $\Psi_l(p)$ and $\Psi_u(p)$ have the same value, call it p'. Intuitively, p' represents p exactly in this case; the key property of the inverse of Ψ (written Ψ_{inv}) is that $\Psi_{inv}(p') = p$. If $\Psi_l(p) \neq \Psi_u(p)$, then p is not represented exactly in $\mathcal{Q}'$. In this case, p is not in the image of Ψ_{inv}. Characterizing when $\Psi_{inv}(p')$ is defined (and what its value is) helps to show what agents in $\mathcal{Q}.D$ can be represented exactly (not just conservatively) by agents in $\mathcal{Q}'.D$.

Before formalizing the idea of the inverse of a conservative approximation, we prove a lemma needed to show that it is uniquely defined. The result applies only if the algebras are partially ordered (i.e., the order is antisymmetric). Once the inverse of a conservative approximation is defined, we show that it is one-to-one and that it is monotonic.

Lemma 2.54. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. For every $p' \in \mathcal{Q}'.D$, there is at most one $p \in \mathcal{Q}.D$ such that $\Psi_l(p) = p'$ and $\Psi_u(p) = p'$.

Proof: The proof is by contradiction. Assume there exist two distinct agents p_1 and p_2 in $\mathcal{Q}.D$ such that $\Psi_l(p_1)$, $\Psi_u(p_1)$, $\Psi_l(p_2)$ and $\Psi_u(p_2)$ are all equal to p'. This implies $\Psi_u(p_1) \preceq \Psi_l(p_2)$ and $\Psi_u(p_2) \preceq \Psi_l(p_1)$. Thus, by the definition of a conservative approximation (def. 2.52), $p_1 \preceq p_2$ and $p_2 \preceq p_1$. Therefore, $p_1 = p_2$, which is a contradiction. □

Definition 2.55 (Inverse of Conservative Approximation). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. Let $\mathcal{Q}_1.D$ be the set of $p \in \mathcal{Q}.D$ such that $\Psi_l(p) = \Psi_u(p)$. Let $\mathcal{Q}'_1.D$ be the image of $\mathcal{Q}_1.D$ under Ψ_l. The *inverse of* Ψ is the partial function Ψ_{inv} with domain $\mathcal{Q}'.D$ and codomain $\mathcal{Q}.D$

that is defined for all $p' \in \mathcal{Q}'_1.D$ so that $\Psi_{inv}(p') = p$, where p is the unique (by lemma 2.54 and the definition of $\mathcal{Q}'_1.D$) agent such that $\Psi_l(p) = p'$ and $\Psi_u(p) = p'$.

Corollary 2.56. Ψ_{inv} is one-to-one. Furthermore, when restricted to the image of Ψ_{inv}, the functions Ψ_l and Ψ_u are equal and are the inverse of Ψ_{inv}.

Proof: For an arbitrary p' in $\mathcal{Q}'_1.D$, let $p = \Psi_{inv}(p')$. By the definition of Ψ_{inv}, $p' = \Psi_l(p)$ and $p' = \Psi_l(p)$. Thus, when restricted to the image of Ψ_{inv}, the functions Ψ_l and Ψ_u are equal and are the inverse of Ψ_{inv}. Therefore, since an inverse of Ψ_{inv} exists, Ψ_{inv} is one-to-one. □

We now show that if $\Psi = (\Psi_l, \Psi_u)$ is a conservative approximation, then Ψ_l and Ψ_u are indeed lower and upper bounds.

Theorem 2.57. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. Let $p \in \mathcal{Q}.D$ be an agent such that both $\Psi_{inv}(\Psi_l(p))$ and $\Psi_{inv}(\Psi_u(p))$ are defined. Then

$$\Psi_{inv}(\Psi_l(p)) \preceq p \preceq \Psi_{inv}(\Psi_u(p)).$$

Proof: Clearly, by definition 2.55,

$$\begin{aligned}
\Psi_l(\Psi_{inv}(\Psi_u(p))) &= \Psi_u(p) \\
\Psi_u(\Psi_{inv}(\Psi_l(p))) &= \Psi_l(p).
\end{aligned}$$

Therefore,

$$\begin{aligned}
\Psi_u(p) &\preceq \Psi_l(\Psi_{inv}(\Psi_u(p))) \\
\Psi_u(\Psi_{inv}(\Psi_l(p))) &\preceq \Psi_l(p).
\end{aligned}$$

Hence, since Ψ is a conservative approximation, by definition 2.52,

$$\begin{aligned}
p &\preceq \Psi_{inv}(\Psi_u(p)) \\
\Psi_{inv}(\Psi_l(p)) &\preceq p.
\end{aligned}$$

□

As expected, the inverse of a conservative approximation, when defined, is monotonic. This is true whether or not the upper and lower bound of the abstraction are monotonic. If they are, however, then the inverse of a conservative approximation preserve the ordering of agents in both directions. The following theorem proves these facts.

Theorem 2.58. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. Assume p'_1 and p'_2 are agents in $\mathcal{Q}'.D$ such that $\Psi_{inv}(p'_1)$ and $\Psi_{inv}(p'_2)$ are both defined. Then

- if $p'_1 \preceq p'_2$, then $\Psi_{inv}(p'_1) \preceq \Psi_{inv}(p'_2)$.
- if either Ψ_u or Ψ_l is monotonic, then $p'_1 \preceq p'_2$ if and only if $\Psi_{inv}(p'_1) \preceq \Psi_{inv}(p_2)$.

Proof: The first part of the proof is composed of the following series of implications.

$p'_1 \preceq p'_2$

by corollary 2.56

$\Rightarrow \quad \Psi_u(\Psi_{inv}(p'_1)) = p'_1 \preceq p'_2 = \Psi_l(\Psi_{inv}(p'_2))$

since Ψ is a conservative approximation, by definition 2.52

$\Rightarrow \quad \Psi_{inv}(p'_1) \preceq \Psi_{inv}(p'_2)$.

For the second part, assume Ψ_u is monotonic. Then,

$\Psi_{inv}(p'_1) \preceq \Psi_{inv}(p'_2)$

since Ψ_u is monotonic

$\Rightarrow \quad \Psi_u(\Psi_{inv}(p'_1)) \preceq \Psi_u(\Psi_{inv}(p'_2))$

by corollary 2.56, $\Psi_u(\Psi_{inv}(p'_1)) = p'_1$ and $\Psi_u(\Psi_{inv}(p'_2)) = p'_2$, therefore

$\Leftrightarrow \quad p'_1 \preceq p'_2$.

The proof is similar if Ψ_l is monotonic. □

Every agent $p' \in \mathcal{Q}'.D$ determines two equivalence classes in $\mathcal{Q}.D$: the class of the agents p such that $\Psi_u(p) = p'$, and the class of the agents p such that $\Psi_l(p) = p'$. The inverse is defined on p' if and only if it is the greatest element of the first class and the lowest element of the second class.

Theorem 2.59. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. Let $p' \in \mathcal{Q}'.D$ and $p \in \mathcal{Q}.D$ be agents. Then $\Psi_{inv}(p') = p$ if and only if

$$\begin{aligned} p &= \max\{\, p_1 : \Psi_u(p_1) = p' \}, \\ p &= \min\{\, p_1 : \Psi_l(p_1) = p' \}. \end{aligned}$$

Proof: For the forward direction, assume $\Psi_{inv}(p') = p$. Let p_1 be such that $\Psi_u(p_1) = p'$. Then, by definition 2.55, $\Psi_u(p_1) = \Psi_l(p)$. Therefore, since Ψ_{inv} is a conservative approximation, by definition 2.52, $p_1 \preceq p$. Therefore, $p = \max\{\, p_1 : \Psi_u(p_1) = p'\}$. Similarly, if p_1 is such that $\Psi_l(p_1) = p'$, then $\Psi_u(p) = \Psi_l(p_1)$, and therefore $p \preceq p_1$. Hence, $p = \min\{\, p_1 : \Psi_l(p_1) = p'\}$.

For the reverse direction, clearly, since $p = \max\{\, p_1 : \Psi_u(p_1) = p'\}$, $\Psi_u(p) = p'$. Similarly, $\Psi_l(p) = p'$. Therefore, $\Psi_{inv}(p')$ is defined and $\Psi_{inv}(p') = p$. □

2.6.3 Compositional Conservative Approximations

In this section we discuss compositionality issues for both the upper and lower bound of a conservative approximation, and for the inverse. At the end of this section we give sufficient conditions for the inverse of a conservative approximation to be an isomorphism between two partially ordered agent subalgebras. We will later use the properties of the inverse of a conservative approximation to "embed" one partially ordered agent algebra into another.

A refinement verification problem is often of the form $[\![\, E \,]\!] \preceq q$, where q is the specification and E is an expression over the agent algebra. Computing $\Psi_u([\![\, E \,]\!])$ involves evaluating the expression E in the concrete domain, a potentially expensive operation. A compositional conservative approximation allows us to avoid this computation by translating the expression into the abstract domain.

As an example, consider the verification problem

$$\mathit{proj}(A)(p_1 \parallel p_2) \preceq p,$$

where p_1, p_2 and p are agents in $\mathcal{Q}.D$. This corresponds to checking whether an implementation consisting of two components p_1 and p_2 (along with some internal signals that are removed by the projection operation) satisfies the specification p. We say that a conservative approximation Ψ is a *compositional* conservative approximation if showing

$$\mathit{proj}(A)(\Psi_u(p_1) \parallel \Psi_u(p_2)) \preceq \Psi_l(p)$$

is sufficient to show that the original implementation satisfies its specification. The following definition makes this notion precise.

Definition 2.60 (Compositional Conservative Approx.). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be preordered agent algebras, and let Ψ_l and Ψ_u be functions from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$. We say $\Psi = (\Psi_l, \Psi_u)$ is a *compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$* if and only if for all closed expressions E

over $\mathcal{Q}$, and for all agents $p_1 \in \mathcal{Q}.D$,

$$[\![\, E[p/\Psi_u(p)] \,]\!] \preceq \Psi_l(p_1) \Rightarrow [\![\, E \,]\!] \preceq p_1.$$

Obviously, a compositional conservative approximation is also a conservative approximation. Note that the implication in the definition must be interpreted in the following sense: if $[\![\, E[p/\Psi_u(p)] \,]\!]$ is defined and $[\![\, E[p/\Psi_u(p)] \,]\!] \preceq \Psi_l(p_1)$, *then* $[\![\, E \,]\!]$ is defined and $[\![\, E \,]\!] \preceq p_1$. This form of the implication is a consequence of interpreting undefinedness as the $\top$ element of an extended set, as explained in section 2.4.

The remainder of this section proves theorems that provide sufficient conditions for showing that some Ψ is a compositional conservative approximation. First we show that if $\Psi' = (\Psi'_l, \Psi'_u)$ provides looser lower and upper bounds than a compositional conservative approximation Ψ (i.e., $\Psi'_l(p) \preceq \Psi_l(p)$ and $\Psi_u(p) \preceq \Psi'_u(p)$ for all p), then Ψ' is also a compositional conservative approximation. Also, the functional composition of two compositional conservative approximations yields another compositional conservative approximation. Although the theorems are stated in terms of compositional conservative approximations, they apply to conservative approximations, as well.

Theorem 2.61. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be preordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. If $\Psi' = (\Psi'_l, \Psi'_u)$ is such that $\Psi'_l(p) \preceq \Psi_l(p)$ and $\Psi_u(p) \preceq \Psi'_u(p)$ for all $p \in \mathcal{Q}.D$, then Ψ' is a compositional conservative approximation.

Proof: Let E be a closed expression over $\mathcal{Q}$. We first show that if $\Psi_u(p) \preceq \Psi'_u(p)$ for all agents $p \in \mathcal{Q}.D$, then if $[\![\, E[p/\Psi'_u(p)] \,]\!]$ is defined, then $[\![\, E[p/\Psi_u(p)] \,]\!]$ is defined and $[\![\, E[p/\Psi_u(p)] \,]\!] \preceq [\![\, E[p/\Psi'_u(p)] \,]\!]$. The proof is by induction on the structure of expressions. Here we prove the base case, and the case for projection. The other cases are similar.

- Let $E = p$ for some agent $p \in \mathcal{Q}.D$. Clearly, by definition 2.51 and by hypothesis,

$$[\![\, E[p/\Psi_u(p)] \,]\!] = \Psi_u(p) \preceq \Psi'_u(p) = [\![\, E[p/\Psi'_u(p)] \,]\!].$$

- Let $E = \textit{proj}(B)(E_1)$ be an expression and assume that if $[\![\, E_1[p/\Psi'_u(p)] \,]\!]$ is defined, then $[\![\, E_1[p/\Psi_u(p)] \,]\!]$ is defined and $[\![\, E_1[p/\Psi_u(p)] \,]\!] \preceq [\![\, E_1[p/\Psi'_u(p)] \,]\!]$. Assume also that $[\![E[p/\Psi'_u(p)]]\!]$ is defined. Then

$[\![E[p/\Psi'_u(p)]]\!]\downarrow$

By definition 2.51,

$\Rightarrow \quad [\![E[p/\Psi'_u(p)]]\!] = [\![\textit{proj}(B)(E_1[p/\Psi'_u(p)])]\!]$

By definition 2.46,

$\Rightarrow \quad [\![E[p/\Psi'_u(p)]]\!] = \textit{proj}(B)([\![E_1[p/\Psi'_u(p)]]\!])$

Since *proj* is $\top$-monotonic,

and since by induction hypothesis $[\![E_1[p/\Psi_u(p)]]\!] \preceq [\![E_1[p/\Psi'_u(p)]]\!]$,

$\textit{proj}(B)([\![E_1[p/\Psi_u(p)]]\!])$ is defined and

$\Rightarrow \quad \textit{proj}(B)([\![E_1[p/\Psi_u(p)]]\!]) \preceq \textit{proj}(B)([\![E_1[p/\Psi'_u(p)]]\!]) = [\![E[p/\Psi'_u(p)]]\!]$

By definition 2.46 and by definition 2.51,

$\Rightarrow \quad [\![E[p/\Psi_u(p)]]\!] = \textit{proj}(B)([\![E_1[p/\Psi_u(p)]]\!]) \preceq [\![E[p/\Psi'_u(p)]]\!]$.

Therefore, by induction, if $[\![E[p/\Psi'_u(p)]]\!]$ is defined, then $[\![E[p/\Psi_u(p)]]\!]$ is defined and $[\![E[p/\Psi_u(p)]]\!] \preceq [\![E[p/\Psi'_u(p)]]\!]$.

To show that Ψ' is a compositional conservative approximation, let p_1 be an agent in $\mathcal{Q}.D$, and assume $[\![E[p/\Psi'_u(p)]]\!]$ is defined and $[\![E[p/\Psi'_u(p)]]\!] \preceq \Psi'_l(p_1)$. We must show that $[\![E]\!] \preceq p_1$. The proof is composed of the following series of implications.

$[\![E[p/\Psi'_u(p)]]\!] \preceq \Psi'_l(p_1)$

by our earlier result, $[\![E[p/\Psi_u(p)]]\!]$ is also defined and

$\Rightarrow \quad [\![E[p/\Psi_u(p)]]\!] \preceq [\![E[p/\Psi'_u(p)]]\!]$

by transitivity, since $[\![E[p/\Psi'_u(p)]]\!] \preceq \Psi'_l(p_1)$,

$\Rightarrow \quad [\![E[p/\Psi_u(p)]]\!] \preceq \Psi'_l(p_1)$

by transitivity, since $\Psi'_l(p_1) \preceq \Psi_l(p_1)$

$\Rightarrow \quad [\![E[p/\Psi_u(p)]]\!] \preceq \Psi_l(p_1)$

since Ψ is a compositional conservative approximation, by definition 2.60,

$\Rightarrow \quad [\![E]\!] \preceq p_1$.

Therefore, by definition 2.60, $\Psi' = (\Psi'_l, \Psi'_u)$ is a compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. $\square$

Theorem 2.62. Let $\mathcal{Q}$, $\mathcal{Q}'$ and $\mathcal{Q}''$ be preordered agent algebras and let $\Psi = (\Psi_l, \Psi_u)$ and $\Psi' = (\Psi'_l, \Psi'_u)$ be compositional conservative approximations from $\mathcal{Q}$ to $\mathcal{Q}$ and from $\mathcal{Q}'$ to $\mathcal{Q}''$, respectively. Then $\Psi'' = (\Psi''_l, \Psi''_u)$ is a compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}''$, where

$$\begin{aligned} \Psi''_l(p) &= \Psi'_l(\Psi_l(p)) \\ \Psi''_u(p) &= \Psi'_u(\Psi_u(p)). \end{aligned}$$

Proof: Let E be a closed expression over $\mathcal{Q}$. To show that Ψ' is a compositional conservative approximation, let p_1 be an agent in $\mathcal{Q}.D$, and assume $[\![\, E[p/\Psi''_u(p)] \,]\!] \preceq \Psi''_l(p_1)$. We must show that $[\![\, E \,]\!] \preceq p_1$. The proof is composed of the following series of implications.

$[\![\, E[p/\Psi''_u(p)] \,]\!] \preceq \Psi''_l(p_1)$

by definition 2.51,

$\Rightarrow \quad [\![\, E[p/\Psi_u(p)][p/\Psi'_u(p)] \,]\!] \preceq \Psi''_l(p_1)$

by hypothesis, since $\Psi''_l(p) = \Psi'_l(\Psi_l(p))$

$\Rightarrow \quad [\![\, E[p/\Psi_u(p)][p/\Psi'_u(p)] \,]\!] \preceq \Psi'_l(\Psi_l(p_1))$

since Ψ' is a compositional conservative approximation, by definition 2.60,

$\Rightarrow \quad [\![\, E[p/\Psi_u(p)] \,]\!] \preceq \Psi_l(p_1)$

since Ψ is a compositional conservative approximation, by definition 2.60,

$\Rightarrow \quad [\![\, E \,]\!] \preceq p_1.$

Therefore, by definition 2.60, $\Psi'' = (\Psi''_l, \Psi''_u)$ is a compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}''$. □

The next result gives sufficient conditions for a conservative approximation to be also compositional. The conditions are restrictions on the upper bound of the conservative approximation. The theorem can be understood by recalling the example verification problem described above, and by considering the following chain of implications:

$$\begin{aligned}
&\mathit{proj}(A)(\Psi_u(p_1) \parallel \Psi_u(p_2)) \preceq \Psi_l(p) \\
&\quad \text{assuming } \Psi_u(p_1 \parallel p_2) \preceq \Psi_u(p_1) \parallel \Psi_u(p_2) \\
\Rightarrow\quad &\mathit{proj}(A)(\Psi_u(p_1 \parallel p_2)) \preceq \Psi_l(p) \\
&\quad \text{assuming } \Psi_u(\mathit{proj}(A)(p')) \preceq \mathit{proj}(A)(\Psi_u(p')) \\
\Rightarrow\quad &\Psi_u(\mathit{proj}(A)(p_1 \parallel p_2)) \preceq \Psi_l(p) \\
&\quad \text{assuming } \Psi_u(p') \preceq \Psi_l(p) \text{ implies } p' \preceq p \\
\Rightarrow\quad &\mathit{proj}(A)(p_1 \parallel p_2) \preceq p.
\end{aligned}$$

The theorem formalizes the above assumptions (along with an assumption for the renaming operation) and proves that they are sufficient to show that Ψ is a compositional conservative approximation.

Theorem 2.63. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be preordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. If the following propositions S1 through S3 are satisfied for all agents p, p_1 and p_2 in $\mathcal{Q}.D$, then Ψ is a compositional conservative approximation.

S1. If $\Psi_u(p_1) \parallel \Psi_u(p_2)$ is defined, then $\Psi_u(p_1 \parallel p_2) \preceq \Psi_u(p_1) \parallel \Psi_u(p_2)$.

S2. If $\mathit{proj}(B)(\Psi_u(p))$ is defined, then $\Psi_u(\mathit{proj}(B)(p)) \preceq \mathit{proj}(B)(\Psi_u(p))$.

S3. If $\mathit{rename}(r)(\Psi_u(p))$ is defined, then $\Psi_u(\mathit{rename}(r)(p)) \preceq \mathit{rename}(r)(\Psi_u(p))$.

Proof: Let E be a closed expression over $\mathcal{Q}$. We first show that if $[\![E[p/\Psi_u(p)]]\!]$ is defined, then $[\![E]\!]$ is defined and $\Psi_u([\![E]\!]) \preceq [\![E[p/\Psi_u(p)]]\!]$. The proof is by induction on the structure of expressions. Here we prove the base case, and the case for projection. The other cases are similar.

- Let $E = p$ for some agent $p \in \mathcal{Q}.D$. Clearly

  $$\Psi_u([\![E]\!]) = \Psi_u(p)$$

  and, by definition 2.51,

  $$[\![E[p/\Psi_u(p)]]\!] = \Psi_u(p).$$

  Therefore, since by reflexivity $\Psi_u(p) \preceq \Psi_u(p)$, $\Psi_u([\![E]\!]) \preceq [\![E[p/\Psi_u(p)]]\!]$.

- Let $E = \textit{proj}(B)(E_1)$ and assume that if $[\![E_1[p/\Psi_u(p)]]\!]$ is defined, then $[\![E_1]\!]$ is defined and $\Psi_u([\![E_1]\!]) \preceq [\![E_1[p/\Psi_u(p)]]\!]$. Assume $[\![E[p/\Psi_u(p)]]\!]$ is defined. Then

  $[\![E[p/\Psi_u(p)]]\!]\downarrow$

  By definition 2.51,

  $\Rightarrow \quad [\![E[p/\Psi_u(p)]]\!] = [\![\textit{proj}(B)(E_1[p/\Psi_u(p)])]\!]$

  By definition 2.46,

  $\Rightarrow \quad [\![E[p/\Psi_u(p)]]\!] = \textit{proj}(B)([\![E_1[p/\Psi_u(p)]]\!])$

  Since *proj* is $\top$-monotonic,

  and since by hypothesis $\Psi_u([\![E_1]\!]) \preceq [\![E_1[p/\Psi_u(p)]]\!]$,

  $\textit{proj}(B)(\Psi_u([\![E_1]\!])$ is defined and

  $\Rightarrow \quad \textit{proj}(B)(\Psi_u([\![E_1]\!]) \preceq \textit{proj}(B)([\![E_1[p/\Psi_u(p)]]\!]) = [\![E[p/\Psi_u(p)]]\!]$

  By S2, $\Psi_u(\textit{proj}(B)([\![E_1]\!]))$ is defined and

  $\Rightarrow \quad \Psi_u(\textit{proj}(B)([\![E_1]\!])) \preceq \textit{proj}(B)(\Psi_u([\![E_1]\!]) \preceq [\![E[p/\Psi_u(p)]]\!]$

  By definition 2.46,

  $\Rightarrow \quad \Psi_u([\![E]\!]) = \Psi_u([\![\textit{proj}(B)(E_1)]\!] \preceq \textit{proj}(B)(\Psi_u([\![E_1]\!]) \preceq [\![E[p/\Psi_u(p)]]\!]$.

  Hence, since $\preceq$ is transitive, $\Psi_u([\![E]\!]) \preceq [\![E[p/\Psi_u(p)]]\!]$.

Therefore, by induction, if $[\![E[p/\Psi_u(p)]]\!]$ is defined, then $[\![E]\!]$ is defined and $\Psi_u([\![E]\!]) \preceq [\![E[p/\Psi_u(p)]]\!]$.

To show that Ψ is a compositional conservative approximation, let p_1 be an agent in $\mathcal{Q}.D$, and assume $[\![E[p/\Psi_u(p)]]\!] \preceq \Psi_l(p_1)$. We must show that $[\![E]\!] \preceq p_1$. By transitivity of the refinement ordering, since $\Psi_u([\![E]\!]) \preceq [\![E[p/\Psi_u(p)]]\!]$, $\Psi_u([\![E]\!]) \preceq \Psi_l(p_1)$. Therefore, since Ψ is a conservative approximation, by definition 2.52, $[\![E]\!] \preceq p_1$. Hence, by definition 2.60, $\Psi = (\Psi_l, \Psi_u)$ is a compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. $\square$

When the upper bound of a conservative approximation satisfies S1 through S3 then we can prove similar properties for the inverse of the conservative approximation.

Theorem 2.64. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ satisfying S1 through S3. Let p_1' and p_2' be agents in $\mathcal{Q}'.D$ such that $\Psi_{inv}(p_1')$ and $\Psi_{inv}(p_2')$ are both defined. Then

1. If $\Psi_{inv}(p'_1 \parallel p'_2)$ is defined, then $\Psi_{inv}(p'_1) \parallel \Psi_{inv}(p'_2) \preceq \Psi_{inv}(p'_1 \parallel p'_2)$.
2. If $\Psi_{inv}(\mathit{proj}(B)(p'_1))$ is defined, then $\mathit{proj}(B)(\Psi_{inv}(p'_1)) \preceq \Psi_{inv}(\mathit{proj}(B)(p'_1))$.
3. If $\Psi_{inv}(\mathit{rename}(r)(p'_1))$ is defined, then $\mathit{rename}(r)(\Psi_{inv}(p'_1)) \preceq \Psi_{inv}(\mathit{rename}(r)(p'_1))$.

Proof: We prove the parallel composition case. The other cases are similar.

Let p'_1 and p'_2 be agents in $\mathcal{Q}'.D$ such that $\Psi_{inv}(p'_1)$ and $\Psi_{inv}(p'_2)$ are defined, and assume that $\Psi_{inv}(p'_1 \parallel p'_2)$ is also defined.

Since $\Psi_{inv}(p'_1)$ and $\Psi_{inv}(p'_2)$ are both defined, then

$$p'_1 \parallel p'_2 = \Psi_u(\Psi_{inv}(p'_1)) \parallel \Psi_u(\Psi_{inv}(p'_2)).$$

Then, since Ψ_u satisfies S1, $\Psi_u(\Psi_{inv}(p'_1) \parallel \Psi_{inv}(p'_2))$ is defined and

$$\Psi_u(\Psi_{inv}(p'_1) \parallel \Psi_{inv}(p'_2)) \preceq \Psi_u(\Psi_{inv}(p'_1)) \parallel \Psi_u(\Psi_{inv}(p'_2)) = p'_1 \parallel p'_2.$$

Likewise,

$$p'_1 \parallel p'_2 = \Psi_l(\Psi_{inv}(p'_1 \parallel p'_2)).$$

Therefore,

$$\Psi_u(\Psi_{inv}(p'_1) \parallel \Psi_{inv}(p'_2)) \preceq \Psi_l(\Psi_{inv}(p'_1 \parallel p'_2)).$$

Hence, since Ψ is a conservative approximation,

$$\Psi_{inv}(p'_1) \parallel \Psi_{inv}(p'_2) \preceq \Psi_{inv}(p'_1 \parallel p'_2).$$

□

In general we are mostly interested in compositionality for the upper bound side, or implementation side, of the refinement inequality, as definition 2.60 shows. But we may consider compositionality rules for the lower bound, or specification side, as well. The rules for $\top$-monotonicity require that we state this result in a dual way, by considering a sort of conservative "counter-approximation".

Definition 2.65 (Spec-Compositional Conservative Approx.). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be preordered agent algebras, and let Ψ_l and Ψ_u be functions from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$. We say $\Psi = (\Psi_l, \Psi_u)$ is a *spec-compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$* if and only if for all closed expressions E over $\mathcal{Q}$, and for all agents $p \in \mathcal{Q}.D$,

$$p \not\preceq [\![E]\!] \Rightarrow \Psi_u(p) \not\preceq [\![E[p_1/\Psi_l(p_1)]]\!].$$

The implication must again be interpreted in the sense that if $[\![E]\!]$ is defined and $p \not\preceq [\![E]\!]$, then $[\![E[p_1/\Psi_l(p_1)]]\!]$ is defined and $\Psi_u(p) \not\preceq [\![E[p_1/\Psi_l(p_1)]]\!]$.

Results similar to theorem 2.63 and theorem 2.64 apply to compositionality for the lower bound. We prove them below.

Theorem 2.66. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be preordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. If the following propositions S4 through S6 are satisfied for all agents p, p_1 and p_2 in $\mathcal{Q}.D$, then Ψ is a spec-compositional conservative approximation.

S4. If $\Psi_l(p_1 \parallel p_2)$ is defined, then $\Psi_l(p_1) \parallel \Psi_l(p_2) \preceq \Psi_l(p_1 \parallel p_2)$.

S5. If $\Psi_l(\mathit{proj}(B)(p))$ is defined, $\mathit{proj}(B)(\Psi_l(p)) \preceq \Psi_l(\mathit{proj}(B)(p))$.

S6. If $\Psi_l(\mathit{rename}(r)(p))$ is defined, then $\mathit{rename}(r)(\Psi_l(p)) \preceq \Psi_l(\mathit{rename}(r)(p))$.

Proof: Let E be a closed expression over agent algebra $\mathcal{Q}$. We first show that if $[\![E]\!]$ is defined, then $[\![E[p/\Psi_l(p)]]\!]$ is defined, and $[\![E[p/\Psi_l(p)]]\!] \preceq \Psi_l([\![E]\!])$. The proof is by induction on the structure of expressions. Here we prove the base case, and the case for projection. The other cases are similar.

- Let $E = p$ for some agent $p \in \mathcal{Q}.D$. Clearly

$$\Psi_l([\![E]\!]) = \Psi_l(p)$$

and, by definition 2.51,

$$[\![E[p/\Psi_l(p)]]\!] = \Psi_l(p).$$

Therefore, since by reflexivity $\Psi_l(p) \preceq \Psi_l(p)$, $[\![E[p/\Psi_l(p)]]\!] \preceq \Psi_l([\![E]\!])$.

- Let $E = \mathit{proj}(B)(E_1)$ and assume that if $[\![E_1]\!]$ is defined, then $[\![E_1[p/\Psi_l(p)]]\!]$ is defined and $[\![E_1[p/\Psi_l(p)]]\!] \preceq \Psi_l([\![E_1]\!])$. Assume $[\![E]\!]$ is defined. Then,

$$[\![E]\!]\downarrow$$

by definition 2.51,

$$\Rightarrow \quad [\![E]\!] = [\![\mathit{proj}(B)(E_1)]\!]$$

by definition 2.46,

$$\Rightarrow \quad [\![E]\!] = \mathit{proj}(B)([\![E_1]\!])$$

therefore $\Psi_l(\mathit{proj}(B)(\llbracket E_1 \rrbracket))\downarrow$, and by S5

$\Rightarrow \quad \mathit{proj}(B)(\Psi_l(\llbracket E_1 \rrbracket)) \preceq \Psi_l(\mathit{proj}(B)(\llbracket E_1 \rrbracket))$

since, by induction hypothesis, $\llbracket E_1[p/\Psi_l(p)] \rrbracket \preceq \Psi_l(\llbracket E_1 \rrbracket)$

and since *proj* is $\top$-monotonic

$\Rightarrow \quad \mathit{proj}(B)(\llbracket E_1[p/\Psi_l(p)] \rrbracket) \preceq \mathit{proj}(B)(\Psi_l(\llbracket E_1 \rrbracket)) \preceq \Psi_l(\mathit{proj}(B)(\llbracket E_1 \rrbracket))$

by definition 2.46,

$\Rightarrow \quad \mathit{proj}(B)(\llbracket E_1[p/\Psi_l(p)] \rrbracket) \preceq \Psi_l(\mathit{proj}(B)(\llbracket E_1 \rrbracket)) = \Psi_l(\llbracket E \rrbracket)$

by definition 2.46 and definition 2.51,

$\Rightarrow \quad \llbracket E[p/\Psi_l(p)] \rrbracket = \mathit{proj}(B)(\llbracket E_1[p/\Psi_l(p)] \rrbracket) = \Psi_l(\llbracket E \rrbracket)$

Thus, by induction, if $\llbracket E \rrbracket$ is defined, then $\llbracket E[p/\Psi_l(p)] \rrbracket$ is defined, and $\llbracket E[p/\Psi_l(p)] \rrbracket \preceq \Psi_l(\llbracket E \rrbracket)$.

Since we know the expressions are defined, we will now show the contrapositive of definition 2.65, that is

$$\Psi_u(p) \preceq \llbracket E[p_1/\Psi_l(p_1)] \rrbracket \Rightarrow p \preceq \llbracket E \rrbracket.$$

Let p be an agent in $\mathcal{Q}.D$, and assume $\Psi_u(p) \preceq \llbracket E[p_1/\Psi_l(p_1)] \rrbracket$. By transitivity of the refinement relation, since $\llbracket E[p_1/\Psi_l(p_1)] \rrbracket \preceq \Psi_l(\llbracket E \rrbracket)$, $\Psi_u(p) \preceq \Psi_l(\llbracket E \rrbracket)$. Therefore, since Ψ is a conservative approximation, $p \preceq \llbracket E \rrbracket$. Hence, by definition 2.65, $\Psi = (\Psi_l, \Psi_u)$ is a spec-compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. $\square$

Theorem 2.67. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a spec-compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ satisfying S4 through S6. Let p_1' and p_2' be agents in $\mathcal{Q}'.D$ such that $\Psi_{inv}(p_1')$ and $\Psi_{inv}(p_2')$ are both defined. Then

1. If $\Psi_{inv}(p_1') \parallel \Psi_{inv}(p_2')$ is defined and $\Psi_{inv}(p_1' \parallel p_2')$ is defined, then
$$\Psi_{inv}(p_1' \parallel p_2') \preceq \Psi_{inv}(p_1') \parallel \Psi_{inv}(p_2').$$
2. If $\mathit{proj}(B)(\Psi_{inv}(p_1'))$ is defined, then if $\Psi_{inv}(\mathit{proj}(B)(p_1'))$ is defined then
$$\Psi_{inv}(\mathit{proj}(B)(p_1')) \preceq \mathit{proj}(B)(\Psi_{inv}(p_1')).$$
3. If $\mathit{rename}(r)(\Psi_{inv}(p_1'))$ is defined, then if $\Psi_{inv}(\mathit{rename}(r)(p_1'))$ is defined then
$$\Psi_{inv}(\mathit{rename}(r)(p_1')) \preceq \mathit{rename}(r)(\Psi_{inv}(p_1')).$$

Proof: We prove the projection case. The other cases are similar. Let p_1' be an agent in $\mathcal{Q}'.D$ such that $\Psi_{inv}(p_1')$ is defined, and assume that $\mathit{proj}(B)(\Psi_{inv}(p_1'))$ is also defined. Then, since Ψ_l is total, $\Psi_l(\mathit{proj}(B)(\Psi_{inv}(p_1')))$ is defined. Therefore, since Ψ satisfies S5, also $\mathit{proj}(B)(\Psi_l(\Psi_{inv}(p_1')))$ is defined and

$$\mathit{proj}(B)(\Psi_l(\Psi_{inv}(p_1'))) = \mathit{proj}(B)(p_1') \preceq \Psi_l(\mathit{proj}(B)(\Psi_{inv}(p_1'))).$$

Assume now that $\Psi_{inv}(\mathit{proj}(B)(p_1'))$ is defined. Then, since by corollary 2.56 Ψ_u is inverse of Ψ_{inv}, $\Psi_u(\Psi_{inv}(\mathit{proj}(B)(p_1'))) = \mathit{proj}(B)(p_1')$. Therefore

$$\Psi_u(\Psi_{inv}(\mathit{proj}(B)(p_1'))) \preceq \Psi_l(\mathit{proj}(B)(\Psi_{inv}(p_1'))).$$

Finally, since Ψ is a conservative approximation,

$$\Psi_{inv}(\mathit{proj}(B)(p_1')) \preceq \mathit{proj}(B)(\Psi_{inv}(p_1')).$$

□

We might be interested in applying compositionality to the implementation and to the specification side of the inequality at the same time. In this case we talk about a *fully* compositional conservative approximation. Considerations of $\top$-monotonicity require that we state a property that is stronger than those for compositional and spec-compositional conservative approximations when they are taken together.

Definition 2.68 (Fully Compositional Conservative Approx.). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be preordered agent algebras, and let Ψ_l and Ψ_u be functions from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$. We say $\Psi = (\Psi_l, \Psi_u)$ is a *fully compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$* if and only if for all closed expressions E_1 and E_2 over $\mathcal{Q}$, if $[\![E_1[p/\Psi_u(p)]]\!]$ is defined then $[\![E_1]\!]$ is defined and if $[\![E_2]\!]$ is defined, then $[\![E_2[p/\Psi_l(p)]]\!]$ is defined, and

$$[\![E_1[p/\Psi_u(p)]]\!] \preceq [\![E_2[p/\Psi_l(p)]]\!] \Rightarrow [\![E_1]\!] \preceq [\![E_2]\!].$$

Although a compositional and spec-compositional conservative approximation is not necessarily fully compositional, the combined properties S1 through S3 and S4 through S6 are sufficient to imply full compositionality.

Theorem 2.69. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be preordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ satisfying propositions S1 through S3 and S4 through S6. Then Ψ is a fully compositional conservative approximation.

Proof: Let E_1 and E_2 be expressions over $\mathcal{Q}$ and assume $[\![E_1[p/\Psi_u(p)]]\!]$ and $[\![E_2]\!]$ are both defined. The combined proofs of theorem 2.63 and theorem 2.66 give us that $[\![E_1]\!]$ and $[\![E_2[p/\Psi_l(p)]]\!]$ are both defined and

$$\begin{aligned}\Psi_u([\![E_1]\!]) &\preceq [\![E_1[p/\Psi_u(p)]]\!]\\ [\![E_2[p/\Psi_l(p)]]\!] &\preceq \Psi_l([\![E_2]\!]).\end{aligned}$$

Assume now that $[\![E_1[p/\Psi_u(p)]]\!] \preceq [\![E_2[p/\Psi_l(p)]]\!]$. Then, by transitivity, $\Psi_u([\![E_1]\!]) \preceq \Psi_l([\![E_2]\!])$. Therefore, since Ψ is a conservative approximation, $[\![E_1]\!] \preceq [\![E_2]\!]$. Hence, by definition 2.68, Ψ is a fully compositional conservative approximation. $\square$

Since the inverse Ψ_{inv} of a conservative approximation is one-to-one (cor. 2.56), it is natural to ask whether it is also an embedding. The next result shows that this is the case when the assumptions of theorem 2.63 and theorem 2.66 are combined. Before we prove this result, we first specialize the notion of homomorphism and of an embedding to the agent algebra case.

Definition 2.70 (Agent Algebra Homomorphism). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras. Let H be a function from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$. The function H is a *homomorphism from $\mathcal{Q}$ to $\mathcal{Q}'$* if and only if

$$\begin{aligned}H(p_1 \parallel p_2) &= H(p_1) \parallel H(p_2),\\ H(\mathit{rename}(r)(p)) &= \mathit{rename}(r)(H(p)),\\ H(\mathit{proj}(B)(p)) &= \mathit{proj}(B)(H(p)).\end{aligned}$$

Definition 2.71 (Agent Algebra Embedding). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let H be a homomorphism from $\mathcal{Q}$ to $\mathcal{Q}'$. Then H is an *embedding* from $\mathcal{Q}$ to $\mathcal{Q}'$ if and only if H is one-to-one.

Theorem 2.72. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a fully compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ satisfying propositions S1 through S3 and S4 through S6. Further assume that for all agents $p \in \mathcal{Q}.D$, $\Psi_l(p) \preceq \Psi_u(p)$. If Ψ_{inv} is defined anywhere, then it is an embedding from a subalgebra of $\mathcal{Q}'$ to $\mathcal{Q}$.

Proof: Let $\mathcal{Q}'_1.D$ be the set of agents for which Ψ_{inv} is defined. We must show that if $\mathcal{Q}'_1.D$ is non-empty, then it forms a subalgebra of $\mathcal{Q}'$, and Ψ_{inv} is an isomorphism from that subalgebra to a subalgebra of $\mathcal{Q}$. In particular, we must show that $\mathcal{Q}'_1.D$ is closed under the operations of projection, renaming and parallel composition. The following lemma proves the case for parallel composition. The other cases are similar.

Lemma 2.73. Let p_1' and p_2' be agents in $\mathcal{Q}_1'.D$. If $p_1' \parallel p_2'$ is defined, then $\Psi_{inv}(p_1' \parallel p_2')$ is defined (i.e., $\mathcal{Q}_1'.D$ is closed under parallel composition), $\Psi_{inv}(p_1') \parallel \Psi_{inv}(p_2')$ is defined and

$$\Psi_{inv}(p_1' \parallel p_2') = \Psi_{inv}(p_1') \parallel \Psi_{inv}(p_2').$$

Proof: Let p_1' and p_2' be agents in $\mathcal{Q}_1'.D$, and let $p_1 = \Psi_{inv}(p')$ and $p_2 = \Psi_{inv}(p_2')$. Clearly, by definition 2.55, $\Psi_u(p_1) = \Psi_l(p_1) = p_1'$ and $\Psi_u(p_2) = \Psi_l(p_2) = p_2'$.

Assume now that $p_1' \parallel p_2'$ is defined. We show that $\Psi_{inv}(p_1') \parallel \Psi_{inv}(p_2')$ is defined. In fact, since $\Psi_u(p_1) = p_1'$ and $\Psi_u(p_2) = p_2'$, $\Psi_u(p_1) \parallel \Psi_u(p_2)$ is also defined and $\Psi_u(p_1) \parallel \Psi_u(p_2) = p_1' \parallel p_2'$. Therefore, by S1, $\Psi_u(p_1 \parallel p_2)$ is defined, which implies $p_1 \parallel p_2 = \Psi_{inv}(p_1') \parallel \Psi_{inv}(p_2')$ is defined.

In addition, by S1, $\Psi_u(p_1 \parallel p_2) \preceq \Psi_u(p_1) \parallel \Psi_u(p_2)$. To show that $\Psi_{inv}(p_1' \parallel p_2')$ is defined, consider the following series of inequalities:

$$\begin{aligned}
\Psi_u(p_1 \parallel p_2) \quad &\preceq \quad \Psi_u(p_1) \parallel \Psi_u(p_2) = p_1' \parallel p_2' \\
&\quad \text{since } \Psi_l(p_1) = p_1' \text{ and } \Psi_l(p_2) = p_2' \\
&\preceq \quad \Psi_l(p_1) \parallel \Psi_l(p_2) \\
&\quad \text{by S4, } \Psi_l(p_1 \parallel p_2) \text{ is defined and} \\
&\preceq \quad \Psi_l(p_1 \parallel p_2) \\
&\quad \text{since, by hypothesis, } \Psi_l(p_1 \parallel p_2) \preceq \Psi_u(p_1 \parallel p_2) \\
&\preceq \quad \Psi_u(p_1 \parallel p_2)
\end{aligned}$$

Therefore all the quantities are equal, and, in particular,

$$\Psi_u(p_1 \parallel p_2) = \Psi_l(p_1 \parallel p_2) = p_1' \parallel p_2'.$$

Hence, by definition 2.55, $\Psi_{inv}(p_1' \parallel p_2')$ is defined, and

$$\Psi_{inv}(p_1' \parallel p_2') = p_1 \parallel p_2.$$

Finally, since $\Psi_{inv}(p_1') \parallel \Psi_{inv}(p_2') = p_1 \parallel p_2$,

$$\Psi_{inv}(p_1' \parallel p_2') = \Psi_{inv}(p_1') \parallel \Psi_{inv}(p_2').$$

□

□

2.7 Conservative Approximations Induced by Galois Connections

In subsection 1.8.5 we have argued that there exists a close relationship between conservative approximations and abstract interpretations that use Galois connections. In this section we explore this relationship in details.

We begin by defining Galois connections [23], and proving some basic results about them, including several necessary conditions for functions that form a Galois connection. These results are common knowledge in the literature, with the exception of theorem 2.91, which slightly extends the standard results to give sufficient conditions for a function to be the abstraction function of some Galois connection. Later we show how a pair of Galois connections can be used to form a conservative approximation.

Subsection 2.7.3 is devoted to abstract interpretations [23]. We define abstract interpretations and characterize them in terms of the abstraction function of a Galois connection. We then show how to use an abstract interpretation and an additional Galois connection to form a compositional conservative approximation. We conclude the section with a discussion of the similarities and significant differences between abstract interpretations and conservative approximations.

2.7.1 Preliminaries

This section can be skipped by readers familiar with Galois connections. The following definition of a Galois connection is adopted from one given by Cousot and Cousot [23], where a Galois connections relates two posets. In order to highlight the relationship with conservative approximations, we restrict the definition here to posets that are the domain of agent algebras.

Definition 2.74 (Galois Connection). Let D and D' be partially ordered sets of agents. A Galois connection $\langle \alpha, \gamma \rangle$ from D to D' consists of an abstraction map $\alpha : D \mapsto D'$ and a concretization map $\gamma : D' \mapsto D$ such that for all $p \in D$ and $p' \in D'$,

$$\alpha(p) \preceq p' \iff p \preceq \gamma(p').$$

If the functions α and γ form a Galois connection, then they are monotonic.

Theorem 2.75. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D'. Then the functions α and γ are monotonic.

Proof: We begin by showing that α is monotonic. Let p_1 and p_2 be agents in D. Then

$$\begin{aligned} p_1 \preceq p_2 &\Leftrightarrow p_1 \preceq p_2 \wedge \alpha(p_2) \preceq \alpha(p_2) \\ &\quad \text{by the definition of a Galois connection (def. 2.74)} \\ &\Leftrightarrow p_1 \preceq p_2 \wedge p_2 \preceq \gamma(\alpha(p_2)) \\ &\quad \text{since } \preceq \text{ is transitive} \\ &\Rightarrow p_1 \preceq \gamma(\alpha(p_2)) \\ &\quad \text{by the definition of a Galois connection (def. 2.74)} \\ &\Leftrightarrow \alpha(p_1) \preceq \alpha(p_2). \end{aligned}$$

The proof that γ is monotonic is analogous: if p'_1 and p'_2 are agents in D', then

$$\begin{aligned} p'_1 \preceq p'_2 &\Leftrightarrow p'_1 \preceq p'_2 \wedge \gamma(p'_1) \preceq \gamma(p'_1) \\ &\quad \text{by the definition of a Galois connection (def. 2.74)} \\ &\Leftrightarrow p'_1 \preceq p'_2 \wedge \alpha(\gamma(p'_1)) \preceq p'_1 \\ &\quad \text{since } \preceq \text{ is transitive} \\ &\Rightarrow \alpha(\gamma(p'_1)) \preceq p'_2 \\ &\quad \text{by the definition of a Galois connection (def. 2.74)} \\ &\Leftrightarrow \gamma(p'_1) \preceq \gamma(p'_2). \end{aligned}$$

□

While the abstraction and concretization maps of a Galois connection are not inverse of each other, a weaker relation can be established.

Theorem 2.76. Let D and D' be partially ordered sets of agents, and let $\langle\alpha, \gamma\rangle$ be a Galois connection from D to D'. For all $p \in D$ and $p' \in D'$,

$$\begin{aligned} p &\preceq \gamma(\alpha(p)) \\ \alpha(\gamma(p')) &\preceq p'. \end{aligned}$$

Proof: Let $p \in D$ be an agent. By reflexivity, $\alpha(p) \preceq \alpha(p)$. Since $\langle\alpha, \gamma\rangle$ is a Galois connection, by definition 2.74, $p \preceq \gamma(\alpha(p))$.

Similarly, if $p' \in D'$ is an agent, then $\gamma(p') \preceq \gamma(p')$. Therefore, since $\langle\alpha, \gamma\rangle$ is a Galois connection, by definition 2.74, $\alpha(\gamma(p')) \preceq p'$. □

Corollary 2.77. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D'. Assume D and D' have least elements, $\perp$ and $\perp'$ respectively. Then,

$$\alpha(\perp) = \perp'.$$

Proof: The proof consists of the following series of implications, which start from the result of theorem 2.76 applied to $\perp'$.

$\alpha(\gamma(\perp')) \preceq \perp'$

Since $\perp \preceq (\gamma(\perp'))$, and since, by theorem 2.75, α is monotonic,

$\Rightarrow \quad \alpha(\perp) \preceq \alpha(\gamma(\perp')) \preceq \perp'$

since $\perp' \preceq \alpha(\perp)$,

$\Rightarrow \quad \alpha(\perp) = \perp'$.

□

The following result shows that a looser abstraction map implies a looser concretization map, and vice versa.

Theorem 2.78. Let D and D' be partially ordered sets of agents, and let $\langle \alpha_1, \gamma_1 \rangle$ and $\langle \alpha_2, \gamma_2 \rangle$ be Galois connections from D to D'. Then the following two statement are equivalent.

1. For all agents $p \in \mathcal{Q}.D$, $\alpha_1(p) \preceq \alpha_2(p)$.

2. For all agents $p' \in \mathcal{Q}'.D$, $\gamma_2(p') \preceq \gamma_1(p')$.

Proof: For the forward direction, let $p' \in \mathcal{Q}'.D$ be an agent. Since $\langle \alpha_2, \gamma_2 \rangle$ is a Galois connection, by theorem 2.76, $\alpha_2(\gamma_2(p')) \preceq p'$. The result is then derived as follows.

$\alpha_2(\gamma_2(p')) \preceq p'$

since by hypothesis $\alpha_1(\gamma_2(p')) \preceq \alpha_2(\gamma_2(p'))$

$\Rightarrow \quad \alpha_1(\gamma_2(p')) \preceq p'$

since $\langle \alpha_1, \gamma_1 \rangle$ is a Galois connection

$\Leftrightarrow \quad \gamma_2(p') \preceq \gamma_1(p')$.

Similarly, for the reverse direction, let $p \in \mathcal{Q}.D$ be an agent. Since $\langle \alpha_2, \gamma_2 \rangle$ is a Galois connection, by theorem 2.76, $p \preceq \gamma_2(\alpha_2(p))$. Then,

$p \preceq \gamma_2(\alpha_2(p))$

since by hypothesis $\gamma_2(\alpha_2(p)) \preceq \gamma_1(\alpha_2(p))$

$\Rightarrow \quad p \preceq \gamma_1(\alpha_2(p))$

since $\langle \alpha_1, \gamma_1 \rangle$ is a Galois connection

$\Leftrightarrow \quad \alpha_1(p) \preceq \alpha_2(p).$

□

The composition of Galois connections is again a Galois connection.

Theorem 2.79. Let D, D' and D'' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D', and $\langle \alpha', \gamma' \rangle$ a Galois connection from D' to D''. Then the pair of functions $\langle \alpha' \circ \alpha, \gamma \circ \gamma' \rangle$ is a Galois connection from D to D'.

Proof: Let $p \in D$ and $p'' \in D''$ be agents. We show that $\alpha'(\alpha(p)) \preceq p''$ if and only if $p \preceq \gamma(\gamma'(p''))$. The result follows from the following series of double implications.

$\alpha'(\alpha(p)) \preceq p''$

since $\langle \alpha', \gamma' \rangle$ is a Galois connection, by definition 2.74

$\Leftrightarrow \quad \alpha(p) \preceq \gamma'(p'')$

since $\langle \alpha, \gamma \rangle$ is a Galois connection, by definition 2.74

$\Leftrightarrow \quad p \preceq \gamma(\gamma'(p'')).$

□

In the following we characterize Galois connections in terms of least upper bounds and greatest lower bounds of sets of agents. Here we specialize definitions and results on upper bounds and least upper bounds to the case of sets of agents. The specializations for lower bounds and greatest lower bounds are dual.

Definition 2.80 (Upper Bound). Let D be a partially ordered set of agents and let $D_0 \subseteq D$. An agent $p \in D$ is an *upper bound* of D_0 if for all agents $p_0 \in D_0, p_0 \preceq p$.

Definition 2.81 (Least Upper Bound). Let D be a partially ordered set of agents and let $D_0 \subseteq D$. An agent $p \in D$ is a *least upper bound* of D_0, written $p = \bigsqcup D_0$, if p is an upper bound of D_0 and for all upper bounds q of D, $p \preceq q$.

Lemma 2.82. Let D be a partially ordered set of agents and let $D_0 \subseteq D$. The least upper bound of D_0, if it exists, is unique.

Lemma 2.83. Let D be a partially ordered set of agents and let $D_0 \subseteq D$. If $\bigsqcup D_0$ exists, then for all agents $p \in D$, $\bigsqcup D_0 \preceq p$ if and only if p is an upper bound of D_0, i.e.,

$$\forall p \in D \left[\bigsqcup D_0 \preceq p \Leftrightarrow \forall p_0 \in D_0 \; [p_0 \preceq p] \right].$$

Corollary 2.84. If $\bigsqcup D_0$ exists, then for all agents $p_0 \in D_0$, $p_0 \preceq \bigsqcup D_0$.

Lemma 2.85. Let D be a partially ordered set of agents and let $D_0 \subseteq D$. If there exists $q_0 \in D_0$ such that for all $p_0 \in D_0$, $p_0 \preceq q_0$, then

$$q_0 = \bigsqcup D_0.$$

The next series of results derive necessary and sufficient conditions for a function α to be the abstraction map of a Galois connection. We first show how to characterize the concretization map γ in terms of the abstraction map α as the upper bound of a sort of inverse function on the powersets. This function is then used to show certain properties of the abstraction map, which will then be proved sufficient for its characterization as a Galois connection.

Definition 2.86. Let D and D' be partially ordered sets of agents, and let α be a function from D to D'. Define Γ_α to be the function from D' to 2^D such that for all agents $p' \in D'$,

$$\Gamma_\alpha(p') = \{ p \in D : \alpha(p) \preceq p' \}.$$

Theorem 2.87. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D'. For all $p' \in D'$,

$$\gamma(p') \in \Gamma_\alpha(p'). \tag{2.1}$$

$$\gamma(p') = \bigsqcup \Gamma_\alpha(p') \tag{2.2}$$

Proof: By definition 2.86 and definition 2.74,

$$\Gamma_\alpha(p') = \{ p \in D : p \preceq \gamma(p') \}. \tag{2.3}$$

By reflexivity, $\gamma(p') \preceq \gamma(p')$, therefore $\gamma(p') \in \Gamma_\alpha(p')$. In addition, by equation 2.3, if $p \in \Gamma_\alpha(p')$, then $p \preceq \gamma(p')$. Therefore, by lemma 2.85, $\gamma(p') = \bigsqcup \Gamma_\alpha(p')$. □

Corollary 2.88. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ and $\langle \alpha', \gamma' \rangle$ be Galois connections from D to D'. If $\alpha = \alpha'$ then $\gamma = \gamma'$.

In other words, if α is the abstraction map of a Galois connection, the corresponding concretization map γ is uniquely determined.

The following theorem strengthen the results of theorem 2.76 in case α maps certain agents to unique agents.

Theorem 2.89. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D'. Let $p \in D$ be an agent such that for all agents $p_1 \in D$, if $\alpha(p) = \alpha(p_1)$ then $p = p_1$. Then $\gamma(\alpha(p)) = p$.

Proof: To prove the theorem we show that $p = \bigsqcup \Gamma_\alpha(\alpha(p))$. The result then follows from theorem 2.87.

By definition 2.86,

$$\Gamma_\alpha(\alpha(p)) = \{\, p_1 \in D : \alpha(p_1) \preceq \alpha(p) \}.$$

Then, clearly, since $\alpha(p) \preceq \alpha(p)$, $p \in \Gamma_\alpha(\alpha(p))$. Let now $p_1 \in \Gamma_\alpha(\alpha(p))$ be such that $p \preceq p_1$. Since $p_1 \in \Gamma_\alpha(\alpha(p))$, $\alpha(p_1) \preceq \alpha(p)$. In addition, since, by theorem 2.75, α is monotonic, and since $p \preceq p_1$, $\alpha(p) \preceq \alpha(p_1)$. Thus, by antisymmetry, $\alpha(p) = \alpha(p_1)$. Therefore, by hypothesis, $p = p_1$. Consequently, since by theorem 2.87, $\bigsqcup \Gamma_\alpha(\alpha(p)) \in \Gamma_\alpha(\alpha(p))$, and since by definition 2.80, $p \preceq \bigsqcup \Gamma_\alpha(\alpha(p))$, by the above result, $p = \bigsqcup \Gamma_\alpha(\alpha(p))$. □

The following properties hold of the abstraction function of a Galois connection.

Theorem 2.90. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D'. If D_0 is a subset of D such that $\bigsqcup D_0$ is defined, then

$$\alpha\Big(\bigsqcup D_0\Big) = \bigsqcup \alpha(D_0),$$

where α is naturally extended to sets.

Proof: By theorem 2.75, α is monotonic. Thus, for all $p_0 \in D_0$, since $p_0 \preceq \bigsqcup D_0$, then $\alpha(p_0) \preceq \alpha(\bigsqcup D_0)$. Therefore $\alpha(\bigsqcup D_0)$ is an upper bound of $\alpha(D_0)$.

Consider now the following chain of implications that begins with the definition of $\bigsqcup D_0$ and lemma 2.83:

$$\forall p \in D \left[\left(\bigsqcup D_0 \right) \preceq p \Leftrightarrow \forall p_0 \in D_0 \; [p_0 \preceq p] \right]$$

by specialization of the universally quantified variable p to $\gamma(p')$

$$\Rightarrow \quad \forall p' \in D' \left[\left(\bigsqcup D_0 \right) \preceq \gamma(p') \Leftrightarrow \forall p_0 \in D_0 \; [p_0 \preceq \gamma(p')] \right]$$

by the definition of a Galois connection (def. 2.74)

$$\Leftrightarrow \quad \forall p' \in D' \left[\alpha\left(\bigsqcup D_0 \right) \preceq p' \Leftrightarrow \forall p_0 \in D_0 \; [\alpha(p_0) \preceq p'] \right].$$

Assume now p' is an upper bound of $\alpha(D_0)$. Then, for all $p_0 \in D_0$, $\alpha(p_0) \preceq p'$. Thus, by the above result, $\alpha\left(\bigsqcup D_0\right) \preceq p'$. Therefore, since $\alpha\left(\bigsqcup D_0\right)$ is an upper bound of $\alpha(D_0)$, it is also the least, i.e.,

$$\alpha\left(\bigsqcup D_0 \right) = \bigsqcup \alpha(D_0).$$

□

Theorem 2.91. Let D and D' be partially ordered sets of agents. A function α from D to D' is the abstraction map of some Galois connection if and only if for all p' in D',

1. α is monotonic,

2. $\Gamma_\alpha(p')$ contains a unique maximal element, which implies that $\bigsqcup \Gamma_\alpha(p')$ is defined and is an element of $\Gamma_\alpha(p')$, and

3. $\alpha\left(\bigsqcup \Gamma_\alpha(p') \right) = \bigsqcup \alpha(\Gamma_\alpha(p'))$, where α is naturally extended to sets.

Proof: The forward implication follows from theorem 2.75, theorem 2.87 and theorem 2.90. To prove the reverse implication, we show that if item 1 through item 3 (above) hold, then $\langle \alpha, \gamma \rangle$ satisfies definition 2.74, i.e.,

$$\forall p \in D, \forall p' \in D' \; [\alpha(p) \preceq p' \Leftrightarrow p \preceq \gamma(p')].$$

where γ is given by equation 2.2,

$$\gamma(p') = \bigsqcup \Gamma_\alpha(p').$$

It follows from item 2 that the necessary least upper bound exists so that $\gamma(p')$ is defined. We separately prove the forward and backward implications of definition 2.74 as follows. For the forward direction,

$\alpha(p) \preceq p'$

as noted above in formula 2.4

$\Leftrightarrow \quad p \in \Gamma_\alpha(p')$

by the definition of least upper bound (def. 2.81)

$\Rightarrow \quad p \preceq \bigsqcup \Gamma_\alpha(p')$

by the definition of $\gamma(p')$, above

$\Leftrightarrow \quad p \preceq \gamma(p')$.

For the reverse direction, let p be an agent in D. It follows from the definition of $\Gamma_\alpha(p')$ (def. 2.86) that

$$p \in \Gamma_\alpha(p') \Leftrightarrow \alpha(p) \preceq p'.$$

Therefore, by definition 2.80, p' is an upper bound of $\alpha(\Gamma_\alpha(p'))$. Hence, by definition 2.81,

$$\bigsqcup \alpha(\Gamma_\alpha(p')) \preceq p'. \tag{2.4}$$

The proof is then completed by the following series of implications.

$p \preceq \gamma(p')$

since α in monotonic, by item 1

$\Rightarrow \quad \alpha(p) \preceq \alpha(\gamma(p'))$

by the definition of $\gamma(p')$, above

$\Leftrightarrow \quad \alpha(p) \preceq \alpha\Big(\bigsqcup \Gamma_\alpha(p')\Big)$

by item 3

$\Leftrightarrow \quad \alpha(p) \preceq \bigsqcup \alpha(\Gamma_\alpha(p'))$

by transitivity and by formula 2.4

$\Rightarrow \quad \alpha(p) \preceq p'$.

$\square$

These results give necessary and sufficient conditions for a function α to be the abstraction map of a Galois connection (thm. 2.91), and characterize the uniquely determined concretization map (cor. 2.88). Similarly, it is possible to characterize the abstraction map of a Galois connection in terms of the concretization function. We here give the definitions and state the results without proof.

Definition 2.92. Let D and D' be partially ordered sets of agents, and let γ be a function from D' to D. Define Δ_γ to be the function from D to $2^{D'}$ such that for all agents $p \in D$,

$$\Delta_\gamma(p) = \{p' \in D' : p \preceq \gamma(p')\}.$$

Theorem 2.93. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D'. For all $p' \in D'$,

$$\alpha(p) \in \Delta_\gamma(p). \tag{2.5}$$

$$\alpha(p) = \bigsqcap \Delta_\gamma(p) \tag{2.6}$$

Corollary 2.94. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ and $\langle \alpha', \gamma' \rangle$ be Galois connections from D to D'. If $\gamma = \gamma'$ then $\alpha = \alpha'$.

In other words, if γ is the concretization map of a Galois connection, the corresponding abstraction map γ is uniquely determined.

Theorem 2.95. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D'. Let $p' \in D'$ be an agent such that for all agents $p'_1 \in D'$, if $\gamma(p') = \gamma(p'_1)$ then $p' = p'_1$. Then $\alpha(\gamma(p')) = p'$.

Theorem 2.96. Let D and D' be partially ordered sets of agents, and let $\langle \alpha, \gamma \rangle$ be a Galois connection from D to D'. If D'_0 is a subset of D' such that $\bigsqcap D'_0$ is defined, then

$$\gamma\Big(\bigsqcap D'_0\Big) = \bigsqcap \gamma(D'_0),$$

where γ is naturally extended to sets.

Theorem 2.97. Let D and D' be partially ordered sets of agents. A function γ from D' to D is the concretization map of some Galois connection if and only if for all p in D,

1. γ is monotonic,

2. $\Delta_\gamma(p)$ contains a unique minimal element, which implies that $\bigsqcap \Delta_\gamma(p)$ is defined and is an element of $\Delta_\gamma(p)$, and

3. $\gamma\Big(\bigsqcap \Delta_\gamma(p)\Big) = \bigsqcap \gamma(\Delta_\gamma(p))$, where γ is naturally extended to sets.

In a Galois connection, if p is an agent in D, then $p \preceq \gamma(\alpha(p))$ (see thm. 2.76). In other words, going to D' through α and then back through γ always results in an agent that is greater than or equal to the one that we started from. Alternatively, a pair of functions may be such that the resulting agent is always less than or equal to the original agent. This is the case, for example, if we invert the direction of the refinement relationship in the definition of a Galois connection. We refer to this kind of connection as a co-Galois connection [92].

Definition 2.98 (Co-Galois Connection). Let D and D' be partially ordered sets of agents. A co-Galois connection $\langle \alpha, \gamma \rangle$ from D to D' consists of an abstraction map $\alpha : D \mapsto D'$ and a concretization map $\gamma : D' \mapsto D$ such that for all $p \in D$ and $p' \in D'$,

$$p' \preceq \alpha(p) \iff \gamma(p') \preceq p.$$

The choice of name is intentional. In fact, a co-Galois connections is simply a Galois connection that goes in the reverse direction, as shown by the following result.

Lemma 2.99. Let D and D' be partially ordered sets of agents. Then $\langle \alpha, \gamma \rangle$ is a Galois connection from D to D' if and only if $\langle \gamma, \alpha \rangle$ is a co-Galois connection from D' to D.

Proof: The result is immediate from the definitions. $\square$

It follows that the abstraction and concretization maps of a co-Galois connection can be characterized in terms of the corresponding map, as shown above for Galois connections.

2.7.2 Conservative Approximations and Galois Connections

In the rest of this section we will use Galois and co-Galois connections in combination. To simplify the presentation we will systematically take advantage of the result of lemma 2.99 and always refer to a Galois connection from $\mathcal{Q}'$ to $\mathcal{Q}$ in place of a co-Galois connection from $\mathcal{Q}$ to $\mathcal{Q}'$. For our notation, we will use symbols $\langle \alpha_u, \gamma_u \rangle$ for a Galois connection from $\mathcal{Q}$ to $\mathcal{Q}'$, and $\langle \gamma_l, \alpha_l \rangle$ for a Galois connection from $\mathcal{Q}'$ to $\mathcal{Q}$. This choice will be made clear later by our results on the correspondence between conservative approximations and abstract interpretations.

The first result shows that a pair of Galois connections $\langle \alpha_u, \gamma_u \rangle$ and $\langle \gamma_l, \alpha_l \rangle$ forms a conservative approximation if and only if $\gamma_u \preceq \gamma_l$.

Theorem 2.100. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. Then the following two statements are equivalent:

1. For all agents $p' \in \mathcal{Q}'.D$, $\gamma_u(p') \preceq \gamma_l(p')$.
2. For all agents p_1 and p_2 in $\mathcal{Q}.D$, $\alpha_u(p_1) \preceq \alpha_l(p_2) \Rightarrow p_1 \preceq p_2$.

Proof: For the forward direction ($1 \Rightarrow 2$), let p_1 and p_2 be agents from $\mathcal{Q}.D$, and assume $\alpha_u(p_1) \preceq \alpha_l(p_2)$. Since $\langle \alpha_u, \gamma_u \rangle$ is a Galois connection, by theorem 2.76, $p_1 \preceq \gamma_u(\alpha_u(p_1))$. The proof is then completed by the following series of implications.

$p_1 \preceq \gamma_u(\alpha_u(p_1))$

since by hypothesis $\alpha_u(p_1) \preceq \alpha_l(p_2)$,

and since, by theorem 2.75, γ_u is monotonic

$\Rightarrow \quad p_1 \preceq \gamma_u(\alpha_l(p_2))$

since by hypothesis, $\gamma_u \preceq \gamma_l$

$\Rightarrow \quad p_1 \preceq \gamma_l(\alpha_l(p_2))$

since, by theorem 2.76, $\gamma_l(\alpha_l(p_2)) \preceq p_2$, and by transitivity

$\Rightarrow \quad p_1 \preceq p_2$.

For the reverse direction ($2 \Rightarrow 1$), let $p' \in \mathcal{Q}'.D$ be an agent. By theorem 2.76, $\alpha_u(\gamma_u(p')) \preceq p'$ and $p' \preceq \alpha_l(\gamma_l(p'))$. Therefore, by transitivity, $\alpha_u(\gamma_u(p')) \preceq \alpha_l(\gamma_l(p'))$ and consequently, by hypothesis, $\gamma_u(p') \preceq \gamma_l(p')$. □

Corollary 2.101. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. Then (α_l, α_u) is a conservative approximation if and only if for all agents $p' \in \mathcal{Q}'.D$, $\gamma_u(p') \preceq \gamma_l(p')$.

Corollary 2.101 justifies the following definition.

Definition 2.102 (Conservative Approx. induced by a pair of Galois connections). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$ such that for all agents $p' \in \mathcal{Q}'.D$, $\gamma_u(p') \preceq \gamma_l(p')$.

By corollary 2.101 (above), (α_l, α_u) is a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$, which we call a *conservative approximation induced by the pair of Galois connections* $\langle \alpha_u, \gamma_u \rangle$ and $\langle \gamma_l, \alpha_l \rangle$.

The second result characterizes the inverse of the conservative approximation induced by a pair of Galois connection. It shows that the inverse is defined if and only if γ_u and γ_l are equal, and are "mutually" injective.

Theorem 2.103. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$ such that for all agents $p' \in \mathcal{Q}'.D$, $\gamma_u(p') \preceq \gamma_l(p')$. Then for all agents $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}'.D$ the following two statements are equivalent:

1. $\alpha_u(p) = \alpha_l(p) = p'$
2. - $\gamma_u(p') = \gamma_l(p') = p$, and
   - if $p'_1 \in \mathcal{Q}'.D$ is an agent such that $\gamma_u(p'_1) = \gamma_l(p'_1) = p$, then $p'_1 = p'$.

Proof: For the forward direction ($1 \Rightarrow 2$), let p be an agent from $\mathcal{Q}.D$, and assume $\alpha_u(p) = \alpha_l(p) = p'$. Since $\langle \alpha_u, \gamma_u \rangle$ is a Galois connection, by theorem 2.76, $p \preceq \gamma_u(\alpha_u(p))$. Consider the following series of implications.

$p \preceq \gamma_u(\alpha_u(p))$

since by hypothesis $\alpha_u(p) = \alpha_l(p)$

$\Rightarrow \quad p \preceq \gamma_u(\alpha_u(p)) = \gamma_u(\alpha_l(p))$

since by hypothesis, $\gamma_u \preceq \gamma_l$

$\Rightarrow \quad p \preceq \gamma_u(\alpha_u(p)) = \gamma_u(\alpha_l(p)) \preceq \gamma_l(\alpha_l(p))$

by theorem 2.76

$\Rightarrow \quad p \preceq \gamma_u(\alpha_u(p)) \preceq \gamma_l(\alpha_l(p)) \preceq p$

since by hypothesis $\alpha_u(p) = \alpha_l(p) = p'$

$\Rightarrow \quad p \preceq \gamma_u(p') \preceq \gamma_l(p') \preceq p.$

Therefore $\gamma_u(p') = \gamma_l(p') = p$. Let now $p'_1 \in \mathcal{Q}'.D$ be such that $\gamma_u(p'_1) = \gamma_l(p'_1) = p$. Then, since $\langle \alpha_u, \gamma_u \rangle$ and $\langle \alpha_l, \gamma_l \rangle$ are Galois connections, and since by hypothesis $p \preceq \gamma_u(p'_1)$ and $\gamma_l(p'_1) \preceq p$,

$$\alpha_u(p) \preceq p'_1 \preceq \alpha_l(p).$$

Since by hypothesis $\alpha_u(p) = \alpha_l(p) = p'$, $p' \preceq p'_1 \preceq p'$. Therefore, $p'_1 = p'$.

For the reverse direction ($2 \Rightarrow 1$), let $p' \in \mathcal{Q}'.D$ be an agent such that $\gamma_u(p') = \gamma_l(p') = p$, and assume that if $p'_1 \in \mathcal{Q}'.D$ is such that $\gamma_u(p'_1) = \gamma_l(p_1) = p$ then $p'_1 = p'$. Note that because $p = \gamma_u(p')$, it is also $p \preceq \gamma_u(p')$, and therefore, since $\langle \alpha_u, \gamma_u \rangle$ is a Galois connection, $\alpha_u(p) \preceq p'$. Then, consider the following series of implications that start from the result of theorem 2.76:

$$
\begin{aligned}
& p \preceq \gamma_u(\alpha_u(p)) \\
& \quad \text{since by hypothesis, } \gamma_u \preceq \gamma_l \\
& \Rightarrow \quad p \preceq \gamma_u(\alpha_u(p)) \preceq \gamma_l(\alpha_u(p)) \\
& \quad \text{since by the argument above } \alpha_u(p) \preceq p' \text{ and since } \gamma_l \text{ is monotonic (by thm. 2.75)} \\
& \Rightarrow \quad p \preceq \gamma_u(\alpha_u(p)) \preceq \gamma_l(\alpha_u(p)) \preceq \gamma_l(p') \\
& \quad \text{since by hypothesis } \gamma_l(p') = p \\
& \Rightarrow \quad p \preceq \gamma_u(\alpha_u(p)) \preceq \gamma_l(\alpha_u(p)) \preceq \gamma_l(p') = p.
\end{aligned}
$$

Therefore, $\gamma_u(\alpha_u(p)) = \gamma_l(\alpha_u(p)) = p$. Hence, by hypothesis, $\alpha_u(p) = p'$. The proof that $\alpha_l(p) = p'$ is similar. □

Corollary 2.104. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. Assume $\Psi = (\alpha_l, \alpha_u)$ is a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. Then for all agents $p' \in \mathcal{Q}'.D$, $\Psi_{inv}(p')$ is defined and $\Psi_{inv}(p') = p$ if and only if

- $\gamma_u(p') = \gamma_l(p') = p$, and
- if $p'_1 \in \mathcal{Q}'.D$ is an agent such that $\gamma_u(p'_1) = \gamma_l(p'_1) = p$, then $p'_1 = p'$.

Given Galois connections $\langle \alpha_u, \gamma_u \rangle$ and $\langle \gamma_l, \alpha_l \rangle$, the closer γ_l is to γ_u, the tighter the resulting conservative approximation, as shown by the next result.

Corollary 2.105. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. Let $\langle \gamma'_l, \alpha'_l \rangle$ be a Galois connection between $\mathcal{Q}'.D$ and $\mathcal{Q}.D$ such that for all agents $p' \in \mathcal{Q}'.D$,

$$\gamma_u(p') \preceq \gamma_l(p') \preceq \gamma'_l(p').$$

Then $\Psi = (\alpha_l, \alpha_u)$ and $\Psi' = (\alpha'_l, \alpha_u)$ are conservative approximations such that for all agents $p \in \mathcal{Q}.D$,

$$\alpha'_l(p) \preceq \alpha_l(p).$$

Proof: The result follows from corollary 2.101 and theorem 2.78. □

Given a Galois connection $\langle \alpha_u, \gamma_u \rangle$ between $\mathcal{Q}.D$ and $\mathcal{Q}'.D$, the tightest conservative approximation induced by a pair of Galois connections is obtained by a Galois connection $\langle \gamma, \alpha_l \rangle$ from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$ such that $\gamma_u = \gamma_l$. Note that γ_u is a concretization function and that γ_l is an abstraction function of their respective Galois connections. Therefore, in order for γ_u to be equal to γ_l, γ_u must necessarily satisfy all the conditions of both theorem 2.97 and theorem 2.91. However, while γ_u clearly satisfies the conditions of theorem 2.97 (since it is the concretization function of a Galois connection), it does not necessarily satisfy the condition of theorem 2.91. In other words, γ_u is not in general the abstraction function of any Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. In that case, several “maximal” approximations may exist, but no tightest approximation.

Our last result gives sufficient conditions for a conservative approximation to form a pair of Galois connections. It is sufficient that the upper and lower bound be monotonic (which is a necessary condition for Galois connections), and that the inverse of the conservative approximation be defined everywhere.

Theorem 2.106. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be agent algebras and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ such that

1. Ψ_u and Ψ_l are monotonic, and
2. $\Psi_{inv}(p')$ is defined for all $p' \in \mathcal{Q}'.D$.

Then

- $\langle \Psi_u, \Psi_{inv} \rangle$ is a Galois connection from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$, and
- $\langle \Psi_{inv}, \Psi_l \rangle$ is a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$.

Proof: We show that $\langle \Psi_u, \Psi_{inv} \rangle$ is a Galois connection by proving that for all agents $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}'.D$,

$$\Psi_u(p) \preceq p' \Leftrightarrow p \preceq \Psi_{inv}(p').$$

We separately prove the forward and backward implications as follows.

$\Psi_u(p) \preceq p'$

by definition 2.55

$\Leftrightarrow \quad \Psi_u(p) \preceq \Psi_l(\Psi_{inv}(p'))$

by definition 2.52, since (Ψ_l, Ψ_u) is a conservative approximation

$\Rightarrow \quad p \preceq \Psi_{inv}(p')$.

Similarly,

$p \preceq \Psi_{inv}(p')$

since, by hypothesis, Ψ_u is monotonic

$\Rightarrow \quad \Psi_u(p) \preceq \Psi_u(\Psi_{inv}(p'))$

by definition 2.55

$\Leftrightarrow \quad \Psi_u(p) \preceq p'$.

The proof that $\langle \Psi_{inv}, \Psi_l \rangle$ is a Galois connection is similar. □

In the previous result, the condition that Ψ_{inv} be defined everywhere is crucial. In fact, there are monotonic conservative approximations such that the abstraction functions are not abstraction maps of any Galois connections. This occurs when the equivalence classes induced by Ψ_u and Ψ_l do not have the necessary greatest and lowest element (see thm. 2.59).

2.7.3 Abstract Interpretations

Abstract interpretations were originally developed for static analysis of sequential programs in optimizing compilers [22]. They have also been used for abstracting and formally verifying models of both sequential and reactive systems. This section discusses the relationship between abstract interpretations and conservative approximations.

In the theory of abstract interpretations, a poset is used to model, for example, the data values that can be manipulated by a computer program. Functions over the poset represent the primitive operations available to the program. An abstract interpretation provides a formalization of what it means for one poset (and it's associated functions) to be an abstraction of another.

As we did for Galois connections, we restrict the definition here to posets that are sets of agents. The standard definition of an abstract interpretation [23] also designates a least element

$\perp$ that is used as the starting point for least fixed point computations. We have no need for such computations, so we have removed $\perp$ from the definition.

Definition 2.107 (Abstract Interpretation). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras. Then $\mathcal{Q}'$ is an abstract interpretation of $\mathcal{Q}$ if and only if there exists a Galois connection $\langle \alpha, \gamma \rangle$ from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ such that for all p'_1 and p'_2 in $\mathcal{Q}'.D$,

1. if $p'_1 \parallel p'_2$ is defined, then $\alpha(\gamma(p'_1) \parallel \gamma(p'_2)) \preceq p'_1 \parallel p'_2$,

2. if $\mathit{proj}(B)(p'_1)$ is defined, then $\alpha(\mathit{proj}(B)(\gamma(p'_1))) \preceq \mathit{proj}(B)(p'_1)$, and

3. if $\mathit{rename}(r)(p'_2)$ is defined, then $\alpha(\mathit{rename}(r)(\gamma(p'_2))) \preceq \mathit{rename}(r)(p'_2)$.

The three conditions of definition 2.107 are equivalent to the conditions S1 through S3 of theorem 2.63, as shown in the next theorem.

Theorem 2.108. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras. Then $\mathcal{Q}'$ is an abstract interpretation of $\mathcal{Q}$ if and only if there exists a function α from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ that is the abstraction function of some Galois connection and such that for all p_1 and p_2 in $\mathcal{Q}.D$,

1. if $\alpha(p_1) \parallel \alpha(p_2)$ is defined, then $\alpha(p_1 \parallel p_2) \preceq \alpha(p_1) \parallel \alpha(p_2)$,

2. if $\mathit{proj}(B)(\alpha(p_1))$ is defined, then $\alpha(\mathit{proj}(B)(p_1)) \preceq \mathit{proj}(B)(\alpha(p_1))$, and

3. if $\mathit{rename}(r)(\alpha(p_2))$ is defined, then $\alpha(\mathit{rename}(r)(p_2)) \preceq \mathit{rename}(r)(\alpha(p_2))$.

Proof: We only give the proof for the composition operator case. The projection operator case and the rename operator case are analogous (but notationally simpler, since they involve a unary operator rather than a binary operator).

To prove the forward implication, assume that $\mathcal{Q}'$ is an abstract interpretation of $\mathcal{Q}$. This implies by definition 2.107 that there is a Galois connection $\langle \alpha, \gamma \rangle$ from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and for all p'_1 and p'_2 in $\mathcal{Q}'.D$, if $p'_1 \parallel p'_2$ is defined,

$$\alpha(\gamma(p'_1) \parallel \gamma(p'_2)) \preceq p'_1 \parallel p'_2.$$

Let p_1 and p_2 be in $\mathcal{Q}.D$ and assume that $\alpha(p_1) \parallel \alpha(p_2)$ is defined. The desired result can be derived as follows:

$\alpha(p_1) \parallel \alpha(p_2)\downarrow$

by definition 2.107

$\Rightarrow \quad \alpha(\gamma(\alpha(p_1)) \parallel \gamma(\alpha(p_2))) \preceq \alpha(p_1) \parallel \alpha(p_2)$

since by theorem 2.76, $p_1 \preceq \gamma(\alpha(p_1))$ and $p_2 \preceq \gamma(\alpha(p_2))$

and since parallel composition is $\top$-monotonic

$\Rightarrow \quad \alpha(p_1 \parallel p_2) \preceq \alpha(p_1) \parallel \alpha(p_2).$

To prove the reverse implication, assume there is a Galois connection $\langle \alpha, \gamma \rangle$ from D to D' and that for all p_1 and p_2 in D, if $\alpha(p_1) \parallel \alpha(p_2)$ is defined, then

$$\alpha(p_1 \parallel p_2) \preceq \alpha(p_1) \parallel \alpha(p_2).$$

Let p'_1 and p'_2 be agents in D' such that $p'_1 \parallel p'_2$ is defined. The desired result can be derived as follows:

$p'_1 \parallel p'_2\downarrow$

since by theorem 2.76, $\alpha(\gamma(p'_1)) \preceq p_1$ and $\alpha(\gamma(p'_2)) \preceq p_2$

and since parallel composition is $\top$-monotonic

$\Rightarrow \quad \alpha(\gamma(p'_1)) \parallel \alpha(\gamma(p'_2)) \preceq p'_1 \parallel p'_2$

by hypothesis

$\Rightarrow \quad \alpha(\gamma(p'_1) \parallel \gamma(p'_2)) \preceq p'_1 \parallel p'_2.$

□

Abstract interpretations induce *compositional* conservative approximations.

Corollary 2.109. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\mathcal{Q}'$ be an abstract interpretation of $\mathcal{Q}$ by a Galois connection $\langle \alpha_u, \gamma_u \rangle$. Let $\langle \gamma_l, \alpha_l \rangle$ be Galois connection between $\mathcal{Q}'.D$ and $\mathcal{Q}.D$. Then the following two statements are equivalent:

- For all $p' \in \mathcal{Q}'.D$, $\gamma_u(p') \preceq \gamma_l(p')$.
- (α_l, α_u) is a compositional conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$.

Proof: The result follows from theorem 2.108, corollary 2.101 and theorem 2.63. □

The inverse of the conservative approximation is again characterized as already discussed in corollary 2.104 and corollary 2.105.

Abstract interpretations are used in program analysis because they preserve the application of the operators from the abstract model to the concrete model. This well known result of the theory of abstract interpretations is proved below.

Theorem 2.110. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\mathcal{Q}'$ be an abstract interpretation of $\mathcal{Q}$ by a Galois connection $\langle \alpha_u, \gamma_u \rangle$. Then for all agents p_1 and p_2 in $\mathcal{Q}.D$,

1. if $\alpha(p_1) \parallel \alpha(p_2)$ is defined, then $p_1 \parallel p_2 \preceq \gamma(\alpha(p_1) \parallel \alpha(p_2))$,
2. if $\mathit{proj}(B)(\alpha(p_1))$ is defined, then $\mathit{proj}(B)(p_1) \preceq \gamma(\mathit{proj}(B)(\alpha(p_1)))$, and
3. if $\mathit{rename}(r)(\alpha(p_2))$ is defined, then $\mathit{rename}(r)(p_2) \preceq \gamma(\mathit{rename}(r)(\alpha(p_2)))$.

Proof: We only give the proof for the composition operator case. Let p_1 and p_2 be agents in $\mathcal{Q}.D$ such that $\alpha(p_1) \parallel \alpha(p_2)$ is defined. The desired results can be derived as follows:

$\alpha(p_1) \parallel \alpha(p_2)\downarrow$

by theorem 2.108

$\Rightarrow \quad \alpha(p_1 \parallel p_2) \preceq \alpha(p_1) \parallel \alpha(p_2)$

since, by theorem 2.75, γ is monotonic

$\Rightarrow \quad \gamma(\alpha(p_1 \parallel p_2)) \preceq \gamma(\alpha(p_1) \parallel \alpha(p_2))$

by theorem 2.76

$\Rightarrow \quad p_1 \parallel p_2 \preceq \gamma(\alpha(p_1 \parallel p_2)) \preceq \gamma(\alpha(p_1) \parallel \alpha(p_2))$

□

Corollary 2.111. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\mathcal{Q}'$ be an abstract interpretation of $\mathcal{Q}$ by a Galois connection $\langle \alpha_u, \gamma_u \rangle$. Let E be a closed expression over $\mathcal{Q}$. If $[\![E[p/\alpha(p)]]\!]$ is defined, then

$$[\![E]\!] \preceq \gamma([\![E[p/\alpha(p)]]\!]).$$

Proof: By induction on the structure of expressions, by theorem 2.110. □

Corollary 2.109 shows that abstract interpretations and compositional conservative approximations are very closely related. The two, however, serve very different purposes. Abstract interpretations employ a single Galois connection, and can be used to approximate the evaluation of an expression at the concrete level by the concretization of the evaluation of the corresponding expression at the abstract level, as shown by corollary 2.111. The abstract interpretation guarantees that the result computed at the concrete level conforms to the one computed at the abstract level, where, presumably, the computation is more efficient. If a property φ is preserved by the refinement relationship, then if the evaluation at the abstract level has the property φ, also the evaluation at the concrete level has the property φ. Abstract interpretations, however, are unable to guarantee that a positive refinement verification result at the abstract level implies a positive refinement verification result at the concrete level. In other words, if $\alpha(p_1) \preceq \alpha(p_2)$, then $p_1 \preceq p_2$ is not necessarily true.

Conservative approximations, on the other hand, employ two mappings to guarantee the above verification result. Similarly, abstract interpretations must use a second Galois connection, that connects the abstract to the concrete domain. Corollary 2.109 gives necessary and sufficient conditions for the pair of Galois connections to form a compositional conservative approximation, while corollary 2.104 characterizes the inverse of the conservative approximation. Conservative approximation are however more general, since, unlike Galois connections, the mappings of a conservative approximation are not required to be monotonic. Even if the mappings are monotonic, there are conservative approximations that cannot be expressed in terms of Galois connections, since the necessary least upper bounds and greatest lower bounds do not necessarily exist (cfr. thm. 2.97 and thm. 2.91). We therefore view conservative approximations and abstract interpretations as related, but complementary, concepts.

2.8 Modeling Heterogeneous Systems

In this section we study a model of interaction for agents that belong to two different agent algebras $\mathcal{Q}_1$ and $\mathcal{Q}_2$. If $p_1 \in \mathcal{Q}_1$ and $p_2 \in \mathcal{Q}_2$, there obviously isn't a composition operator defined on the pair p_1 and p_2. One may try, however, to compose the agents according to the parallel composition of either model. This is possible, for example, if there exists a conservative approximation from one algebra to the other, such that the inverse is defined at the agents that are being considered. In that case, in fact, the agent can be represented exactly in both models, and we can choose the representation that best fits our needs.

In the more general case where the inverse is not defined, we must, instead, define the composition in terms of the operators of a third agent algebra $\mathcal{Q}$ which is related to $\mathcal{Q}_1$ and $\mathcal{Q}_2$ by appropriate conservative approximations. We refer to $\mathcal{Q}$ as a "common refinement" of $\mathcal{Q}_1$ and $\mathcal{Q}_2$.

The algebra $\mathcal{Q}$ must be chosen carefully, since we must require that the inverse of the conservative approximations be defined for the agents that we wish to compose. Note that we do not require that the inverses also be embeddings of agent algebras, since this condition could be too strong in many practical cases. Instead, when the inverses are not embeddings, we show that the composition that is obtained in the more concrete model is a refinement of the composition had the inverses been embeddings. In other words, we are simply losing some of the flexibility in the implementation.

2.8.1 Abstraction and Refinement

We have argued in section 2.6 that a single function Ψ_u is not sufficient to characterize an abstraction, and that a second function, which we called Ψ_l, was needed to identify both the upper and lower bound of the abstraction. By doing so, we were able to determine which agents could be represented *exactly* at the abstract level, which led us to the notion of the inverse of the abstraction. Refinement, that is the notion of a correspondence that goes from the more abstract to the more concrete agent model, is no different in our framework, and is symmetrically represented as a pair of functions that form a conservative approximation. Thus, our notion of refinement does not correspond exactly to the inverse of the abstraction, since, as we have noted, the inverse may not be defined for all agents. Nonetheless, we require that if the inverse *is* defined for some agent, then the refinement maps are equal to the inverse. In addition, we will consider an agent model $\mathcal{Q}'$ to be at a *higher level of abstraction* than an agent model $\mathcal{Q}$ whenever every agent in $\mathcal{Q}'$ can be represented exactly by an agent in $\mathcal{Q}$.

In the following we will restrict our attention to conservative approximations induced by a pair of Galois connections. In fact, because abstraction and refinement are symmetric, Galois connections are particularly well behaved and make it easy to derive the tight relationship that exists between the abstraction and the refinement functions. In particular, in the previous sections we have considered agent algebras $\mathcal{Q}$ and $\mathcal{Q}'$ related by a Galois connection $\langle \alpha_u, \gamma_u \rangle$ from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$, and by a Galois connection $\langle \gamma_l, \alpha_l \rangle$ from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. We have shown that (α_l, α_u) forms a conservative approximation if and only if for all $p' \in \mathcal{Q}'.D$, $\gamma_u(p') \preceq \gamma_l(p')$ (theorem 2.100). In addition, if α_u satisfies certain properties, then the conservative approximation is also compositional (see corollary 2.109). It is easy to change our point of view and consider constructing a conservative approximation from $\mathcal{Q}'$ to $\mathcal{Q}$. Observe that our hypothesis are symmetric relative to $\mathcal{Q}$ and $\mathcal{Q}'$. Thus our previous results can be restated by simply replacing all occurrences of α_u by γ_l, and all occurrences of γ_u by α_l, and by exchanging the domains of agents. In particular, we here restate theorem 2.100, corollary 2.101, corollary 2.104 and corollary 2.109.

Theorem 2.112. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. Then the following two statements are equivalent:

1. For all agents $p \in \mathcal{Q}.D$, $\alpha_l(p) \preceq \alpha_u(p)$.
2. For all agents p'_1 and p'_2 in $\mathcal{Q}'.D$, $\gamma_l(p'_1) \preceq \gamma_u(p'_2) \Rightarrow p'_1 \preceq p'_2$.

Corollary 2.113. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. Then (γ_u, γ_l) is a conservative approximation if and only if for all agents $p \in \mathcal{Q}.D$, $\alpha_l(p) \preceq \alpha_u(p)$.

Corollary 2.114. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. Assume $\Psi' = (\gamma_u, \gamma_l)$ is a conservative approximation from $\mathcal{Q}'$ to $\mathcal{Q}$. Then for all agents $p \in \mathcal{Q}.D$, $\Psi'_{inv}(p)$ is defined and $\Psi'_{inv}(p) = p'$ if and only if

- $\alpha_l(p) = \alpha_u(p) = p'$, and
- if $p_1 \in \mathcal{Q}.D$ is an agent such that $\alpha_l(p_1) = \alpha_u(p_1) = p'$, then $p_1 = p$.

Corollary 2.115. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras, and let $\mathcal{Q}$ be an abstract interpretation of $\mathcal{Q}'$ by a Galois connection $\langle \gamma_l, \alpha_l \rangle$ from $\mathcal{Q}'$ to $\mathcal{Q}$. Let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connection between $\mathcal{Q}.D$ and $\mathcal{Q}'.D$. Then the following two statements are equivalent:

- For all $p \in \mathcal{Q}.D$, $\alpha_l(p) \preceq \alpha_u(p)$.
- (γ_u, γ_l) is a compositional conservative approximation from $\mathcal{Q}'$ to $\mathcal{Q}$.

Suppose now that $\mathcal{Q}$ and $\mathcal{Q}'$ are agent algebras, and that $\Psi = (\alpha_l, \alpha_u)$ is a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ induced by a pair of Galois connections $\langle \alpha_u, \gamma_u \rangle$ and $\langle \gamma_l, \alpha_l \rangle$. Corollary 2.113 shows that in order for $\Psi' = (\gamma_u, \gamma_l)$ to be a conservative approximation from $\mathcal{Q}'$ to $\mathcal{Q}$ we need that $\alpha_l(p) \preceq \alpha_u(p)$ for all agents $p \in \mathcal{Q}.D$. This condition is commonly satisfied by a conservative approximation Ψ, and simply formalizes the intuition that the lower bound of an agent must be less than or equal to its upper bound (although, as noted earlier, this is not a necessary condition for a conservative approximation).

Note that the inverses Ψ_{inv} and Ψ'_{inv} of the conservative approximations are inverse of each other, as shown by the next result.

Theorem 2.116. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\langle \alpha_u, \gamma_u \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ and $\langle \gamma_l, \alpha_l \rangle$ a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$. Assume $\Psi = (\alpha_u, \alpha_l)$ is a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$, and that $\Psi' = (\gamma_u, \gamma_l)$ is a conservative approximation from $\mathcal{Q}'$ to $\mathcal{Q}$. Then, for all $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}'.D$,

$$\Psi_{inv}(p') = p \;\Leftrightarrow\; \Psi'_{inv}(p) = p'.$$

Proof: The two implications are symmetric, so we will only prove the forward direction. Assume $\Psi_{inv}(p') = p$. Then, by definition of inverse of conservative approximation, $\alpha_u(p) = \alpha_l(p) = p'$. In addition, by lemma 2.54, if p_1 is such that $\alpha_u(p_1) = \alpha_l(p_1) = p'$, then $p_1 = p$. Therefore, by corollary 2.114, $\Psi'_{inv}(p) = p'$. □

The situation is therefore the one depicted in figure 2.1, where Galois connections are denoted by pairs of dotted arcs and by a straight arrow that indicates the direction of the connection. The shaded region in $\mathcal{Q}$ corresponds to the set of agents that can be represented exactly in $\mathcal{Q}'$. This

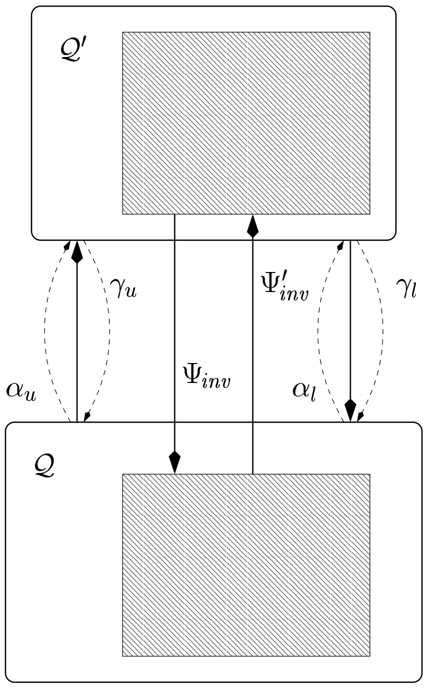

Figure 2.1: Abstraction, Refinement and their inverses

region is isomorphic to the corresponding shaded region in $\mathcal{Q}'$ which consists of the agents in $\mathcal{Q}'$ that can be represented exactly in $\mathcal{Q}$. In other words, a subset of the agents of the two semantic domains can be represented indifferently in either domain, while the remaining agents can only be approximated by the other domain (i.e., their upper and lower bound do not coincide).

If $\mathcal{Q}'$ is strictly more abstract than $\mathcal{Q}$, in the sense that the agents in $\mathcal{Q}'$ contain strictly less information than those in $\mathcal{Q}$, then Ψ_{inv} is total (assuming Ψ is the tightest conservative approximation), and therefore, by theorem 2.58, it preserves the refinement relationship in both directions (since Galois connections are always monotonic). In that case, the conservative approximation from $\mathcal{Q}'$ to $\mathcal{Q}$ is essentially an embedding of $\mathcal{Q}'.D$ into $\mathcal{Q}.D$, equipped with the respective orders, and the shaded region in $\mathcal{Q}'$ would extend to the whole domain.

If $\mathcal{Q}'$ is an abstract interpretation of $\mathcal{Q}$, then if $\Psi = (\alpha_l, \alpha_u)$ is a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ then Ψ is also compositional. In this case, while $\Psi' = (\gamma_u, \gamma_l)$ may still be a conservative approximation from $\mathcal{Q}'$ to $\mathcal{Q}$, it is more difficult to have it also be compositional. This occurs when $\mathcal{Q}$ is an abstract interpretation of $\mathcal{Q}'$. However, by theorem 2.108, $\mathcal{Q}$ is an abstract interpretation of $\mathcal{Q}'$ by the Galois connection $\langle \gamma_l, \alpha_l \rangle$ if and only if

$$\begin{aligned}
\gamma_l(p'_1) \parallel \gamma_l(p'_2)\downarrow \quad &\Rightarrow \quad \gamma_l(p'_1 \parallel p'_2) \preceq \gamma_l(p'_1) \parallel \gamma_l(p'_2), \\
\mathit{proj}(B)(\gamma_l(p'_1))\downarrow \quad &\Rightarrow \quad \gamma_l(\mathit{proj}(B)(p'_1)) \preceq \mathit{proj}(B)(\gamma_l(p'_1)), \\
\mathit{rename}(r)(\gamma_l(p'_2))\downarrow \quad &\Rightarrow \quad \gamma_l(\mathit{rename}(r)(p'_2)) \preceq \mathit{rename}(r)(\gamma_l(p'_2)).
\end{aligned} \tag{2.7}$$

In addition, by corollary 2.104, for agents $p' \in \mathcal{Q}'.D$ such that $\gamma_u(p') = \gamma_l(p')$ (and if γ_u and γ_l are jointly injective on p'), γ_l corresponds to the inverse conservative approximation Ψ_{inv} of $\Psi = (\alpha_l, \alpha_u)$. Therefore, since, by theorem 2.108, α_u satisfies the hypothesis of theorem 2.64, for the agents p' such that $\gamma_u(p') = \gamma_l(p')$, γ_l also satisfies

$$\begin{aligned}
\gamma_l(p'_1 \parallel p'_2)\downarrow \quad &\Rightarrow \quad \gamma_l(p'_1) \parallel \gamma_l(p'_2) \preceq \gamma_l(p'_1 \parallel p'_2), \\
\gamma_l(\mathit{proj}(B)(p'))\downarrow \quad &\Rightarrow \quad \mathit{proj}(B)(\gamma_l(p'_1)) \preceq \gamma_l(\mathit{proj}(B)(p')) \\
\gamma_l(\mathit{rename}(r)(p'))\downarrow \quad &\Rightarrow \quad \mathit{rename}(r)(\gamma_l(p'_1)) \preceq \gamma_l(\mathit{rename}(r)(p')).
\end{aligned} \tag{2.8}$$

The conditions in equation 2.7 and equation 2.8, taken together, imply that the inverse conservative approximation Ψ_{inv} of Ψ is an embedding of agent algebras. In other words, if Ψ_{inv} is defined everywhere, then $\mathcal{Q}'$ must essentially be a subalgebra of $\mathcal{Q}$.

2.8.2 Interaction of Heterogeneous Models

When agents p and p' belong to different agent algebras, we define their composition in terms of the composition rules of either model. To do so, we require that the algebras be related by a pair of conservative approximations, and that the inverse of the conservative approximation be defined.

Definition 2.117 (Co-composition). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be agent algebras related by conservative approximations Ψ from and $\mathcal{Q}$ to $\mathcal{Q}'$ and Ψ' from $\mathcal{Q}'$ to $\mathcal{Q}$ induced by a pair of Galois connections. Let also $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}'.D$ be agents such that $\Psi_{inv}(p')$ is defined. Then the *co-composition of p and p' in the context of $\mathcal{Q}$*, written $p \parallel_{\mathcal{Q}} p'$, is given by

$$p \parallel_{\mathcal{Q}} p' = p \parallel \Psi_{inv}(p').$$

Note that the co-composition of agents that belong to different models is always defined in terms of one of the two models. This is possible when an agent can be represented exactly in the other model, i.e., when the inverse of the conservative approximation is defined. Notice that while the abstraction ensures that the interpretation of p' and $\Psi_{inv}(p')$ is the same in the two models, the rules for composition may be very different. In particular, the result of the co-composition computed in $\mathcal{Q}$ may or may not be represented exactly in $\mathcal{Q}'$.

When the inverse of a conservative approximation $\Psi'_{inv}(p')$ is defined at agent p', then p' can be considered *polymorphic*, in the sense that the agent can be used under different notions of composition. This is similar to the notion of domain polymorphism introduced in the Ptolemy II project [64]. Note how our notion of polymorphism is derived from the particular abstraction being used. The abstraction, in other words, formalizes the interpretation that one model of computation has of the other model. A polymorphic agent under one abstraction may no longer be polymorphic under a different abstraction.

It is possible that both $\Psi_{inv}(p')$ and $\Psi_{inv}(p)$ are defined. In that case the co-composition may be carried either in the context of $\mathcal{Q}$ or in the context of $\mathcal{Q}'$. If the conservative approximation is compositional, then it is possible to relate the result of these composition, as shown next.

Theorem 2.118. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be agent algebras related by conservative approximations Ψ from and $\mathcal{Q}$ to $\mathcal{Q}'$ and Ψ' from $\mathcal{Q}'$ to $\mathcal{Q}$ induced by a pair of Galois connections. Assume Ψ is a compositional conservative approximation satisfying S1 through S3. Let also $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}'.D$ be agents such that $\Psi_{inv}(p')$ and $\Psi'_{inv}(p)$ are defined. Then, if $\Psi_{inv}(p \parallel_{\mathcal{Q}'} p')$ is defined,

$$p \parallel_{\mathcal{Q}} p' \preceq \Psi_{inv}(p \parallel_{\mathcal{Q}'} p').$$

Proof: By definition 2.117,

$$\Psi_{inv}(p \ \|_{\mathcal{Q}'} \ p') = \Psi_{inv}(\Psi'_{inv}(p) \parallel p').$$

Hence, by theorem 2.64,

$$\Psi_{inv}(\Psi'_{inv}(p)) \parallel \Psi_{inv}(p') \preceq \Psi_{inv}(p \ \|_{\mathcal{Q}'} \ p').$$

Therefore, by theorem 2.116,

$$p \parallel \Psi_{inv}(p') = p \ \|_{\mathcal{Q}} \ p' \preceq \Psi_{inv}(p \ \|_{\mathcal{Q}'} \ p').$$

□

The assumptions of theorem 2.118 imply that $\mathcal{Q}'$ is an abstract interpretation of $\mathcal{Q}$. Hence, when the composition is carried out in the more concrete model of computation, then the result is an implementation of the corresponding result in the more abstract model. The equality holds in case Ψ_{inv} is an embedding (see theorem 2.72).

In practice, if two semantic domains $\mathcal{Q}_1$ and $\mathcal{Q}_2$ corresponds to two different models of computation, it is not always clear how to construct Galois connections between them, especially if neither one is strictly more abstract than the other. In that case, it is possible to derive the appropriate connections if a third common refinement $\mathcal{Q}$ of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ is available. For the next result, refer to figure 2.2.

Theorem 2.119. Let $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}$ be partially ordered agent algebras. Let $\langle \alpha_u^1, \gamma_u^1 \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}_1.D$ and $\langle \gamma_l^1, \alpha_l^1 \rangle$ a Galois connection from $\mathcal{Q}_1.D$ to $\mathcal{Q}.D$ and assume that (α_l^1, α_u^1) and (γ_u^1, γ_l^1) are conservative approximations. Similarly, let $\langle \alpha_u^2, \gamma_u^2 \rangle$ be a Galois connections from $\mathcal{Q}.D$ to $\mathcal{Q}_2.D$ and $\langle \gamma_l^2, \alpha_l^2 \rangle$ a Galois connection from $\mathcal{Q}_2.D$ to $\mathcal{Q}.D$ and assume that (α_l^2, α_u^2) and (γ_u^2, γ_l^2) are conservative approximations. Let

$$\begin{aligned}
\alpha_u &= \alpha_u^1 \circ \gamma_l^2, \\
\gamma_u &= \alpha_l^2 \circ \gamma_u^1, \\
\alpha_l &= \alpha_l^1 \circ \gamma_u^2, \\
\gamma_l &= \alpha_u^2 \circ \gamma_l^1.
\end{aligned}$$

Then

1. $\langle \alpha_u, \gamma_u \rangle$ is a Galois connection from $\mathcal{Q}_2.D$ to $\mathcal{Q}_1.D$ and $\langle \gamma_l, \alpha_l \rangle$ is a Galois connection from $\mathcal{Q}_1.D$ to $\mathcal{Q}_2.D$.

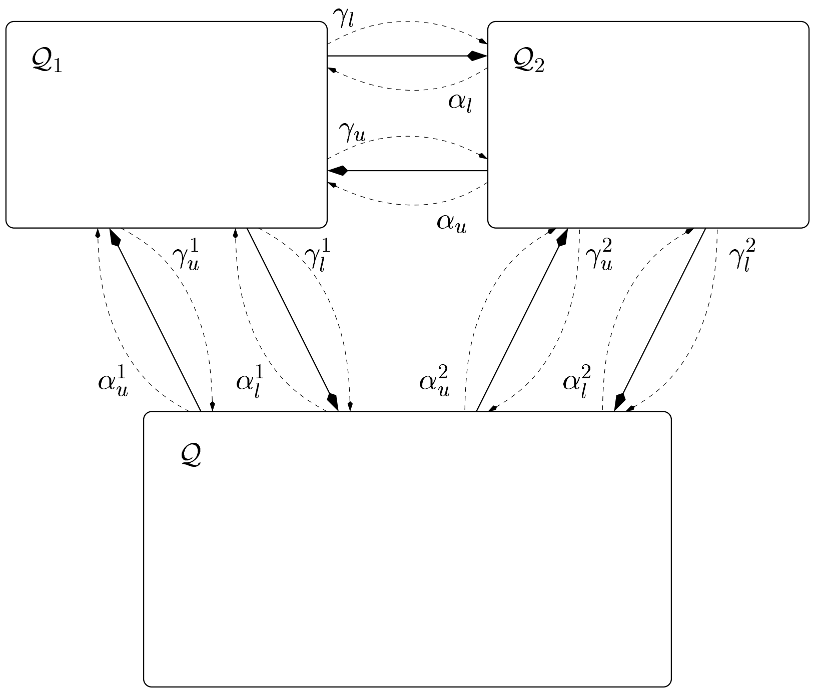

Figure 2.2: Determination of relations through common refinement

2. (α_l, α_u) is a conservative approximation from $\mathcal{Q}_2$ to $\mathcal{Q}_1$ and (γ_u, γ_l) is a conservative approximation from $\mathcal{Q}_1$ to $\mathcal{Q}_2$.

Proof: Item 1 follows easily from theorem 2.79. For item 2, let $p_1 \in \mathcal{Q}_1.D$ be an agent. Then,

$$\begin{aligned}
\gamma_u(p_1) &= \alpha_l^2(\gamma_u^1(p_1)) \\
&\quad \text{since } (\gamma_u^2, \gamma_l^2) \text{ is a conservative approximation, by corollary 2.101,} \\
&\preceq \alpha_u^2(\gamma_u^1(p_1)) \\
&\quad \text{since } (\alpha_l^1, \alpha_u^1) \text{ is a conservative approximation} \\
&\quad \text{and since } \alpha_u^2 \text{ is monotonic, by corollary 2.101,} \\
&\preceq \alpha_u^2(\gamma_l^1(p_1)) \\
&= \gamma_l(p_1).
\end{aligned}$$

Therefore, by corollary 2.101, (α_l, α_u) is a conservative approximation from $\mathcal{Q}_2$ to $\mathcal{Q}_1$. The proof that (γ_u, γ_l) is a conservative approximation from $\mathcal{Q}_1$ to $\mathcal{Q}_2$ is similar. □

The relationship between the agents in $\mathcal{Q}_1$ and the agents in $\mathcal{Q}_2$ is embodied by the derived conservative approximations between the two models. Notice that, in particular, this relationship depends on the conservative approximations that are used to embed $\mathcal{Q}_1$ and $\mathcal{Q}_2$ in their common refinement. If we choose a different refinement, or if we choose a different conservative approximation into the same refinement, the relation between $\mathcal{Q}_1$ and $\mathcal{Q}_2$ will likely change. This is not surprising, as the interaction between agents that belong to different models ultimately depends on the implementation strategy, as already discussed in section 1.3.

The above construction is also useful in case the inverse is not defined on the agents of interest when considering conservative approximations that directly relate the two models of computation $\mathcal{Q}_1$ and $\mathcal{Q}_2$, but it is defined relative to the common refinement $\mathcal{Q}$. In this case, the parallel composition can only then be understood in the context of $\mathcal{Q}$. It is possible, however, to relate the result back to the initial domains.

As an example, consider the problem of composing the two agents $p'_1 \in \mathcal{Q}_1$ and $p'_2 \in \mathcal{Q}_2$ depicted in figure 2.3. Here we assume $\mathcal{Q}$, $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are related by conservative approximations Ψ^1 from $\mathcal{Q}$ to $\mathcal{Q}_1$ and Ψ^2 from $\mathcal{Q}$ to $\mathcal{Q}_2$, and such that $p_1 = \Psi^1_{inv}(p'_1)$ and $p_2 = \Psi^2_{inv}(p'_2)$ are both defined. Because p_1 and p_2 belong to the same semantic domain, we can obtain their composition $p = p_1 \parallel p_2$.

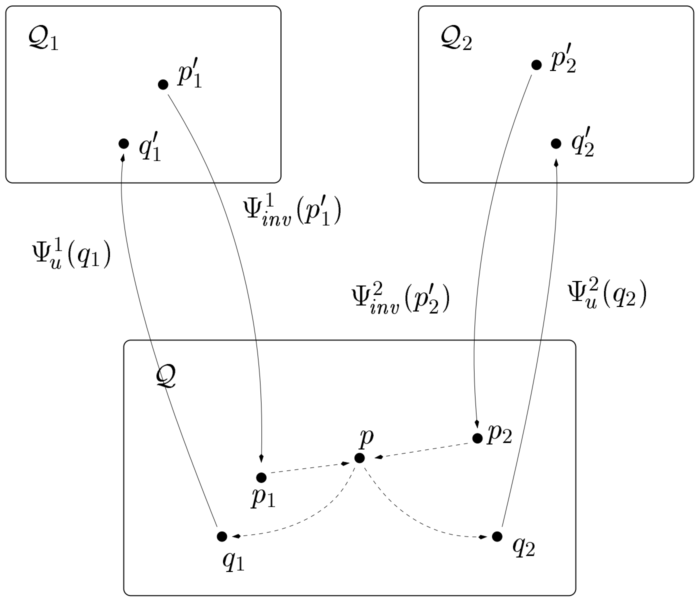

Figure 2.3: Heterogeneous composition in common refinement

We now want to reflect the effect of this composition back to the higher levels of abstraction. To do so, we consider the minimal $q_1 \preceq p_1$ and $q_2 \preceq p_2$ such that the composition of q_1 and q_2 is p (minimality is intended with respect to the agent ordering). These objects represent the behaviors of p_1 and p_2 that are mutually compatible and concur in creating a behavior of the compound object p. In other words, q_1 and q_2 represent the behaviors of p_1 and p_2 as constrained by the composition. These agents exist in many practical cases, and in particular they exist for the trace-based agent algebra models described later in chapter 4. In that case, $q_1 = \mathit{proj}(\alpha(p_1))(p)$ and $q_2 = \mathit{proj}(\alpha(p_2))(p)$. The abstractions $q'_1 = \Psi^1_u(q_1)$ and $q'_2 = \Psi^2_u(q_2)$ represent at the higher level of abstraction the constrained behaviors that are due to the effect of the composition. This technique is therefore interesting when one agent model, say $\mathcal{Q}_1$ is a model of behaviors, while the other agent model, say $\mathcal{Q}_2$, is a performance model. The net effect is therefore that of the constraint propagation from one model to the other.

The result of applying this procedure is not to obtain a new compound object, as such an object might not be defined in either $\mathcal{Q}_1$ or $\mathcal{Q}_2$. Instead, we obtain in each domain the restricted behavior that is caused by the existence of an interaction. The particular effect of the interaction can only be understood at a lower level of abstraction that can talk about both models at the same time. Hence the composition is not only dependent upon the definition of composition at the lower level, but also on the particular process of refinement employed to derive the new model.

An alternative technique consists of considering the maximal agent q_1 in $\mathcal{Q}$ such that $q_1 \parallel p_2 \preceq p_1 \parallel p_2$. The solution for q_1 is an instance of the local specification synthesis problem, to be described in section 3.4. Intuitively, q_1 in this case represents the flexibility for implementing p_1. In other words, more behaviors, which are incompatible with p_2 and therefore are not observed in the composition, may be added, without altering the result. At the abstract level, $\Psi^1_u(q_1)$ represents the flexibility for p'_1, or its "behavioral don't cares". By doing so, it is possible to find alternative implementation for the agents in the system that may prove more optimal relative to a cost function.

2.8.3 A Hierarchy of Models

Theorem 2.119 gives us a way to compute a conservative approximation between two models starting from their relation to a third common domain. Note however that the assumptions of theorem 2.119 do not imply that the common semantic domain $\mathcal{Q}$ be necessarily a "refinement" of the other two, as figure 2.2 implies. That is, a common "abstraction" could also be employed to compute a set of derived Galois connections, and therefore a pair of conservative approximations, between two semantic domains. This provides us a way of organizing different models into a hierarchy, and of checking the consistency of the relations that exist between the various models in the hierarchy, by transitively applying the construction described in theorem 2.119.

More specifically, each application area must be equipped with the set of models of computation that best support the design methodology and the formal techniques that are most appropriate for the design and verification process. Several of these models can then be directly connected by relations of abstraction and refinement, as depicted by the solid lines shown in figure 2.4. That is, we assume that these direct connections are conservative approximations whose inverse is always defined, thus clearly establishing a containment relation in terms of the information embodied by each model. These relations also establish an order on the set of models, which becomes a lattice structure whenever a greatest (more abstract) and a least (more concrete) model exist for the specific application area.

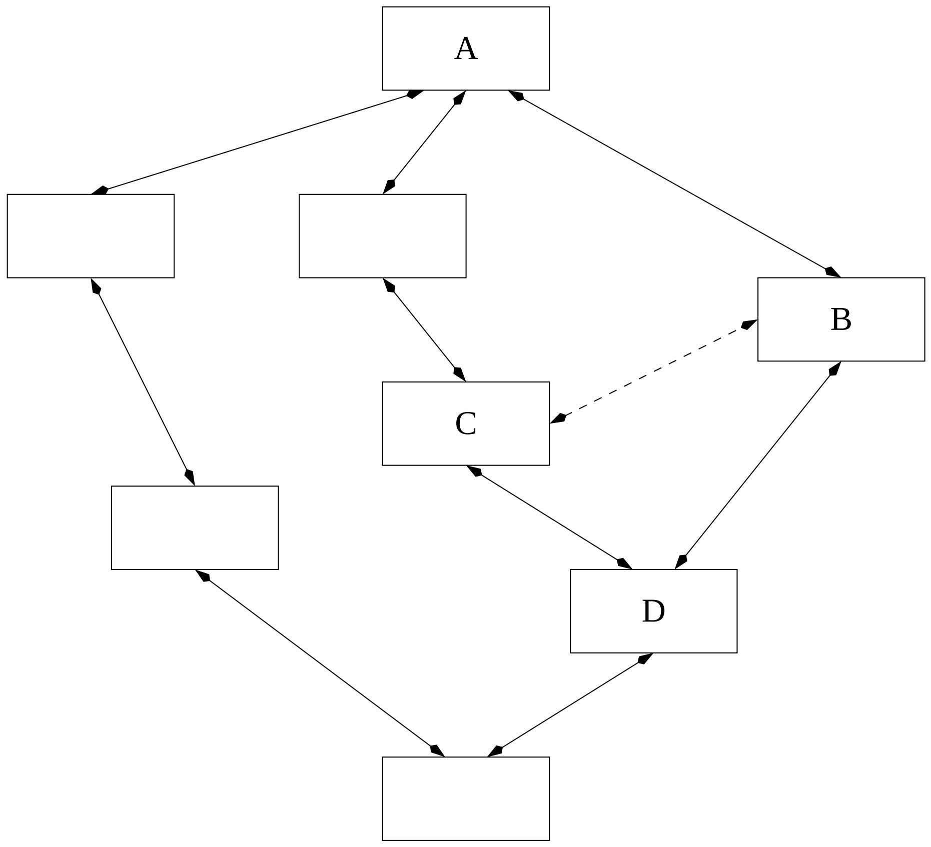

Figure 2.4: A lattice of models

The levels of abstraction of models of computation that are *not* directly related are incomparable, in the sense that each model is able to express information that the other ignores, and vice versa. Nonetheless, these models can be related by conservative approximations, whether they are established a priori by the designer, or whether they are derived from existing connections. Note also that these relations can be derived using different abstractions and refinement paths. For example, the relation between the semantic domains B and C in figure 2.4 denoted by the dashed line can be obtained by way of either the common abstraction A or the common refinement D. Since the common refinement D is able to express more information than the common abstraction A, the approximation Ψ_D derived from D will necessarily be *stronger* than the approximation Ψ_A derived from A. However, the two approximations cannot be arbitrarily different. If the existing approximations do not contradict each other, then Ψ_A must be a looser approximation than Ψ_D, in the sense of theorem 2.61. In fact, we say that a hierarchy of models is *consistent* if the conservative approximations between any pair of two models (whether the approximations are derived or not) are related by a looser or tighter relationship. A consistent hierarchy of models essentially ensures that the relations between the models interpret abstraction and refinement of the agents in the same way.

Note also that hierarchies of models are not immutable or fixed. Different application areas require different models, and therefore different hierarchies. Even when they employ the same models, there might be differences in the implementation strategies. That means that abstraction and refinement may be interpreted differently in different application areas, and therefore give rise to different hierarchies.

2.8.4 Model Translations

Consider the configuration of agent algebras depicted in figure 2.1 such that $\mathcal{Q}'$ is an abstraction of $\mathcal{Q}$ by a conservative approximation Ψ induced by a pair of Galois connections, and such that Ψ_{inv} is defined for all agents in $\mathcal{Q}'$. Since, in this case, all agents of $\mathcal{Q}'$ can be represented exactly in $\mathcal{Q}$, it is straightforward to consider the heterogeneous composition in the context of $\mathcal{Q}$.

The co-composition in the opposite direction is more problematic. Assume in particular that $p \in \mathcal{Q}.D$ is an agent such that $\Psi_u(p) \neq \Psi_l(p)$. In that case, p is not represented exactly in $\mathcal{Q}'$, or, to put it another way, p is not polymorphic relative to the chosen domains. There are different ways to get around this problem, and they mainly consist of encapsulating p using a translator that *does* make the combination polymorphic. This is, for example, the technique used in the Ptolemy II framework, where an intermediate director compatible with the agent is used to mediate the communication between the agent that is not polymorphic, and the domain in which the designer wishes to use it.

Translations in our framework take the form of closure or interior systems. We have already seen that a conservative approximation $\Psi = (\Psi_l, \Psi_u)$ determines for each agent the equivalence classes of the agents that have the same upper bound and the same lower bound, respectively. Theorem 2.59 shows that if the inverse of a conservative approximation is defined for an agent p', then $\Psi_{inv}(p')$ is at the greatest and least element, respectively, of these equivalence classes. It is easy to show when the upper and lower bound are monotonic functions, then these elements constitute a closure and an interior for the elements of their respective equivalence classes.

Theorem 2.120. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$ such that Ψ_l and Ψ_u are monotonic and Ψ_{inv} is defined for all agents p' in $\mathcal{Q}'$. Let $C, I : \mathcal{Q}.D \mapsto \mathcal{Q}.D$ be operators of $\mathcal{Q}$ defined as

$$\begin{aligned} C(p) &= \Psi_{inv}(\Psi_u(p)), \\ I(p) &= \Psi_{inv}(\Psi_l(p)). \end{aligned}$$

Then C is a closure operator, and I is an interior operator.

Proof: To prove that C is a closure operator we must show that for all agents $p, p_1, p_2 \in \mathcal{Q}.D$,

$$\begin{aligned} \text{Monotonic} &\quad p_1 \preceq p_2 \Rightarrow C(p_1) \preceq C(p_2). \\ \text{Increasing} &\quad p \preceq \Psi_{inv}(\Psi_u(p)). \\ \text{Idempotent} &\quad C(C(p)) = C(p). \end{aligned}$$

It is easy to show that C is monotonic, since Ψ_u is monotonic by hypothesis, and Ψ_{inv} is monotonic by theorem 2.58. In addition, C is increasing by theorem 2.57. Finally, for all agents p, since Ψ_u is inverse of Ψ_{inv}, $\Psi_u(\Psi_{inv}(\Psi_u(p))) = \Psi_u(p)$. Therefore C is also idempotent.

The proof that I is an interior operator is similar, and it implies showing that I is decreasing. □

The hypothesis of theorem 2.120 also imply, by theorem 2.106, that $\langle \Psi_u, \Psi_{inv} \rangle$ is a Galois connection from $\mathcal{Q}$ to $\mathcal{Q}'$ and that $\langle \Psi_{inv}, \Psi_l \rangle$ is a Galois connection from $\mathcal{Q}'$ to $\mathcal{Q}$.

The closure and the interior operator essentially "complete" an agent in order to make it compatible with the requirements of the abstract domain. The closure produces an abstraction within $\mathcal{Q}$ by choosing the greatest element of the equivalence class induced by Ψ_u, thus potentially "adding" behaviors that are required by the abstract domain. The interior, on the other hand, computes a refinement in $\mathcal{Q}$, by choosing the least element of the equivalence class induced by Ψ_l, and thus “removing” behaviors that are incompatible with the abstract domain. Other forms of completion are also possible. We do not however explore them further here, and reserve them for our future work.

2.8.5 Platform-Based Design

The framework that we have presented is useful to formally describe the process of successive refinement in a platform-based design methodology. There, refinement is interpreted as the concretization of a function in terms of the elements of an architecture. The process of design consists of evaluating the performance of different kinds of architectures by mapping the functionality onto its different elements. The implementation is then chosen on the basis of some cost function.

Both the functionality and the architecture can be represented at different levels of abstraction. For example, an architecture may employ a generic communication structure that includes point-to-point connections for all elements, and unlimited bandwidth. On a more accurate level, the communication structure may be described as a bus with a particular arbitration policy and limited bandwidth. Similarly, the functionality could be described as the interconnection of agents that communicate through either unbounded (more abstract) or bounded (more concrete) queues.

In order to characterize the process of mapping and performance evaluation, we use three distinct semantic domains. Two domains, called the *architecture platform* and the *function platform*, are devoted to describing the architecture and the function, respectively. The third, called the *semantic platform*, is an intermediate domain that is used to map the function onto an architecture.

An architecture platform, depicted in figure 2.5 on the right, is composed of a set of elements, called the *library elements*, and of *composition rules* that define the admissible topologies. In order to obtain an appropriate domain of agents to model an architecture platform we start from the set of library elements. We then construct the free algebra generated by the library elements by taking the closure under the operation of composition. In other words, we construct all the topologies that are admissible by the composition rules, and add them to the set of agents in the algebra. Thus, each agent in the architecture platform algebra, called a *platform instance*, is a particular topology that is consistent with the rules of the platform. This construction is similar to a term algebra, subject to the constraints of the composition rules. For most architecture platforms the composition must be constrained, since the number of available resources is bounded. For example an architecture platform may provide only one instance of a particular processor. In that case, topologies that employ two ore more instances are ruled out.

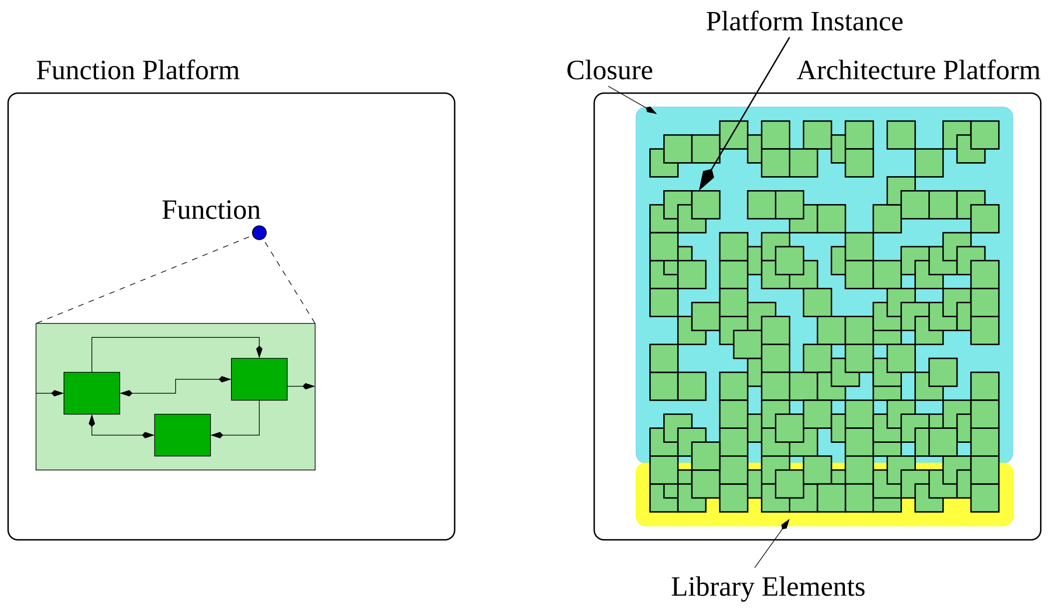

Figure 2.5: Architecture and Function Platforms

Similarly to the architecture platform, the function platform, depicted in figure 2.5 on the left, is represented as an agent algebra. Here the desired function is represented denotationally, as the collective behavior of a composition of agents. However, unlike the architecture platform which is used to select one particular instance among several, the function is fixed and is used as the specification for the refinement process.

The specification and the implementation come together in an intermediate algebra, called the *semantic platform*. The semantic platform plays the role of the common refinement $\mathcal{Q}$ of figure 2.2, and is used to combine the properties of both the architecture and the function platform. In fact, the function platform may be too abstract to talk about the performance indices that are characteristic of the more concrete architecture, while at the same time the architecture platform is a mere composition of components, without a notion of behavior. In particular, we assume that there exists a conservative approximation between the semantic platform and the function platform, and that the inverse of the conservative approximation is defined at the function that we wish to evaluate. The function therefore is mapped onto the semantic platform as shown in figure 2.6. This mapping also includes all the refinements of the function that are consistent with the performance constraints, which can be interpreted in the semantic platform.

The correspondence between the architecture and the semantic platform is more complex. A platform instance, i.e., an agent in the architecture platform, usually includes programmable

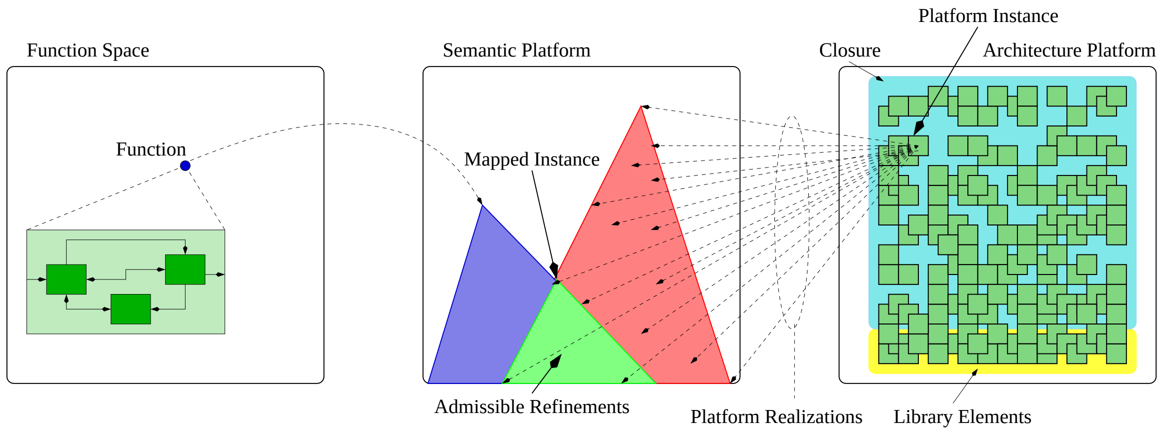

Figure 2.6: Mapping of function and architecture

elements (microprocessors, programmable logic) that may be customized for the particular function required. Therefore, each platform instance may be used to implement a variety of functions, or behaviors. Each of these functions is in turn represented as one agent in the semantic platform. A platform instance is therefore projected onto the semantic platform by considering the collection of the agents that can be implemented by the particular instance. These, too, can be organized as a refinement hierarchy, since the same function could be implemented using different algorithms and employing different resources even within a particular platform instance. Note that the projection of the platform instance onto the semantic platform, represented by the rays that originate from the architecture platform in figure 2.6, may or may not have a greatest element. If it does, the greatest element represents the non-deterministic choice of any of the functions that are implementable by the architecture.

An architecture and a function platform may be related using different semantic platforms, and under different notions of refinement. The choice of semantic platform is particularly important. The agents in the semantic platform must in fact be detailed enough to represent the performance values of interest in choosing a particular platform instance, and a particular realization (via programmability) of the instance. However, if the semantic platform is too detailed, the correspondence between the platform instance and its realizations may be impractical to compute. This correspondence is therefore usually obtained by estimation techniques, rather than by analytical methods.

The semantic platform is partitioned into four different areas. We are interested in the area that corresponds to the intersection of the refinements of the function and of the functions that are implementable by the platform instance. This area is marked “Admissible Refinements” in figure 2.6. In fact, the agents that refine the function, but do not refine the architecture, are possible implementations that are not supported by the platform instance. The agents that refine the platform instance, but not the function, are possible behaviors of the architecture that are either inconsistent with the function (they do something else), or they do not meet the performance constraints. The rest of the agents that are not in the image of any of the maps correspond to behaviors that are inconsistent with the function and are not implementable by the chosen platform instance.

Among all the possible implementations, one must be chosen as the function to be used for the next refinement step. Each of the admissible refinements encodes a particular mapping of the components of the function onto the services offered by the selected platform instance. Of all those agents, we are usually interested in the ones that are closer to the greatest element, as those implementations more likely offer the most flexibility when the same refinement process is iterated to descend to an even more concrete level of abstraction. In addition, several different platform instances may be considered to search among the different topologies and available resources and services.

Once a suitable implementation has been chosen, the process continues with the next refinement step. The new function platform is obtained as the combination of the semantic platform that provides information on the desired behavior, and the architecture platform, which provides information on the topology and the structure of the mapped implementation. The new function is then mapped to a new architecture, employing the same device of a semantic platform as an intermediate domain.

Chapter 3

Conformance, Mirrors and Local Specification Synthesis

This chapter is devote to studying the problem of refinement verification in general, and the problem of local specification synthesis in particular. The techniques that we present here are a generalization to agent algebras of the corresponding concepts introduced by Dill for asynchronous trace structures [34]. By abstracting the notion of an agent, our definitions are based on more fundamental properties of models of computation. Our results therefore provide more insights on the interplay between the ordering on the agents, the notion of compatibility and its maximal elements.

In this chapter we show that the order of an agent algebra can often be characterized in terms of substitutability as a *conformance* relation. We introduce the definition of a conformance order by considering the effect of substituting an agent for another agent in every possible context. We parameterize the notion of substitutability using a set of agents, called a *conformance set*. Conformance can be used to verify the order relationship between agents. In this chapter we also introduce the notion of the *mirror* of an agent, which, together with a conformance order, reduces the task of refinement verification to computing a parallel composition and checking membership with the conformance set. The relationship between mirrors and conservative approximations is also illustrated here.

The conformance order and the mirror function are used in agent algebras to formulate and to solve the problem of synthesizing a local specification subject to a context. This construction, which is independent of the particular agent algebra considered, is useful in developing synthesis techniques, as already discussed in section 1.3. The solution to the problem requires that the context expression be presented in a particular normal form. We show that under certain conditions every expression can be transformed into an equivalent expression in this form. The normal form is useful to deal with systems specified as complex interaction of hierarchies, as it allows flattening the system to a single parallel composition. The normal form is also important for obtaining closed form solutions to equations involving expressions on agents.

3.1 Expression Equivalence and Normal Forms

In this section we define what it means for two agent expressions to be equivalent and prove that every expression can be transformed into an equivalent expression in a specific (normal) form.

We say that two expressions are equivalent if they have the same value for all possible assignments.

Definition 3.1 (Expression Equivalence). Let $\mathcal{Q}$ be an agent algebra. Two expressions E_1 and E_2 over $\mathcal{Q}$ are equivalent, written $E_1 \equiv E_2$, if and only if for all assignments σ, $[\![E_1]\!]\sigma = [\![E_2]\!]\sigma$.

In particular the above definition implies that if two expressions are equivalent then they are defined or undefined for exactly the same assignments. Notice also that because equivalence depends on the evaluation of the expression, two expressions may be equivalent relative to one agent algebra and not equivalent relative to another agent algebra. In other words, expression equivalence depends on the particular choice of underlying agent algebra.

Sometimes it is convenient to consider only a subset of the possible assignments. In that case we talk about equivalence modulo a set of assignments Σ'.

Definition 3.2 (Expression Equivalence modulo Σ'). Let Σ' be a set of assignments. Two expressions E_1 and E_2 are equivalent modulo Σ', written $E_1 \equiv_{\Sigma'} E_2$, if and only if for all assignments $\sigma \in \Sigma'$, $[\![E_1]\!]\sigma = [\![E_2]\!]\sigma$.

We state the following results for expression equivalence only, but they extend to expression equivalence modulo Σ' in a straightforward way.

Lemma 3.3. Expression equivalence is an equivalence relation.

Because the semantics of expressions is syntax directed, the value of an expression depends only on the value of its subexpressions. Hence

Theorem 3.4. Expression equivalence is a congruence with respect to the operators of the agent algebra.

Proof: We show that if E_1 and E_2 are two agent expressions such that $E_1 \equiv E_2$, then for all alphabets B, $\mathit{proj}(B)(E_1) \equiv \mathit{proj}(B)(E_2)$. The cases for *rename* and $\|$ are similar. The proof consists of the following series of implications:

$E_1 \equiv E_2$

by definition 3.1

$\Leftrightarrow$ for all assignments σ, $[\![E_1]\!]\sigma = [\![E_2]\!]\sigma$

$\Rightarrow$ for all assignments σ, $\mathit{proj}(B)([\![E_1]\!]\sigma) = \mathit{proj}(B)([\![E_2]\!]\sigma)$

by definition 2.46

$\Leftrightarrow$ for all assignments σ, $[\![\mathit{proj}(B)(E_1)]\!]\sigma = [\![\mathit{proj}(B)(E_2)]\!]\sigma$

by definition 3.1

$\Leftrightarrow$ $\mathit{proj}(B)(E_1) \equiv \mathit{proj}(B)(E_2)$

□

Lemma 3.5. Let E be an agent expression and let $\hat{E}$ be a subexpression of E. If $\hat{E} \equiv \hat{E}'$ for some $\hat{E}'$, then $E \equiv E'$, where E' is obtained from E by replacing $\hat{E}$ with $\hat{E}'$.

Proof: The proof is by induction on the structure of E. □

Equivalence is useful when we need to transform an expression into a form that is convenient for certain applications. In that case, we want to make sure that the transformations do not change the meaning (the semantics) of the expression. In this work we are particularly interested in a form where rename operator is applied first, then followed by the parallel composition operator, and finally by the projection operator. We call this the RCP normal form.

Definition 3.6 (RCP Normal Form). Let $\mathcal{Q}$ be an agent algebra and let $\mathcal{E}_0$ be the set of expressions:

$$\mathcal{E}_0 = \{ p : p \in \mathcal{Q}.D \} \cup \{ v : v \in V \}.$$

An agent expression E is said to be in RCP (i.e., rename, compose, project) normal form if it is of the form

$$E = \mathit{proj}(A)(\mathit{rename}(r_1)(E_1) \parallel \cdots \parallel \mathit{rename}(r_n)(E_n))$$

where A is an alphabet, $r_1, \ldots, r_n$ are renaming functions and $E_1, \ldots, E_n$ are expressions in $\mathcal{E}_0$.

The RCP normal form is similar to the normal form Dill defined for circuit algebra expressions [33]. In our case, however, we have extended the definition to expressions involving variables. This normal form corresponds to flattening the hierarchy: all agents are first instantiated using the rename operator, and are subsequently composed in parallel to form the entire system. The final projection is used to hide the internal signals.

Other normal forms are also possible. In the rest of this section we will however concentrate on the RCP normal form, since we will need it to solve inequalities for variables in the application shown in section 3.4. In particular, we are interested in sufficient conditions that an algebra must satisfy in order for all expressions to have an equivalent RCP normal form. We will approach this problem in steps of increasing complexity. First we will consider expressions that do not involve variables, i.e., closed expressions. In that case, the expression is either defined or undefined, a condition that greatly simplifies the search for the normal form. As a second step, we will consider expressions where variables can only be assigned agents with a specific alphabet and that always make the expression defined or not defined. This is a case that is interesting in practice, and that does not require the stronger conditions of the general result. Finally we will explore a set of restrictions that are needed to obtain an equivalent normal form in the general case. We will see that alphabets again play a major role, and that they must be restricted in order for the appropriate renaming functions and projection operators to exist. All of this is formalized in the following definitions and results.

Definition 3.7 (Closed-Normalizable Agent Algebra). Let $\mathcal{Q}$ be an agent algebra. We say that $\mathcal{Q}$ is a *closed-normalizable* agent algebra if the renaming, projection parallel composition operators satisfy the axioms given below, where p and p' are elements of D and $A = \alpha(p)$ and $A' = \alpha(p')$.

A8. If $\mathit{rename}(r)(p)$ is defined, then it is equal to $\mathit{rename}(r\,|_{A \to r(A)})(p)$.

A9. $\mathit{rename}(r')(\mathit{rename}(r)(p)) = \mathit{rename}(r' \circ r)(p)$, if the left hand side of the equation is defined.

A10. If $\mathit{proj}(B)(p)$ is defined, then it is equal to $\mathit{proj}(B \cap A)(p)$.

A11. $\mathit{proj}(B)(\mathit{proj}(B')(p)) = \mathit{proj}(B \cap B')(p)$, if the left hand side of the equation is defined.

A12. If $\mathit{rename}(r)(\mathit{proj}(B)(p))$ is defined, then there exists a function r' such that

$$\mathit{rename}(r)(\mathit{proj}(B)(p)) = \mathit{proj}(r'(B))(\mathit{rename}(r')(p)).$$

A13. If $\mathit{proj}(B)(p)$ is defined, then there exists a function r such that $r(A) \cap A' \subseteq B$ and

$$\mathit{proj}(B)(p) = \mathit{proj}(B)(\mathit{rename}(r)(p)).$$

A14. $\mathit{rename}(r)(p \,\|\, p') = \mathit{rename}(r\,|_{A \to r(A)})(p) \,\|\, \mathit{rename}(r\,|_{A' \to r(A')})(p')$, if the left hand side of the equation is defined.

A15. $\mathit{proj}(B)(p \,\|\, p') = \mathit{proj}(B \cap A)(p) \,\|\, \mathit{proj}(B \cap A')(p')$, if $(A \cap A') \subseteq B$.

The axioms formalize certain assumptions regarding the semantic domain. In particular they formalize the intuition that the renaming operator should only depend on the value of the renaming function for the signals actually used by the argument, and that consecutive applications of the renaming operator are equivalent to a single application with the appropriate renaming function. Similar consideration apply for projection. A12 states that rename and projection commute when the retained set and the renaming function are changed appropriately. Also, A14 states that rename commutes with parallel composition, and A15 states a similar property for projection. Note that projection commutes with parallel composition only if the common signals between the agents being composed are retained after the composition. This is necessary, or else the expression on the right hand side of the equation would lack the necessary information to compute the full synchronization between the agents. Finally, A13 asserts that it is possible to arbitrarily rename the signals of an agent that are not retained in a projection. This is essential to avoid conflicts of names when applying A15 from right to left, in order to make the "local" signals of each of the agents unique.

The axioms can be used to algebraically transform an expression into an equivalent RCP normal form, as the next result shows. Technically, since we are considering only sufficient and not necessary conditions, the term *normalizable* should apply to all agent algebras whose expression can be put in RCP normal form, whether or not they satisfy the axioms. In practice, we restrict our attention to only algebras that do satisfy the axioms for the purpose of normalization, and we therefore use the term to distinguish them from those that do not. We will continue to use this convention for the rest of this document, including the more general cases of normalizable agent algebras.

Theorem 3.8 (Normal Form - Closed Expressions). Let $\mathcal{Q}$ be a closed-normalizable agent algebra, and let E be a closed expression over $\mathcal{Q}$. Then E is equivalent to an expression in RCP normal form.

Proof: Let E be a closed expression. If E is undefined, then E is equivalent to any undefined closed expression in RCP normal form. If E is defined, then we construct an equivalent closed expression in RCP normal form by induction on the structure of expressions.

- Assume $E = p$ for some agent $p \in \mathcal{Q}.D$. Then $E = \mathit{proj}(A)(\mathit{rename}(\mathit{id}_A)(p))$ by A4 and A2.

- Assume $E = \mathit{proj}(B)(E_1)$. Then, by induction, E_1 is equivalent to an expression $E_1' = \mathit{proj}(B')(\mathit{rename}(r_1)(p_1) \parallel \cdots \parallel \mathit{rename}(r_n)(p_n))$ in RCP normal form. Then

$$\begin{aligned} E &= \mathit{proj}(B)(\mathit{proj}(B')(\mathit{rename}(r_1)(p_1) \parallel \cdots \parallel \mathit{rename}(r_n)(p_n))) \\ &\quad \text{By A11} \\ &= \mathit{proj}(B \cap B')(\mathit{rename}(r_1)(p_1) \parallel \cdots \parallel \mathit{rename}(r_n)(p_n)) \end{aligned}$$

  which is in RCP normal form.

- Assume $E = \mathit{rename}(r)(E_1)$. Then, by induction, E_1 is equivalent to an expression $E_1' = \mathit{proj}(B)(\mathit{rename}(r_1)(p_1) \parallel \cdots \parallel \mathit{rename}(r_n)(p_n))$ in RCP normal form. Then

$$\begin{aligned} E &= \mathit{rename}(r)(\mathit{proj}(B)(\mathit{rename}(r_1)(p_1) \parallel \cdots \parallel \mathit{rename}(r_n)(p_n))) \\ &\quad \text{By A12 there exists a renaming function } r' \text{ such that} \\ &= \mathit{proj}(r'(B))(\mathit{rename}(r')(\mathit{rename}(r_1)(p_1) \parallel \cdots \parallel \mathit{rename}(r_n)(p_n))) \\ &\quad \text{By A6, A14 and A3} \\ &= \mathit{proj}(r'(B))(\mathit{rename}(r'|_{r_1(A_1)\to r'(r_1(A_1))})(\mathit{rename}(r_1)(p_1)) \parallel \cdots \\ &\qquad \parallel \mathit{rename}(r'|_{r_n(A_n)\to r'(r_n(A_n))})(\mathit{rename}(r_n)(p_n))) \\ &\quad \text{By A9} \\ &= \mathit{proj}(r'(B))(\mathit{rename}(r'|_{r_1(A_1)\to r'(r_1(A_1))} \circ r_1)(p_1) \parallel \cdots \\ &\qquad \parallel \mathit{rename}(r'|_{r_n(A_n)\to r'(r_n(A_n))} \circ r_n)(p_n)) \end{aligned}$$

  which is in RCP normal form.

- Assume $E = E_1 \parallel E_2$. Then, by induction, E_1 is equivalent to an RCP normal form $E_1' = \mathit{proj}(B_1)(\mathit{rename}(r_{11})(p_{11}) \parallel \cdots \parallel \mathit{rename}(r_{n1})(p_{n1}))$ and E_2 is equivalent to an RCP normal form $E_2' = \mathit{proj}(B_2)(\mathit{rename}(r_{12})(p_{12}) \parallel \cdots \parallel \mathit{rename}(r_{n2})(p_{n2}))$. Let A_1 be the alphabet of expression E_1'' such that $E_1' = \mathit{proj}(B_1)(E_1'')$. We can assume, without loss of generality, that $B_1 \subseteq A_1$ and $B_2 \subseteq A_2$. By A13 there exists a function r_1 such that $r_1(A_1) \cap A_2 \subseteq B_1$ and

$$\mathit{proj}(B_1)(E_1'') = \mathit{proj}(B_1)(\mathit{rename}(r_1)(E_1'')).$$

Similarly, there exists a function r_2 such that $r_2(A_2) \cap (A_1 \cup r_1(A_1)) \subseteq B_2$ and

$$\mathit{proj}(B_2)(E_2'') = \mathit{proj}(B_2)(\mathit{rename}(r_2)(E_2'')).$$

By A10

$$\begin{aligned}\mathit{proj}(B_1)(E_1'') &= \mathit{proj}(B_1 \cap r_1(A_1))(\mathit{rename}(r_1)(E_1''))\\ \mathit{proj}(B_2)(E_2'') &= \mathit{proj}(B_2 \cap r_2(A_2))(\mathit{rename}(r_2)(E_2''))\end{aligned}$$

Note that since $r_1(A_1) \cap A_2 \subseteq B_1$, and since $B_2 \subseteq A_2$, also $r_1(A_1) \cap B_2 \subseteq B_1$. Thus also $r_1(A_1) \cap B_2 \subseteq B_1 \cap r_1(A_1)$. Hence

$$(B_1 \cup B_2) \cap r_1(A_1) = (B_1 \cap r_1(A_1)) \cup (B_2 \cap r_1(A_1)) = B_1 \cap r_1(A_1).$$

Likewise, since $r_2(A_2) \cap (A_1 \cup r_1(A_1)) \subseteq B_2$, also $r_2(A_2) \cap A_1 \subseteq B_2$ and therefore $r_2(A_2) \cap B_1 \subseteq B_2$. Thus $r_2(A_2) \cap B_1 \subseteq B_2 \cap r_2(A_2)$. Hence

$$(B_1 \cup B_2) \cap r_2(A_2) = (B_1 \cap r_2(A_2)) \cup (B_2 \cap r_2(A_2)) = B_2 \cap r_2(A_2).$$

Thus we have

$$\begin{aligned}\mathit{proj}(B_1)(E_1'') &= \mathit{proj}((B_1 \cup B_2) \cap r_1(A_1))(\mathit{rename}(r_1)(E_1''))\\ \mathit{proj}(B_2)(E_2'') &= \mathit{proj}((B_1 \cup B_2) \cap r_2(A_2))(\mathit{rename}(r_2)(E_2''))\end{aligned}$$

Moreover, since $r_2(A_2) \cap (A_1 \cup r_1(A_1)) \subseteq B_2$, we also have $r_2(A_2) \cap r_1(A_1) \subseteq B_2$, so that $r_2(A_2) \cap r_1(A_1) \subseteq B_1 \cup B_2$. Hence by A15

$$\begin{aligned}&\mathit{proj}(B_1)(E_1'') \parallel \mathit{proj}(B_2)(E_2'') =\\ &\quad= \mathit{proj}((B_1 \cup B_2) \cap r_1(A_1))(\mathit{rename}(r_1)(E_1'')) \parallel\\ &\qquad \mathit{proj}((B_1 \cup B_2) \cap r_2(A_2))(\mathit{rename}(r_2)(E_2''))\\ &\quad= \mathit{proj}(B_1 \cup B_2)(\mathit{rename}(r_1)(E_1'') \parallel \mathit{rename}(r_2)(E_2''))\end{aligned}$$

By A9 and A6

$$\mathit{rename}(r_1)(E_1'') = \mathit{rename}(r_1 \circ r_{11})(p_{11}) \parallel \cdots \parallel \mathit{rename}(r_1 \circ r_{n1}(p_{n1})$$
$$\mathit{rename}(r_2)(E_2'') = \mathit{rename}(r_2 \circ r_{12})(p_{12}) \parallel \cdots \parallel \mathit{rename}(r_2 \circ r_{n2}(p_{n2})$$

which proves the result.

□

If we consider an expression that involves variables, the axioms of agent algebras and the axioms of definition 3.7 may not be sufficient to ensure the existence of an equivalent normal form. Consider, for example, the expression

$$E = v.$$

We must find a renaming function r and an alphabet B such that

$$E \equiv \mathit{proj}(B)(\mathit{rename}(r)(v)).$$

The axioms are insufficient for two reasons. In the first place, A4 and A2 ensure the existence of an appropriate renaming function r and alphabet B for each agent. However, the algebra must be such that *the same* renaming function r and alphabet B can be used to construct an equivalent expression for all agents (or, at least, for the subset of agents that are assigned to v). The same is true of all the axioms that for all agents dictate the existence of a certain renaming function or alphabet. To make the algebra normalizable, the order of the quantifiers of these axioms must be exchanged, thus strengthening the requirements.

Secondly, we have dealt with the problem of definedness in theorem 3.8 by deriving a different normal form, according to whether the original closed expression is defined or not. However, unlike a closed expression, an expression may be defined or not defined depending on the assignment to its variables. To ensure equivalence, we must find an expression in normal form that is defined and not defined for exactly the same assignments. In the particular case above, since the expression $E = v$ is defined for all possible assignments to v, we must find a renaming function r and an alphabet B such that $\mathit{proj}(B)(\mathit{rename}(r)(v))$ is also always defined. Consequently, we must strengthen the axioms in two ways: by first requiring that the equalities that occur in the axioms are valid whether or not the left hand side is defined; and by introducing additional assumptions on the definedness of the operators to ensure the existence of the normal form.

However, one case that requires minimal strengthening of the axioms, and that is of great practical interest, is when variables are always assigned agents with the same alphabet, and such that the expression is always defined or always not defined. This is, for instance, the case in [33].

Definition 3.9 (Alpha-Normalizable Agent Algebra). Let $\mathcal{Q}$ be a normalizable agent algebra. We say that $\mathcal{Q}$ is *alpha-normalizable* if the renaming, projection and parallel composition operators satisfy the following axioms:

A16. For all alphabets A there exists a renaming function r' such that for all agents p such that $\alpha(p) = A$, if $\mathit{rename}(r)(\mathit{proj}(B)(p))$ is defined, then

$$\mathit{rename}(r)(\mathit{proj}(B)(p)) = \mathit{proj}(r'(B))(\mathit{rename}(r')(p)).$$

A17. For all alphabets A there exists a renaming function r such that for all agents p such that $\alpha(p) = A$, if $\mathit{proj}(B)(p)$ is defined, then $r(A) \cap A' \subseteq B$ and

$$\mathit{proj}(B)(p) = \mathit{proj}(B)(\mathit{rename}(r)(p)).$$

Note that, as in definition 3.7, the axioms are stated directly in terms of the operators and the agents of the algebra. However, by theorem 3.4, they can be used with expressions whenever every evaluation (possibly restricted to a set of assignments Σ') of the expressions involved satisfies the requirements of the axiom. In that case, the equality must be replaced by equivalence (possibly modulo Σ'). This remark applies especially to the proofs of theorem 3.10 and theorem 3.16 below.

Theorem 3.10 (Normal Form - Same Alphabet). Let $\mathcal{Q}$ be an alpha-normalizable agent algebra. Let E be an expression over $\mathcal{Q}$ and let Σ' be a set of assignments such that for all $\sigma_1, \sigma_2 \in \Sigma'$, $[\![E]\!]\sigma_1\downarrow$ if and only if $[\![E]\!]\sigma_2\downarrow$ and for all variables v, $\alpha(\sigma_1(v)) = \alpha(\sigma_2(v))$. Then E is equivalent modulo Σ' to an expression E' in RCP normal form.

Proof: The proof is similar to the proof of theorem 3.8. In fact, the transformations in the induction are the same and equally valid for every assignment (subject to the restrictions set forth in the statement of the theorem) and therefore preserve the evaluation of the expression no matter what agents replaces the variables. □

In general, an equivalent RCP normal form for an expression that involves unrestricted variables and quantities does not exist. To see why, consider the following expression E:

$$E = \mathit{proj}(B)(p) \parallel v.$$

To normalize this expression we must rename p so that the signals that are in its alphabet and that are not in B do not conflict with the signals in v. The alphabet of v however depends on its assigned value. Thus, if we assume that for each signal in the master alphabet $\mathcal{Q}.\mathcal{A}$ there is an agent that has that signal in its alphabet, then there exists no renaming function with the above property. One could avoid conflicts by renaming v by folding the master alphabet into a subset of itself (this can be done only if the master alphabet is infinite), thus making the extra signals available for p. However, in general this changes the meaning of the expression (since v now appears renamed without being guarded by a projection), thus making it difficult to obtain an equivalent expression.

One could of course require that projection and parallel composition always commute. That, however, would not only unduly restrict the kinds of models of computation that can be studied as agent algebras, but, more importantly, would be contrary to the intuitive interpretation of the operations. Therefore, in the absence of conditions specific to particular agent algebras, we must restrict the extent of the alphabets that are used in the expression and in the assignments to the variables.

In the rest of this section we present sufficient conditions for the existence of an equivalent RCP normal form for expressions involving variables. In particular, we are looking for restrictions on the alphabet of agents while still maintaining full generality. This can be achieved by restricting the use of the master alphabet to only a subset of the available signals, as long as the subset has the same cardinality as the whole and still leaves enough signals available for the operations of renaming. As a consequence, the equivalence will be modulo some set of assignments Σ' that satisfies the restrictions.

In what follows we will make use of the following lemmas and definitions.

Lemma 3.11. Let $\mathcal{Q}$ be an agent algebra. Let E_1 and E_2 be two expression over $\mathcal{Q}$ and Σ' a set of assignments such that $E_1 \equiv_{\Sigma'} E_2$. Then $\alpha([\![E_1]\!]\Sigma') = \alpha([\![E_2]\!]\Sigma')$.

Proof: Let $a \in \alpha([\![E_1]\!]\Sigma')$. Then there exists an assignment $\sigma \in \Sigma'$ such that $[\![E_1]\!]\sigma = p$ and $a \in \alpha(p)$. But since $E_1 \equiv_{\Sigma'} E_2$, then also $[\![E_2]\!]\sigma = p$. Therefore $a \in \alpha([\![E_2]\!]\Sigma')$. The reverse direction is similar. □

Definition 3.12 (Small Subset). Let W be a set and let B be a subset of W. We say that B is a small subset of W, written $B \Subset W$, if:

- W is infinite.

- The cardinality of the complement $W - B$ is greater than or equal to the cardinality of B.

Lemma 3.13. Let X and Z be sets such that $X \Subset Z$. Then there exists a set Y such that $X \Subset Y \Subset Z$.

Proof: Let Y_1 and Y_2 be two subsets of $Z - X$ of the same size such that $Z - X = Y_1 \cup Y_2$, and let $Y = X \cup Y_1$. Since $Z - X$ is infinite, $|Y_1| = |Y_2| = |Z - X|$. Since $|Z - X| \geq |X|$, also $|Y_1| \geq |X|$. Therefore $X \Subset Y$.

Since $X \Subset Y$, $|X| \leq |Y_1|$. Therefore, since $Y = X \cup Y_1$, $|Y| = |Y_1|$. Therefore also $|Y_2| = |Y|$. Hence $Y \Subset Y \cup Y_2 = Z$. □

We now have the vocabulary to state and prove the main result of this section.

Definition 3.14 (Normalizable Agent Algebra). Let $\mathcal{Q}$ be an agent algebra. We say that $\mathcal{Q}$ is a *normalizable* agent algebra if the renaming, projection and parallel composition operators satisfy the following axioms:

A18. For all alphabets A there exists an alphabet B such that $A \subseteq B$ and for all agents p such that $\alpha(p) \subseteq A$

$$p = \mathit{rename}(\mathit{id}_B)(p).$$

A19. For all alphabets B and agents p and for all alphabets A' such that $\alpha(p) \cap A' = \emptyset$

$$\mathit{proj}(B)(p) = \mathit{proj}(B \cup A')(p).$$

A20. For all alphabets B and B', and for all agents p

$$\mathit{proj}(B)(\mathit{proj}(B')(p)) = \mathit{proj}(B \cap B')(p).$$

A21. For all renaming functions r_1 and r_2 and for all agents p

$$\mathit{rename}(r_1)(\mathit{rename}(r_2)(p)) = \mathit{rename}(r_1 \circ r_2)(p)$$

where for all signals a,

$$(r_1 \circ r_2)(a) = \begin{cases} r_1(r_2(a)) & \text{if } r_1(r_2(a))\downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

A22. For all renaming functions r and for all alphabets B there exist renaming functions r' and r'' such that for all agents p

$$\mathit{rename}(r)(\mathit{proj}(B)(p)) = \mathit{proj}(r'(B))(\mathit{rename}(r'')(p)).$$

A23. For all alphabets B, for all alphabets A and for all alphabets A' such that $|(\mathcal{Q}.\mathcal{A} - A') - B| \geq |A - B|$ there exists a renaming function r such that $r(A) \cap A' \subseteq B$ and for all agents p such that $\alpha(p) \subseteq A$

$$\mathit{proj}(B)(p) = \mathit{proj}(B)(\mathit{rename}(r)(p)).$$

A24. For all renaming functions r and for all agents p_1 and p_2

$$\mathit{rename}(r)(p_1 \parallel p_2) = \mathit{rename}(r)(p_1) \parallel \mathit{rename}(r)(p_2).$$

A25. For all alphabets B and for all agents p_1 and p_2 such that $\alpha(p_1) \cap \alpha(p_2) \subseteq B$

$$\mathit{proj}(B)(p_1 \parallel p_2) = \mathit{proj}(B)(p_1) \parallel \mathit{proj}(B)(p_2).$$

The axioms of normalizable agent algebras are essentially equivalent to those of definition 3.7 for closed-normalizable algebras. They differ in the way alphabets and variables are handled.

It is easy to show that A19 is equivalent to a similar form that involves restricting the retained alphabet, rather than extending it.

Lemma 3.15. Let $\mathcal{Q}$ be a normalizable agent algebra. Then the following two statements are equivalent.

1. $\mathcal{Q}$ satisfies A19, i.e., for all alphabets B and agents p and for all alphabets A' such that $\alpha(p) \cap A' = \emptyset$

$$\mathit{proj}(B)(p) = \mathit{proj}(B \cup A')(p).$$

2. for all alphabets B and agents p and for all alphabets A' such that $\alpha(p) \subseteq A'$

$$\mathit{proj}(B)(p) = \mathit{proj}(B \cap A')(p).$$

Proof: For the forward implication, assume item 1 is true. Let B be an alphabet, p an agent and A' an alphabet such that $\alpha(p) \subseteq A'$. Let $K = B \cap A'$ and $J' = B - A'$. Then

$$\alpha(p) \subseteq A'$$

since $J' \cap A' = \emptyset$

$$\Leftrightarrow \quad \alpha(p) \cap J' = \emptyset$$

by item 1

$$\Rightarrow \quad \mathit{proj}(K)(p) = \mathit{proj}(K \cup J')(p)$$

$$\Leftrightarrow \quad \mathit{proj}(B \cap A')(p) = \mathit{proj}((B \cap A') \cup (B - A'))(p)$$

$$\Leftrightarrow \quad \mathit{proj}(B \cap A')(p) = \mathit{proj}(B)(p)$$

For the reverse implication, assume item 2 is true. Let B be an alphabet, p an agent and A' an alphabet such that $\alpha(p) \cap A' = \emptyset$. Let $K = B \cup A'$ and $J' = \alpha(p) \cup B$. Then

$$\alpha(p) \subseteq J'$$

by item 2

$$\Rightarrow \quad \mathit{proj}(K)(p) = \mathit{proj}(K \cap J')(p)$$

$$\Leftrightarrow \quad \mathit{proj}(B \cup A')(p) = \mathit{proj}((B \cup A') \cap (\alpha(p) \cup B))(p)$$

$$\Leftrightarrow \quad \mathit{proj}(B \cup A')(p) = \mathit{proj}((B \cap \alpha(p)) \cup (B \cap B) \cup (A' \cap \alpha(p)) \cup (A' \cap B))(p)$$

since $B \cap \alpha(p) \subseteq B$, $A' \cap \alpha(p) = \emptyset$ and $A' \cap B \subseteq B$

$$\Leftrightarrow \quad \mathit{proj}(B \cup A')(p) = \mathit{proj}(B)(p)$$

□

The following theorem shows that in a normalizable agent algebra any expression can be turned into an equivalent expression in RCP normal form when enough signals are available. The notation is simpler if we assume that the expression does not contain constants. This assumption is without loss of generality, since the case when an expression contains constants can be obtained by representing the constants with unique variables and by considering only assignments that assign the corresponding constant to the variables.

Theorem 3.16 (Normal Form). Let $\mathcal{Q}$ be an agent algebra such that $\mathcal{Q}.\mathcal{A}$ is infinite, and let E be an expression over $\mathcal{Q}$ that does not involve constants. Let Σ' be a set of assignments and $W \subseteq \mathcal{Q}.\mathcal{A}$ be an alphabet such that $\alpha(\llbracket\, \mathit{sub}(E) \,\rrbracket\Sigma') \Subset W$. Then E is equivalent modulo Σ'

to an expression E' in RCP normal form such that $\alpha(\llbracket \mathit{sub}(E') \rrbracket \Sigma') \subseteq W$. In addition, if a variable v appears k times in E, then it appears k times in E'.

Proof: The proof uses the following result.

Lemma 3.17. Let $E = \mathit{rename}(r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r_n)(v_n)$ be an expression such that $\alpha(\llbracket E \rrbracket \Sigma') = A$. Then

$$\mathit{rename}(r)(E) \equiv_{\Sigma'} \mathit{rename}(r \circ r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r \circ r_n)(v_n)$$

and both $\alpha(\llbracket \mathit{rename}(r)(E) \rrbracket \Sigma') = r(A)$ and for all i, $\alpha(\llbracket \mathit{rename}(r \circ r_i)(v_i) \rrbracket \Sigma') \subseteq r(A)$.

Proof: By A6 and A24

$$\mathit{rename}(r)(E) \equiv_{\Sigma'} \mathit{rename}(r)(\mathit{rename}(r_1)(v_1)) \parallel \cdots \parallel \mathit{rename}(r)(\mathit{rename}(r_n)(v_n)),$$

and by A21

$$\mathit{rename}(r)(E) \equiv_{\Sigma'} \mathit{rename}(r \circ r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r \circ r_n)(v_n).$$

Since $\alpha(\llbracket E \rrbracket \Sigma') = A$, and by A5, for all i, $\alpha(\llbracket \mathit{rename}(r_i)(v_i) \rrbracket \Sigma') \subseteq A$. Therefore by A3, $\alpha(\llbracket \mathit{rename}(r)(E) \rrbracket \Sigma') = r(A)$ and for all i, $\alpha(\llbracket \mathit{rename}(r \circ r_i)(v_i) \rrbracket \Sigma') \subseteq r(A)$. □

The proof is by induction on the structure of expressions.

- Let $E = v$. Let $A = \alpha(\llbracket E \rrbracket \Sigma')$. By A18 there exists an alphabet B such that $A \subseteq B$ and for all p such that $\alpha(p) \subseteq A$

$$p = \mathit{rename}(\mathit{id}_B)(p).$$

Let now $\sigma \in \Sigma'$ be an assignment. Then by definition 2.46

$$\begin{aligned}
[\![v]\!]\sigma &= \sigma(v) \\
&\quad \text{Since } \alpha(\sigma(v)) \subseteq A \\
&= \mathit{rename}(\mathit{id}_B)(\sigma(v)) \\
&\quad \text{by A2, since } \alpha(\sigma(v)) = \alpha(\mathit{rename}(\mathit{id}_B)(\sigma(v))) \\
&= \mathit{proj}(\alpha(\sigma(v)))(\mathit{rename}(\mathit{id}_B)(\sigma(v))) \\
&\quad \text{by A19, since } \alpha(\sigma(v)) \cap A \subseteq \alpha(\sigma(v)) \\
&= \mathit{proj}(A)(\mathit{rename}(\mathit{id}_B)(\sigma(v))) \\
&\quad \text{by definition 2.46} \\
&= [\![\mathit{proj}(A)(\mathit{rename}(\mathit{id}_B)(v))]\!]\sigma.
\end{aligned}$$

Thus by definition 3.2

$$v \equiv_{\Sigma'} \mathit{proj}(A)(\mathit{rename}(\mathit{id}_B)(v)) = E'$$

which is in RCP normal form.

By inspection

$$\mathit{sub}(E') = \{ v, \mathit{rename}(\mathit{id}_B)(v), \mathit{proj}(A)(\mathit{rename}(\mathit{id}_B)(v)) \}.$$

By hypothesis,

$$\alpha([\![v]\!]\Sigma') = A \subseteq W.$$

Since $v \equiv_{\Sigma'} \mathit{rename}(\mathit{id}_B)(v)$, by lemma 3.11

$$\alpha([\![\mathit{rename}(\mathit{id}_B)(v)]\!]\Sigma') = A \subseteq W.$$

Since $v \equiv_{\Sigma'} \mathit{proj}(A)(\mathit{rename}(\mathit{id}_B)(v))$, by lemma 3.11

$$\alpha([\![\mathit{proj}(A)(\mathit{rename}(\mathit{id}_B)(v))]\!]\Sigma') = A \subseteq W.$$

Therefore, $\alpha([\![\mathit{sub}(E')]\!]\Sigma') \subseteq W$.

By inspection, if a variable v appears k times in E, it appears k times in the normal form.

- Let $E = \mathit{proj}(B)(E_1)$. By hypothesis, $\alpha([\![\, \mathit{sub}(E) \,]\!]\Sigma') \Subset W$. Then, since $\mathit{sub}(E_1) \subseteq \mathit{sub}(E)$, also $\alpha([\![\, \mathit{sub}(E_1) \,]\!]\Sigma') \Subset W$. Then, by induction hypothesis, E_1 is equivalent to an expression E_1' in RCP normal form

  $$E_1' = \mathit{proj}(B')(\mathit{rename}(r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r_n)(v_n))$$

  and $\alpha([\![\, \mathit{sub}(E_1') \,]\!]\Sigma') \subseteq W$. Then by theorem 3.4

  $$E \equiv_{\Sigma'} \mathit{proj}(B)(\mathit{proj}(B')(\mathit{rename}(r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r_n)(v_n))).$$

  By A20, E is equivalent modulo Σ' to an expression E'

  $$E \equiv_{\Sigma'} E' = \mathit{proj}(B \cap B')(\mathit{rename}(r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r_n)(v_n))$$

  which is in RCP normal form.

  Let

  $$E_1'' = \mathit{rename}(r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r_n)(v_n).$$

  Then

  $$\mathit{sub}(E') = \{\, E' \} \cup \mathit{sub}(E_1'').$$

  Since by hypothesis $\alpha([\![\, E \,]\!]\Sigma') \subseteq W$, and since $E \equiv_{\Sigma'} E'$, by lemma 3.11

  $$\alpha([\![\, E' \,]\!]\Sigma') \subseteq W.$$

  Since $\mathit{sub}(E_1'') \subseteq \mathit{sub}(E_1')$, and since $\alpha([\![\, E_1' \,]\!]\Sigma') \subseteq W$,

  $$\alpha([\![\, \mathit{sub}(E_1'') \,]\!]\Sigma') \subseteq W.$$

  Therefore, $\alpha([\![\, \mathit{sub}(E') \,]\!]\Sigma') \subseteq W$.

  In addition, if a variable v appears k times in E, it appears k times in E_1, and therefore, by induction, it appears k times in E_1' and in the final normal form.

- Let $E = \mathit{rename}(r)(E_1)$. By hypothesis, $\alpha([\![\, \mathit{sub}(E) \,]\!]\Sigma') \Subset W$. Then, since clearly $\mathit{sub}(E_1) \subseteq \mathit{sub}(E)$, also $\alpha([\![\, \mathit{sub}(E_1) \,]\!]\Sigma') \Subset W$. Then by induction E_1 is equivalent to an expression E_1' in RCP normal form

  $$E_1' = \mathit{proj}(B')(\mathit{rename}(r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r_n)(v_n))$$

and $\alpha(\llbracket\, \mathit{sub}(E_1')\,\rrbracket\Sigma') \subseteq W$.

Let $E_1'' = \mathit{rename}(r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r_n)(v_n)$. Then, by theorem 3.4

$$E \equiv_{\Sigma'} \mathit{rename}(r)(\mathit{proj}(B')(E_1'')).$$

By A22 there exist renaming function r' and r'' such that

$$E \equiv_{\Sigma'} \mathit{proj}(r'(B'))(\mathit{rename}(r'')(E_1'')).$$

Let now $A = \alpha(\llbracket\, \mathit{rename}(r'')(E_1'')\,\rrbracket\Sigma')$ and $B = r'(B') \cap A$. Since $\alpha(\llbracket\, E\,\rrbracket\Sigma') \Subset W$, by A1 and lemma 3.11 also $B \Subset W$. By lemma 3.15

$$E \equiv_{\Sigma'} \mathit{proj}(B)(\mathit{rename}(r'')(E_1'')).$$

Let now $A' = \mathcal{Q}.\mathcal{A} - W$. Then $(\mathcal{Q}.\mathcal{A} - A') - B = W - B$. Note that r'' is a bijection, and for any assignment $\sigma \in \Sigma'$, if $\llbracket\, \mathit{rename}(r'')(E_1'')\,\rrbracket\sigma$ is defined then also $\llbracket\, E_1''\,\rrbracket\sigma$ is defined. Thus, since $\alpha(\llbracket\, E_1''\,\rrbracket\Sigma') \subseteq W$, $|A| = |\alpha(\llbracket\, \mathit{rename}(r'')(E_1'')\,\rrbracket\Sigma')| \leq |\alpha(\llbracket\, E_1''\,\rrbracket\Sigma')| \leq |W|$. Hence, since $B \subseteq W$ and $B \subseteq A$, $|W - B| \geq |A - B|$. Therefore, by A23 there exists a renaming function r''' such that $r'''(A) \cap A' \subseteq B$ and

$$E \equiv_{\Sigma'} \mathit{proj}(B)(\mathit{rename}(r''')(\mathit{rename}(r'')(E_1''))).$$

By A21

$$E \equiv_{\Sigma'} \mathit{proj}(B)(\mathit{rename}(r''' \circ r'')(E_1'')).$$

By lemma 3.17

$$E \equiv_{\Sigma'} E' = \mathit{proj}(r'(B))(\mathit{rename}(r''' \circ r'' \circ r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r''' \circ r'' \circ r_n)(v_n))$$

which is in RCP normal form.

By inspection

$$\begin{aligned}
\mathit{sub}(E') &= \{\, v_1, \ldots, v_n \} \cup \\
&= \cup \{\, \mathit{rename}(r''' \circ r'' \circ r_1)(v_1), \ldots, \mathit{rename}(r''' \circ r'' \circ r_n)(v_n) \} \cup \\
&= \cup \{\, \mathit{rename}(r''' \circ r'' \circ r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r''' \circ r'' \circ r_n)(v_n) \} \cup \\
&= \cup \{\, E' \}
\end{aligned}$$

Since for all i, $v_i \in \mathit{sub}(E_1')$, and since $\alpha(\llbracket\, \mathit{sub}(E_1')\,\rrbracket\Sigma') \subseteq W$,

$$\alpha(\llbracket\, \{v_1, \ldots, v_n\}\,\rrbracket\Sigma') \subseteq W.$$

Note that $r'''(A) \cap A' \subseteq B$ implies $r'''(A) \subseteq W$. Therefore, by lemma 3.17,

$$\begin{aligned}&\alpha(\llbracket\, \{\mathit{rename}(r''' \circ r'' \circ r_1)(v_1), \ldots, \mathit{rename}(r''' \circ r'' \circ r_n)(v_n)\}\,\rrbracket\Sigma') \subseteq r'''(A)\\ &\subseteq \; W,\end{aligned}$$

and

$$\begin{aligned}&\alpha(\llbracket\, \mathit{rename}(r''' \circ r'' \circ r_1)(v_1) \parallel \cdots \parallel \mathit{rename}(r''' \circ r'' \circ r_n)(v_n)\,\rrbracket\Sigma') \subseteq r'''(A)\\ &\subseteq \; W.\end{aligned}$$

Since by hypothesis $\alpha(\llbracket\, E\,\rrbracket\Sigma') \subseteq W$, and since $E \equiv_{\Sigma'} E'$, by lemma 3.11,

$$\alpha(\llbracket\, E'\,\rrbracket\Sigma') \subseteq W.$$

Therefore, $\alpha(\llbracket\, \mathit{sub}(E')\,\rrbracket\Sigma') \subseteq W$.

In addition, if a variable v appears k times in E, it appears k times in E_1, and therefore, by induction, it appears k times in E_1' and in the final normal form.

- Let $E = E_1 \parallel E_2$.

  Let $A = \alpha(\llbracket\, E\,\rrbracket\Sigma')$.

  Since by hypothesis $A \Subset W$, by lemma 3.13, there exists an alphabet X such that $A \Subset X$ and $X \Subset W$.

  Then, since $\mathit{sub}(E_1) \subseteq \mathit{sub}(E)$ and $\mathit{sub}(E_2) \subseteq \mathit{sub}(E)$, also $\alpha(\llbracket\, E_1\,\rrbracket\Sigma') \Subset X$ and $\alpha(\llbracket\, E_2\,\rrbracket\Sigma') \Subset X$. Then, by induction, E_1 and E_2 are equivalent to expressions E_1' and E_2' in RCP normal form

$$\begin{aligned}E_1' &= \mathit{proj}(B_1')(\mathit{rename}(r_{1,1})(v_{1,1}) \parallel \cdots \parallel \mathit{rename}(r_{1,n})(v_{1,n}))\\ E_2' &= \mathit{proj}(B_2')(\mathit{rename}(r_{2,1})(v_{2,1}) \parallel \cdots \parallel \mathit{rename}(r_{2,m})(v_{2,m}))\end{aligned}$$

  and $\alpha(\llbracket\, \mathit{sub}(E_1')\,\rrbracket\Sigma') \subseteq X$ and $\alpha(\llbracket\, \mathit{sub}(E_2')\,\rrbracket\Sigma') \subseteq X$.

Let

$$\begin{aligned}
E_1'' &= \textit{rename}(r_{1,1})(v_{1,1}) \parallel \cdots \parallel \textit{rename}(r_{1,n})(v_{1,n}) \\
A_1'' &= \alpha([\![\, E_1'' \,]\!]\Sigma') \\
B_1 &= B_1' \cap A_1'' \\
E_2'' &= \textit{rename}(r_{2,1})(v_{2,1}) \parallel \cdots \parallel \textit{rename}(r_{2,m})(v_{2,m}) \\
A_2'' &= \alpha([\![\, E_1'' \,]\!]\Sigma') \\
B_2 &= B_2' \cap A_2''
\end{aligned}$$

Then by theorem 3.4 and lemma 3.15

$$\begin{aligned}
E_1' &\equiv_{\Sigma'} \textit{proj}(B_1)(E_1'') \\
E_2' &\equiv_{\Sigma'} \textit{proj}(B_2)(E_2'')
\end{aligned}$$

Since $X \Subset W$, by lemma 3.13 there exists an alphabet Y such that $X \Subset Y$ and $Y \Subset W$.

Let $A_1' = \mathcal{Q}.\mathcal{A} - (W - Y)$ and $A_2' = \mathcal{Q}.\mathcal{A} - (Y - X)$. Clearly since $X \Subset Y \Subset W$ and $A_1'' \subseteq X$, $|W - Y| \geq |Y| \geq |X| \geq |A_1''|$. Therefore, since $B_1 \subseteq Y$, $|(\mathcal{Q}.\mathcal{A} - A_1') - B_1| = |(W - Y) - B_1| = |W - Y| \geq |A_1'' - B_1|$.

Similarly $|Y - X| \geq |X| \geq |A_2''|$. Therefore, since $B_2 \subseteq X$, $|(\mathcal{Q}.\mathcal{A} - A_2') - B_2| = |(Y - X) - B_2| = |Y - X| \geq |A_2'' - B_2|$.

Therefore, by A23 there exist renaming functions r_1' and r_2' such that $r_1'(A_1'') \cap A_1' \subseteq B_1$, $r_2'(A_2'') \cap A_2' \subseteq B_2$ and

$$\begin{aligned}
E_1' &\equiv_{\Sigma'} \textit{proj}(B_1)(\textit{rename}(r_1')(E_1'')) \\
E_2' &\equiv_{\Sigma'} \textit{proj}(B_2)(\textit{rename}(r_2')(E_2''))
\end{aligned}$$

By definition, $B_2 \subseteq X \subseteq A_1'$ and $\alpha([\![\, \textit{rename}(r_1')(E_1'') \,]\!]\Sigma') = r_1'(A_1'')$, therefore since $r_1'(A_1'') \cap A_1' \subseteq B_1$, also $r_1'(\alpha([\![\, \textit{rename}(r_1')(E_1'') \,]\!]\Sigma')) \cap B_2 \subseteq B_1$. Similarly, $r_2'(\alpha([\![\, \textit{rename}(r_1')(E_1'') \,]\!]\Sigma')) \cap B_1 \subseteq B_2$. Therefore by A19, denoting $B = B_1 \cup B_2$

$$\begin{aligned}
E_1' &\equiv_{\Sigma'} \textit{proj}(B)(\textit{rename}(r_1')(E_1'')) \\
E_2' &\equiv_{\Sigma'} \textit{proj}(B)(\textit{rename}(r_2')(E_2''))
\end{aligned}$$

By the previous definitions,

$$\begin{aligned}\alpha(\llbracket\, \mathit{rename}(r_1')(E_1'')\,\rrbracket\Sigma') &\subseteq B_1 \cup (W - Y),\\ \alpha(\llbracket\, \mathit{rename}(r_2')(E_2'')\,\rrbracket\Sigma') &\subseteq B_2 \cup (Y - X).\end{aligned}$$

In addition, since $(W - Y) \cap (Y - X) = \emptyset$, $(B_1 \cup (W - Y)) \cap (B_2 \cup (Y - X)) = B_1 \cap B_2 \subseteq B$. Hence $\alpha(\llbracket\, \mathit{rename}(r_1')(E_1'')\,\rrbracket\Sigma') \cap \alpha(\llbracket\, \mathit{rename}(r_2')(E_2'')\,\rrbracket\Sigma') \subseteq B$. Therefore, by A25

$$E \equiv_{\Sigma'} \mathit{proj}(B)(\mathit{rename}(r_1')(E_1'') \parallel \mathit{rename}(r_2')(E_2'')).$$

By lemma 3.17

$$\begin{aligned}\mathit{rename}(r_1')(E_1'') &\equiv_{\Sigma'} \mathit{rename}(r_1' \circ r_{1,1})(v_{1,1}) \parallel \cdots \parallel \mathit{rename}(r_1' \circ r_{1,n})(v_{1,n})\\ \mathit{rename}(r_2')(E_2'') &\equiv_{\Sigma'} \mathit{rename}(r_2' \circ r_{2,1})(v_{2,1}) \parallel \cdots \parallel \mathit{rename}(r_2' \circ r_{2,m})(v_{2,m}).\end{aligned}$$

Therefore by theorem 3.4

$$\begin{aligned}E \equiv_{\Sigma'} E' = \mathit{proj}(B)(&\mathit{rename}(r_1' \circ r_{1,1})(v_{1,1}) \parallel \cdots \parallel \mathit{rename}(r_1' \circ r_{1,n})(v_{1,n}) \parallel\\ &\parallel \mathit{rename}(r_2' \circ r_{2,1})(v_{2,1}) \parallel \cdots \parallel \mathit{rename}(r_2' \circ r_{2,m})(v_{2,m}))\end{aligned}$$

which is in RCP normal form.

By inspection

$$\begin{aligned}\mathit{sub}(E') &= \{\, v_{1,1}, \ldots, v_{1,n} \,\} \cup\\ &= \cup \{\, v_{2,1}, \ldots, v_{2,m} \,\} \cup\\ &= \cup \{\, \mathit{rename}(r_1' \circ r_{1,1})(v_{1,1}), \ldots, \mathit{rename}(r_2' \circ r_{2,m})(v_{2,m}) \,\} \cup\\ &= \cup \{\, \mathit{rename}(r_1' \circ r_{1,1})(v_{1,1}) \parallel \cdots \parallel \mathit{rename}(r_2' \circ r_{2,m})(v_{2,m}) \,\} \cup\\ &= \cup \{\, E' \,\}\end{aligned}$$

Since for all i, $v_{1,i} \in \mathit{sub}(E_1')$, and since $\alpha(\llbracket\, \mathit{sub}(E_1')\,\rrbracket\Sigma') \subseteq W$,

$$\alpha(\llbracket\, \{\, v_{1,1}, \ldots, v_{1,n} \,\}\,\rrbracket\Sigma') \subseteq W.$$

Similarly

$$\alpha(\llbracket\, \{\, v_{2,1}, \ldots, v_{2,m} \,\}\,\rrbracket\Sigma') \subseteq W.$$

Note that $r'_1(A''_1) \cap A'_1 \subseteq B_1$ implies $r'_1(A''_1) \subseteq W$, since $\mathcal{Q}.\mathcal{A} - W \subseteq A'_1$ and $B_1 \subseteq A''_1 \subseteq W$. Therefore by lemma 3.17

$$\alpha([\![\{ \mathit{rename}(r'_1 \circ r_{1,1})(v_{1,1}), \ldots, \mathit{rename}(r'_1 \circ r_{1,n})(v_{1,n})\}]\!]\Sigma') \subseteq r'_1(A''_1) \subseteq W$$
$$\alpha([\![\mathit{rename}(r'_1 \circ r_{1,1})(v_{1,1}) \parallel \cdots \parallel \mathit{rename}(r'_1 \circ r_{1,n})(v_{1,n})]\!]\Sigma') \subseteq r'_1(A''_1) \subseteq W$$

Similarly

$$\alpha([\![\{ \mathit{rename}(r'_2 \circ r_{2,1})(v_{2,1}), \ldots, \mathit{rename}(r'_2 \circ r_{2,m})(v_{2,m})\}]\!]\Sigma') \subseteq r'_2(A''_2) \subseteq W$$
$$\alpha([\![\mathit{rename}(r'_2 \circ r_{2,1})(v_{2,1}) \parallel \cdots \parallel \mathit{rename}(r'_2 \circ r_{2,m})(v_{2,m})]\!]\Sigma') \subseteq r'_2(A''_2) \subseteq W$$

Since by hypothesis $\alpha([\![E]\!]\Sigma') \subseteq W$, and since $E \equiv_{\Sigma'} E'$, by lemma 3.11

$$\alpha([\![E']\!]\Sigma') \subseteq W.$$

Therefore, $\alpha([\![\mathit{sub}(E')]\!]\Sigma') \subseteq W$.

In addition, assume a variable appears m times in E. Then it appears j times in E_1 and k times in E_2 such that $m = j + k$. By induction, it appears j times in E'_1 and k times in E'_2, and therefore it appears $j + k$ times in the final normal form.

□

The rest of this section is devoted to proving the validity of some of the axioms for a few examples.

Example 3.18 (Alphabet Algebra). The alphabet agent algebra $\mathcal{Q}$ described in example 2.26 is a normalizable agent algebra. Here we show that A23 is satisfied.

Lemma 3.19. $\mathcal{Q}$ satisfies A23.

Proof: Let B, A and A' be alphabets over $\mathcal{Q}$ such that $|(\mathcal{Q}.\mathcal{A} - A') - B| \geq |A - B|$. Let $r' : (A - B) \mapsto (\mathcal{Q}.\mathcal{A} - A') - B$ be any injection from $A - B$ to $\mathcal{Q}.\mathcal{A} - A' - B$. The injection exists because of the assumption on the cardinality of the sets. Then define an injection $r : A \mapsto \mathcal{Q}.\mathcal{A}$ as follows:

$$r(a) = \begin{cases} r'(a) & \text{if } a \in A - B \\ id_A(a) & \text{otherwise} \end{cases}$$

Then if p is an agent such that $\alpha(p) \subseteq A$, and restricting the codomain of r to $r(A)$,

$$\begin{aligned} \mathit{proj}(B)(p) &= B \cap \alpha(p) \\ &= B \cap r(\alpha(p)) \\ &= \mathit{proj}(B)(\mathit{rename}(r)(p)). \end{aligned}$$

Therefore A23 is satisfied. □

Example 3.20 (IO Agent Algebra). The IO agent algebra $\mathcal{Q}$ described in example 2.29 is normalizable. Here we show that A25 is satisfied.

Lemma 3.21. $\mathcal{Q}$ satisfies A25.

Proof: Let B be an alphabet and let p_1 and p_2 be two agents such that $\alpha(p_1) \cap \alpha(p_2) \subseteq B$. Assume $\mathit{proj}(B)(p_1 \parallel p_2)$ is defined. Then by definition $(I_1 \cup I_2) - (O_1 \cup O_2) \subseteq B$. We now show that $I_1 \subseteq B$. Let $i \in I_1$ be a signal. Then by definition $i \notin O_1$. Assume $i \notin O_2$. Then $i \in (I_1 \cup I_2) - (O_1 \cup O_2)$ and therefore $i \in B$. On the other hand, assume $i \in O_2$. Then $i \in \alpha(p_1) \cap \alpha(p_2)$. Therefore $i \in B$. Hence $I_1 \subseteq B$. Similarly, $I_2 \subseteq B$. Therefore $\mathit{proj}(B)(p_1) \parallel \mathit{proj}(B)(p_2)$ is defined. In addition

$$\mathit{proj}(B)(p_1 \parallel p_2) = ((I_1 \cup I_2) - (O_1 \cup O_2), (O_1 \cup O_2) \cap B)$$
$$\mathit{proj}(B)(p_1) \parallel \mathit{proj}(B)(p_2) = ((I_1 \cup I_2) - ((O_1 \cup O_2) \cap B), (O_1 \cup O_2) \cap B)$$

Clearly

$$(I_1 \cup I_2) - (O_1 \cup O_2) \subseteq (I_1 \cup I_2) - ((O_1 \cup O_2) \cap B).$$

Let now $i \in (I_1 \cup I_2) - ((O_1 \cup O_2) \cap B)$. Then either $i \in I_1$ or $i \in I_2$ (or both). If $i \in I_1$ then $i \notin O_1$ and $i \notin O_2 \cap B$. If $i \in O_2$, then $i \in \alpha(p_1) \cap \alpha(p_2)$ and therefore $i \in B$. But then $i \in O_2 \cap B$, a contradiction. Hence $i \notin O_2$. Then $i \in (I_1 \cup I_2) - (O_1 \cup O_2)$. Similarly if $i \in I_2$. Therefore

$$(I_1 \cup I_2) - ((O_1 \cup O_2) \cap B) = (I_1 \cup I_2) - (O_1 \cup O_2)$$

and

$$\mathit{proj}(B)(p_1 \parallel p_2) = \mathit{proj}(B)(p_1) \parallel \mathit{proj}(B)(p_2).$$

Assume now $\mathit{proj}(B)(p_1 \parallel p_2)$ is not defined. Then there exists $i \in (I_1 \cup I_2) - (O_1 \cup O_2)$ such that $i \notin B$. But then either $i \in I_1$ or $i \in I_2$, and therefore either

$I_1 \not\subseteq B$ or $I_2 \not\subseteq B$. Hence either *proj*$(B)(p_1)$ or *proj*$(B)(p_2)$ (or both) is undefined. Therefore *proj*$(B)(p_1) \parallel$ *proj*$(B)(p_2)$ is undefined. □

Example 3.22 (Dill's IO Agent Algebra). The Dill's style IO agent algebra $\mathcal{Q}$ described in example 2.32 is not normalizable. In fact, it does not satisfy A18. The IO agent algebra described in example 3.20 is a generalization of $\mathcal{Q}$ that is normalizable.

However, Dill's style IO agent algebra is closed-normalizable. This doesn't appear to be a limitation in Dill's work, since the notion of refinement requires that two agents have the same sets of inputs and outputs signals. Dill shows that, if assignments to variables are restricted to assign only certain input and output signals, then the expressions are normalizable [33]. While this is sufficient to prove the results on refinement verification using mirrors, it is a special case that we need to generalize in our framework.

3.1.1 Construction of Algebras

It is natural to ask whether the constructions introduced in section 2.3 and subsection 2.4.1 preserve the properties of normalization of the expressions. In other words, if $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are normalizable agent algebras, is their product $\mathcal{Q}_1 \times \mathcal{Q}_2$ and their disjoint sum $\mathcal{Q}_1 \uplus \mathcal{Q}_2$ also normalizable?

Clearly, this is the case for disjoint sum, since the resulting agent algebra is simply the juxtaposition of two agent algebras, that are otherwise unrelated.

Theorem 3.23. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be normalizable agent algebras. Then their disjoint sum $\mathcal{Q}_1 \uplus \mathcal{Q}_2$ is a normalizable agent algebra.

The same is not necessarily true for products. The problem lies in A22 and A23, both of which demand the existence of renaming functions that make certain equations true.

Take for example A22. Since $\mathcal{Q}_1$ is normalizable, for all renaming functions r and all alphabets B, there exist renaming functions r_1' and r_1'' such that for all agents $p_1 \in \mathcal{Q}_1.D$,

$$\textit{rename}(r)(\textit{proj}(B)(p_1)) = \textit{proj}(r_1'(B))(\textit{rename}(r_1'')(p_1)).$$

Similarly, since $\mathcal{Q}_2$ is normalizable, there exist renaming functions r_2' and r_2'' such that for all agents $p_2 \in \mathcal{Q}_2.D$,

$$\textit{rename}(r)(\textit{proj}(B)(p_2)) = \textit{proj}(r_2'(B))(\textit{rename}(r_2'')(p_2)).$$

In order for the product $\mathcal{Q}_1 \times \mathcal{Q}_2$ to be normalizable, there must be renaming functions r' and r'' such that for all agents $p = (p_1, p_2) \in \mathcal{Q}_1.D \times \mathcal{Q}_2.D$,

$$\mathit{rename}(r)(\mathit{proj}(B)(p)) = \mathit{proj}(r'(B))(\mathit{rename}(r'')(p)).$$

But the result follows from the hypothesis only if $r_1' = r_2'$ and $r_1'' = r_2''$, which is not necessarily the case.

In most practical cases, however, the product of normalizable agent algebras is in fact normalizable. This is because in practice the renaming operator is defined similarly for different algebras, given its strong intuitive interpretation. In addition, all other axioms are indeed preserved by the product, so that only the validity of A22 and A23 must be established.

3.2 Conformance

Let p and p' be two agents in an ordered agent algebra. Intuitively, if we interpret the order as refinement, if $p \preceq p'$ then p can be substituted for p' in every context in which p' occurs. If this is the case we say that p *conforms* to p'. In this section we make this notion of substitutability precise. In our formalization, conformance is parameterized by a set of agents G, called a conformance set, and we only require that for p to conform to p', p can be substituted for p' for all contexts that evaluate in G. Intuitively, the set G forms an initial partition of the agents. This partition is then refined by considering the contexts whose evaluation falls in the conformance set. The remaining contexts, which are of no interest for substitutability, are therefore ignored.

Conformance can be made more general by explicitly considering only a subset of the possible contexts. We call this notion *relative conformance*. In this section we will study these generalizations, and show the conditions under which relative conformance corresponds to conformance. We are particularly interested in composition contexts, also called *environments*, which are limited to the parallel composition with a single agent. Composition contexts will be the basis for studying mirror functions in the next section.

The concept of the context of an agent in a system plays a central role in the definition of conformance. It can be formalized using agent expressions.

Definition 3.24 (Expression Context). Let $\mathcal{Q}$ be an agent algebra. An expression context $E[\beta]$ over $\mathcal{Q}$ is an expression over $\mathcal{Q}$ with one free variable.

An expression context may or may not be defined depending on the agent that replaces the free variable. However, the property of $\top$-monotonicity of the operators of an ordered agent algebra transfers to expression contexts, as well.

Theorem 3.25. Let $\mathcal{Q}$ be an ordered agent algebra and $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}.D$ be two agents such that $p \preceq p'$. For all expression contexts $E[\beta]$, if $E[p']$ is defined then $E[p]$ is also defined and $E[p] \preceq E[p']$.

Proof: The proof is by induction on the structure of the expression context.

- If $E[\beta] = q$ or $E[\beta] = \beta$ then the result follows directly from the hypothesis.

- Let $E[\beta] = \mathit{rename}(r)(E'[\beta])$ and assume $E[p']$ is defined. Then also $E'[p']$ is defined. By induction hypothesis $E'[p]$ is defined and $E'[p] \preceq E'[p']$. Since $\mathit{rename}(r)(E'[p'])$ is defined and *rename* is $\top$-monotonic, then $\mathit{rename}(r)(E'[p])$ is defined and

$$\mathit{rename}(r)(E'[p]) \preceq \mathit{rename}(r)(E'[p']).$$

- Let $E[\beta] = \mathit{proj}(B)(E'[\beta])$ and assume $E[p']$ is defined. Then also $E'[p']$ is defined. By induction hypothesis $E'[p]$ is defined and $E'[p] \preceq E'[p']$. Since $\mathit{proj}(B)(E'[p'])$ is defined and *proj* is $\top$-monotonic, then $\mathit{proj}(B)(E'[p])$ is defined and

$$\mathit{proj}(B)(E'[p]) \preceq \mathit{proj}(B)(E'[p']).$$

- Let $E[\beta] = E_1[\beta] \parallel E_2[\beta]$ and assume $E[p']$ is defined. Then also $E_1[p']$ and $E_2[p']$ are defined. By induction hypothesis $E_1[p]$ is defined and $E_1[p] \preceq E_1[p']$. Similarly, $E_2[p]$ is defined and $E_2[p] \preceq E_2[p']$. Since $E_1[p'] \parallel E_2[p']$ is defined and $\parallel$ is $\top$-monotonic, then $E_1[p] \parallel E_2[p']$ is also defined and $E_1[p] \parallel E_2[p'] \preceq E_1[p'] \parallel E_2[p']$. Similarly we conclude $E_1[p] \parallel E_2[p] \preceq E_1[p] \parallel E_2[p']$ and therefore since $\preceq$ is transitive

$$E_1[p] \parallel E_2[p] \preceq E_1[p'] \parallel E_2[p'].$$

$\square$

An ordered agent algebra $\mathcal{Q}$ has a conformance order parameterized by a set of agents G when the order corresponds to substitutability in the following sense.

Definition 3.26 (Conformance Order). Let $\mathcal{Q}$ be an ordered agent algebra and let G be a set of agents of $\mathcal{Q}$. We say $\mathcal{Q}$ has a *G-conformance order* if and only if for all agents p and p', $p \preceq p'$ if and only if for all expression contexts E, if $E[p'] \in G$ then $E[p] \in G$.

The implication in this definition is strong in the sense that if $E[p'] \in G$, then $E[p]$ must be defined (and be a member of G).

Each set of agents G induces a particular order, whether or not the algebra has a G-conformance order.

Definition 3.27. Let $\mathcal{Q}$ be an agent algebra and let G be a set of agents of $\mathcal{Q}$. We define $\mathcal{Q}.\mathit{conf}(G)$ to be the agent algebra that is identical to $\mathcal{Q}$ except that it has a G-conformance order.

We denote the order of $\mathcal{Q}.\mathit{conf}(G)$ with the symbol $\preceq_G$ and we say that G *induces* the order $\preceq_G$. In the rest of this section we will study some of the properties of $\mathcal{Q}.\mathit{conf}(G)$. In particular we are interested in characterizing when $\mathcal{Q}.\mathit{conf}(G)$ is an ordered agent algebra (i.e., the operators of projection, renaming and parallel composition are $\top$-monotonic) and when $\mathcal{Q}$ has a G-conformance order,

Lemma 3.28. Let $\mathcal{Q}$ be an ordered agent algebra and let G be a subset of $\mathcal{Q}.D$. Consider the agent algebras $\mathcal{Q}.\mathit{conf}(G)$ and $\mathcal{Q}.\mathit{conf}(D)$. Then $\mathcal{Q}.\mathit{conf}(G)$ is an ordered agent algebra if and only if for all agents p and p',

$$p \preceq_G p' \Rightarrow p \preceq_D p'.$$

Proof: To show that $\mathcal{Q}.\mathit{conf}(G)$ is an ordered agent algebra, we need only show that its renaming, projection and parallel composition operators are $\top$-monotonic relative to its agent ordering. We prove the projection case (the others are similar).

Let p and p' be two agents such that $p \preceq_G p'$. We must show that if $\mathit{proj}(B)(p')$ is defined, then $\mathit{proj}(B)(p)$ is defined, and $\mathit{proj}(B)(p) \preceq_G \mathit{proj}(B)(p')$.

Since $p \preceq_G p'$, by hypothesis, $p \preceq_D p'$. Therefore, by definition 3.26, for all expression contexts E, if $E[p'] \in D$ then $E[p] \in D$. That is, if $E[p']$ is defined, then $E[p]$ is defined. Hence, if $E = \mathit{proj}(B)(\beta)$, if $\mathit{proj}(B)(p')$ is defined then $\mathit{proj}(B)(p)$ is defined.

Let now $E[\beta]$ be an expression context. We want to show that if $E[\mathit{proj}(B)(p')] \in G$, then $E[\mathit{proj}(B)(p)] \in G$. By lemma 2.50

$$E[\mathit{proj}(B)(p')] = E[\mathit{proj}(B)(\beta')[p']] = E[\beta/\mathit{proj}(B)(\beta)][p'].$$

Let now $E' = E[\beta/\mathit{proj}(B)(\beta)]$. Then

$$\begin{aligned}
&E[\mathit{proj}(B)(p')] \in G \\
&\quad\Rightarrow\quad E'[p'] \in G \\
&\quad\Rightarrow\quad E'[p] \in G \\
&\quad\Rightarrow\quad E[\mathit{proj}(B)(p)] \in G
\end{aligned}$$

Therefore, by definition 3.26

$$\mathit{proj}(B)(p) \preceq_G \mathit{proj}(B)(p').$$

Hence, the projection operator is $\top$-monotonic relative to $\preceq_G$.

Conversely, assume the operators are $\top$-monotonic relative to $\preceq_G$ and let p and p' be two agents such that $p \preceq_G p'$. Then, by theorem 3.25, for all expression contexts E, if $E[p']$ is defined, then $E[p]$ is defined. Hence, if $E[p'] \in D$, then $E[p] \in D$. Therefore, by definition 3.26, $p \preceq_D p'$. □

Corollary 3.29. Let $\mathcal{Q}$ be an ordered agent algebra. Then $\mathcal{Q}.\mathit{conf}(D)$ is an ordered agent algebra.

Although G can be any arbitrary set of agents, G must be downward closed relative to $\mathcal{Q}.\preceq$ in order for $\mathcal{Q}$ to have a G-conformance order.

Theorem 3.30. Let $\mathcal{Q}$ be an agent algebra and let G be a set of agents. Then G is downward closed relative to $\preceq_G$.

Proof: Let $p' \in G$ and let $p \preceq_G p'$. Consider the expression context $E = \beta$. Then clearly $E[p'] \in G$. But then, by definition 3.26, since $p \preceq_G p'$, also $E[p] = p \in G$. Therefore G is downward closed. □

Corollary 3.31. Let $\mathcal{Q}$ be an ordered agent algebra. If $\mathcal{Q}$ has a G-conformance order, then G is downward closed relative to $\mathcal{Q}.\preceq$.

If an expression context evaluates in G for a certain agent p', then it evaluates in G for all agents $p \preceq p'$.

Corollary 3.32. Let $\mathcal{Q}$ be an ordered agent algebra and $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}.D$ be two agents such that $p \preceq p'$. For all expression contexts $E[\beta]$, if $E[p'] \in G$ then $E[p]$ is also defined and $E[p] \in G$.

Proof: The result follows from theorem 3.25 and corollary 3.31. □

In this work we will be particularly interested in the following special case.

Corollary 3.33. Let $\mathcal{Q}$ be an ordered agent algebra with a G-conformance order. Let $p' \parallel q \in G$ and let $p \preceq p'$. Then $p \parallel q$ is defined and $p \parallel q \in G$.

In the following we will explore the relationships between the order of $\mathcal{Q}$ and the orders induced by various conformance sets.

Theorem 3.34. Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents. Then

$$p \preceq q \Rightarrow p \preceq_G q.$$

Proof: Since $p \preceq q$ and the operators are $\top$-monotonic, then by theorem 3.25 for all expression contexts E, if $E[q]$ is defined then $E[p]$ is defined. In addition, $E[p] \preceq E[q]$. Assume now that $E[q] \in G$. Then, since G is downward closed, also $E[p] \in G$. Therefore $p \preceq_G q$. □

Notice that if G is downward closed, then the forward implication in definition 3.26 follows from theorem 3.34. If $\mathcal{Q}$ has a G-conformance order then the order is weak enough to ensure that the reverse implication also holds.

Corollary 3.35. If $\mathcal{Q}$ has a G-conformance order, then $\mathcal{Q}$ and $\mathcal{Q}.\mathit{conf}(G)$ are identical agent algebras.

The set of all agents D plays a special role, since it is always downward closed, no matter what order the agent algebra may have, and the order it induces always makes the operators $\top$-monotonic. The following theorem shows that given an agent algebra $\mathcal{Q}$, the ordered agent algebra $\mathcal{Q}.\mathit{conf}(D)$ has the weakest order that makes the operators $\top$-monotonic.

Corollary 3.36. Let $\mathcal{Q}$ be an ordered agent algebra. Then

$$p \preceq q \Rightarrow p \preceq_D q.$$

Proof: The result follows from theorem 3.34, since D is always downward closed. □

Since the discrete order (i.e., the order such that $p \preceq p'$ if and only if $p = p'$) also makes the operator $\top$-monotonic, any order of an ordered agent algebra is bounded by the discrete order and by $\preceq_D$.

The following two results show that if $\mathcal{Q}$ has the weakest conformance order (i.e., $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(D)$), then any downward closed set of agents characterizes the order.

Corollary 3.37. Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents such that $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G)$. Then

$$p \preceq_G q \Rightarrow p \preceq_D q.$$

Proof: Since $\mathcal{Q}$ is an ordered agent algebra, and $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G)$, also $\mathcal{Q}.\mathit{conf}(G)$ is an ordered agent algebra. Then the result follows from corollary 3.36. □

Corollary 3.38. Let $\mathcal{Q}$ be an ordered agent algebra such that $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(D)$, and let G be a downward closed set of agents such that $\mathcal{Q}.\mathit{conf}(G)$ is an ordered agent algebra. Then $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G)$.

Proof: Let p and q be two agents. We must show that $p \preceq q$ if and only if $p \preceq_G q$. The forward direction follows directly from theorem 3.34.

The following series of implications proves the reverse direction.

$p \preceq_G q$

by corollary 3.37

$\Rightarrow \quad p \preceq_D q$

since $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(D)$

$\Rightarrow \quad p \preceq q$

□

These results show that, in general, an ordered agent algebra $\mathcal{Q}$ can be characterized by several conformance sets. The particular choice of G influences the complexity of verifying the conformance relation, as we will see in the next few sections when we introduce relative conformance and mirror functions.

Note also that $\mathcal{Q}.\mathit{conf}(G)$ is not necessarily an agent algebra, in the sense that the operators may not be $\top$-monotonic relative to the agent ordering, even if they are $\top$-monotonic relative to the original ordering (see lemma 3.28). This is in practice not a problem, since we typically start from an ordered agent algebra, and then characterize its order in terms of a conformance set. In that case, since $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G)$, also $\mathcal{Q}.\mathit{conf}(G)$ is an ordered agent algebra.

3.2.1 Relative Conformance

In definition 3.26, conformance is defined in terms of all expression contexts. More generally, we can define conformance relative to a set of contexts.

Definition 3.39 (Relative Conformance). Let $\mathcal{Q}$ be an agent algebra and let G be a set of agents of $\mathcal{Q}$. We say $\mathcal{Q}$ has a *G-conformance order relative to a set of contexts $\mathcal{E}'$* if and only if for all agents p and p', $p \preceq p'$ if and only if for all expression contexts $E \in \mathcal{E}'$, if $E[p'] \in G$ then $E[p] \in G$.

A particularly interesting subset of contexts is the set of environments that consist of a parallel composition with an arbitrary agent.

Definition 3.40 (Composition Conformance). Let $\mathcal{Q}$ be an agent algebra and let G be a set of agents of $\mathcal{Q}$. We say $\mathcal{Q}$ has a *G-conformance order relative to composition* if and only if for all agents p and p', $p \preceq p'$ if and only if for all agents q, if $p' \parallel q \in G$ then $p \parallel q \in G$.

As with conformance, we define $\mathcal{Q}.\mathit{conf}(G, \mathcal{E}')$ to be the agent algebra that is identical to $\mathcal{Q}$ except that it has a G-conformance order relative to $\mathcal{E}'$. We denote the order of $\mathcal{Q}.\mathit{conf}(G, \mathcal{E}')$ with the symbol $\preceq_G^{\mathcal{E}'}$. In particular, $\mathcal{Q}.\mathit{conf}(G, \parallel)$ and $\preceq_G^{\parallel}$ denote G-conformance relative to composition.

Unlike conformance, $\mathcal{Q}.\mathit{conf}(G, \mathcal{E}')$ is not necessarily an ordered agent algebra even if $p \preceq_G^{\mathcal{E}'} p' \Rightarrow p \preceq_D^{\mathcal{E}'} p'$, since the operators of the algebra may not be $\top$-monotonic (see lemma 3.28). In addition, if $\mathcal{Q}$ has a G-conformance order relative to $\mathcal{E}'$, then G is not necessarily downward closed (see corollary 3.31).

Conformance implies relative conformance in the following sense.

Lemma 3.41. Let $\mathcal{Q}$ be an agent algebra and let $\mathcal{E}'$ be a set of contexts. Then for all agents p and p'

$$p \preceq_G p' \Rightarrow p \preceq_G^{\mathcal{E}'} p'.$$

Proof: Definition 3.39 is verified since the condition is by hypothesis true of all contexts. $\square$

In particular, if $p \preceq_G p'$, then $p \preceq_G^{\parallel} p'$.

Despite the above result, if $\mathcal{Q}$ is an ordered agent algebra and $\mathcal{E}'$ is a set of contexts, $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G)$ does not necessarily imply $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G, \mathcal{E}')$. This is because the reverse implication above does not hold. However, if $\mathcal{Q}$ has a G-conformance order relative to some set of contexts $\mathcal{E}'$ and G is downward closed, then it also has a G-conformance order.

Theorem 3.42. Let $\mathcal{Q}$ be an ordered agent algebra, $\mathcal{E}'$ be a set of contexts and let G be a downward closed set of agents. Assume for all agents p and p',

$$p \preceq_G^{\mathcal{E}'} p' \Rightarrow p \preceq p'.$$

Then $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G, \mathcal{E}') = \mathcal{Q}.\mathit{conf}(G)$.

Proof: We must show that for all agents p and p', $p \preceq p'$ if and only if $p \preceq_G p'$ if and only if $p \preceq_G^{\|} p'$. The result follows from the following circle of implications:

$$\begin{aligned} & p \preceq p' \\ & \quad \text{by theorem 3.34, since } G \text{ is downward closed} \\ \Rightarrow \quad & p \preceq_G p' \\ & \quad \text{by lemma 3.41} \\ \Rightarrow \quad & p \preceq_G^{\mathcal{E}'} p' \\ & \quad \text{by hypothesis} \\ \Rightarrow \quad & p \preceq p'. \end{aligned}$$

□

Corollary 3.43. Let $\mathcal{Q}$ be an ordered agent algebra, $\mathcal{E}'$ be a set of contexts and G a downward closed set of agents such that $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G, \mathcal{E}')$. Then $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(G)$.

Proof: The result follows from theorem 3.42, since by hypothesis $p \preceq_G^{\mathcal{E}'} p' \Rightarrow p \preceq_G p'$. □

In particular, if $\mathcal{Q}$ has a G-conformance order relative to composition and G is downward closed, then it has a G-conformance order. In the examples that follow we will try to show, when possible, that conformance relative to composition corresponds exactly to conformance. When that is the case, it may be possible to find efficient ways to check the conformance relation, as we shall see in section 3.3.

Example 3.44 (Alphabet Algebra). Consider the agent algebra $\mathcal{Q}$ described in example 2.26, with the order such that $p \preceq p'$ if and only if $p \subseteq p'$. This order is the weakest order that makes the operators $\top$-monotonic, hence $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(D)$. However, D does not characterize the order in terms of conformance relative to composition. Instead, conformance relative to composition induces the order such that every agent refines any other agent.

Theorem 3.45. For all agents p and p', $p \preceq_D^{\|} p'$.

Proof: The result follows from the fact that the conformance set in this case is the set of all agents D, and $\|$ is always defined. Therefore the condition in definition 3.40 is always satisfied. □

In order to characterize the order in terms of conformance relative to composition we must consider the set $G = 2^{\mathcal{A}} - \mathcal{A}$, i.e., the set of all subsets of $\mathcal{A}$ except $\mathcal{A}$ itself. Then

Theorem 3.46. Let p and p' be two agents. Then the following statements are equivalent:

1. $p \subseteq p'$.
2. $p \preceq_G p'$.
3. $p \preceq_G^{\|} p'$.

Proof: We already know that $1 \Rightarrow 2$ (by theorem 3.34, since G is downward closed) and that $2 \Rightarrow 3$ (by lemma 3.41). The remaining implication is proved below.

Lemma 3.47. $(3 \Rightarrow 1)$: Let p and p' be agents such that for all agents q, if $p' \| q \in G$ then $p \| q \in G$. Then $p \subseteq p'$.

Proof: Let p and p' be agents such that for all agents q, if $p' \| q \in G$ then $p \| q \in G$. By the definition of G, for all agents q, if $p' \| q \neq \mathcal{A}$ (i.e., $p' \| q \in G$), then $p \| q \neq \mathcal{A}$. Assume now, by contradiction, that $a \in p$ and $a \notin p'$. Consider $q = \mathcal{A} - p$. Then

$$p' \| q = p' \cup q = p' \cup (\mathcal{A} - p).$$

Since $a \notin p'$ and $a \notin \mathcal{A} - p$ (because $a \in p$), then $a \notin p' \| q$. Thus $p' \| q \neq \mathcal{A}$. Thus, by hypothesis, also $p \| q \neq \mathcal{A}$. However

$$p \| q = p \cup q = p \cup (\mathcal{A} - p) = \mathcal{A},$$

a contradiction. Thus $p \subseteq p'$. □

□

This is the only G that characterizes the order in terms of conformance relative to composition. In fact it is easy to show that for all $a \in \mathcal{A}$, the set $\mathcal{A} - \{ a \}$ must be in G. Then, to characterize the order, G must be downward closed. Thus $G = 2^{\mathcal{A}} - \mathcal{A}$.

Example 3.48 (IO Agent Algebra). Consider the IO agent algebra $\mathcal{Q}$ defined in example 2.10 with the order defined in example 2.29. We now characterize the order in terms of conformance. Let $G = \{ (I, O) : I = \emptyset \}$, i.e., the set of all agents that have no inputs. Then

Theorem 3.49. Let p and p' be IO agents. Then the following three statements are equivalent:

1. $p \preceq p'$ (i.e., $I \subseteq I'$ and $O = O'$).
2. $p \preceq_G p'$.
3. $p \preceq_G^{\|} p'$.

Proof: First we show that G is downward closed, then that $p \preceq_G^{\|} p'$ implies $p \preceq p'$. The result then follows from theorem 3.42.

Lemma 3.50. G is downward closed with respect to $\preceq$.

Proof: Let $p' \in G$. Then p' is of the form $(\emptyset, O')$ for some alphabet O'. Let $p = (I, O)$ be an agent such that $p \preceq p'$. Then, by the definition of the order, $I \subseteq \emptyset$, and therefore $I = \emptyset$. Hence $p \in G$. Therefore G is downward closed. $\square$

Lemma 3.51. $(3 \Rightarrow 1)$: Let p and p' be IO agents such that for all agents q, if $p' \| q \in G$ then $p \| q \in G$. Then $I \subseteq I'$ and $O = O'$.

Proof: We prove the result in steps.

($O \subseteq O'$) Assume, by contradiction, that there exists $o \in O$ such that $o \notin O'$. Consider $q = (O', I' \cup \{ o \})$. Then $p' \| q$ is defined because $O' \cap (I' \cup \{ o \}) = \emptyset$ since by hypothesis $O' \cap I' = \emptyset$ and $o \notin O'$. In addition $p' \| q \in G$. But then by hypothesis $p \| q$ is defined and $p \| q \in G$. However $\{ o \} \subseteq O \cap (I' \cup \{ o \})$, hence $O \cap (I' \cup \{ o \}) \neq \emptyset$, a contradiction.

($O' \subseteq O$) Assume, by contradiction, that there exists $o \in O'$ such that $o \notin O$. Consider $q = (O', I')$. By hypothesis $o \notin I'$. Clearly $p' \| q$ is defined and $p' \| q \in G$, so by hypothesis also $p \| q \in G$ is defined and $p \| q \in G$. However

$$p \| q = ((I \cup O') - (O \cup I'), O \cup I')$$

However, since $o \in (I \cup O')$ and $o \notin (O \cup I')$, $p \| q \notin G$, a contradiction.

($I \subseteq I'$) Assume, by contradiction, that there exists $i \in I$ such that $i \notin I'$. By hypothesis we also have $i \notin O$. Consider $q = (O', I')$. Clearly $p' \parallel q$ is defined and $p' \parallel q \in G$, so by hypothesis also $p \parallel q \in G$ is defined and $p \parallel q \in G$. However

$$p \parallel q = ((I \cup O') - (O \cup I'), O \cup I')$$

However, since $i \in (I \cup O')$ and $i \notin (O \cup I')$, $p \parallel q \notin G$, a contradiction.

□

□

Let us now consider the set of agents $G = \mathcal{Q}.D$ that consists of all agents. Then an expression evaluates in G if and only if the expression is defined. The following two theorems show that D still characterizes the order in terms of conformance, but it does not characterize the order in terms of conformance relative to composition.

Theorem 3.52. Let p and p' be IO agents. Then $p \preceq p'$ if and only if $p \preceq_D p'$.

Proof: The forward implication follows from theorem 3.34 since D is downward closed.

For the reverse implication, let $p = (I, O)$ and $p' = (I', O')$ be IO agents such that $p \preceq_D p'$. Then for all expression contexts E, if $E[p']$ is defined, then $E[p]$ is defined.

($I \subseteq I'$) Consider the context $E = \mathit{proj}(I')(\beta)$. Then $E[p'] = \mathit{proj}(I')(p')$ is defined since $I' \subseteq I'$. Then also $E[p] = \mathit{proj}(I')(p)$ must be defined. Therefore $I \subseteq I'$.

($O \subseteq O'$) Assume by contradiction that there exists $o \in O$ such that $o \notin O'$. Consider the agent $q = (\emptyset, \{o\})$ and the context $E = \beta \parallel q$. Then, since $O' \cap \{o\} = \emptyset$, $E[p'] = p' \parallel q$ is defined. Therefore also $E[p] = p \parallel q$ must be defined. But then $O \cap \{o\} = \emptyset$, a contradiction. Hence $O \subseteq O'$.

($O' \subseteq O$) Assume by contradiction that there exists $o \in O'$ such that $o \notin O$. Consider the agent $q = (\{o\}, \emptyset)$ and the context $E = \mathit{proj}(I')(\beta \| q)$. Then, since $I' \cap O' = \emptyset$ and $o \in O'$, $p' \| q = ((I' \cup \{o\}) - O', O') = (I', O')$. Therefore, since $I' \subseteq I'$, $E[p'] = \mathit{proj}(I')(p' \parallel q)$ is defined. Therefore also $E[p] = \mathit{proj}(I')(p \parallel q)$ must be defined. However, since $I \cap O = \emptyset$ and $o \notin O$, $p \parallel q = ((I \cup \{o\}) - O, O) =$

$(I \cup \{o\}, O)$. In addition, $I \cup \{o\} \not\subseteq I'$, since $o \in O'$ implies $o \notin I'$, since $I' \cap O' = \emptyset$. Hence $\mathit{proj}(I')(p \parallel q)$ is not defined, a contradiction. Therefore $O' \subseteq O$.

□

Theorem 3.53. Let $p = (I, O)$ and $p' = (I', O')$ be IO agents. Then $p \preceq_D^{\parallel} p'$ if and only if $O \subseteq O'$.

Proof: For the forward direction, assume $p \preceq_D^{\parallel} p'$. Then for all agents q, if $p' \parallel q$ is defined, then also $p \parallel q$ is defined. Assume by contradiction that there exists $o \in O$ such that $o \notin O'$. Consider the agent $q = (\emptyset, \{o\})$. Then, since $O' \cap \{o\} = \emptyset$, $p' \parallel q$ is defined. Therefore, by definition of conformance, also $p \parallel q$ must be defined. But then $O \cap \{o\} = \emptyset$, a contradiction. Hence $O \subseteq O'$.

For the reverse direction, assume $O \subseteq O'$, and let $q = (I_q, O_q)$ be such that $p' \parallel q$ is defined. Then $O' \cap O_q = \emptyset$. Since $O \subseteq O'$, also $O \cap O_q = \emptyset$. Therefore $p \parallel q$ is defined. □

As expected, the order induced by D relative to composition does not make the operators $\top$-monotonic. The above results also confirm that $\preceq$ is the weakest order such that the operators are $\top$-monotonic.

Example 3.54 (Dill's IO Agent Algebra). Consider Dill's IO agent algebra $\mathcal{Q}$ defined in example 2.11 and example 2.32. The algebra is an ordered agent algebra if and only if $p \preceq p'$ corresponds to $p = p'$. Hence the only possible order is also the weakest possible order. Therefore $\mathcal{Q} = \mathcal{Q}.\mathit{conf}(D)$.

It is difficult however to characterize the order with conformance relative to composition. The following theorems characterize the conformance orders relative to composition induced by several conformance sets, and show that they do not correspond to the algebra's order.

Theorem 3.55. Let $G = \{(I, O) : I = \emptyset\}$ and let $p = (I, O)$ and $p' = (I', O')$ be agents. Then $p \preceq_G^{\parallel} p'$ if and only if $I \subseteq I'$ and $O = O'$.

Proof: The proof is the same as lemma 3.51. □

Theorem 3.56. Let $G = D$ and let $p = (I, O)$ and $p' = (I', O')$ be agents. Then $p \preceq_D^{\parallel} p'$ if and only if $O \subseteq O'$.

Proof: The proof is the same as theorem 3.53. □

Theorem 3.57. Let $G = \{(\emptyset, \mathcal{Q}.\mathcal{A})\}$ and let $p = (I, O)$ and $p' = (I', O')$ be agents. Then $p \preceq_G^{||} p'$ if and only if $O = O'$.

Proof: For the forward direction, assume $p \preceq_G^{||} p'$. Consider the agent $q = (O', \mathcal{A} - O')$. Then $p' \parallel q = (\emptyset, \mathcal{A}) \in G$. Hence also $p \parallel q$ must be defined, and therefore $O \cap (\mathcal{A} - O') = \emptyset$. But then $O \subseteq O'$. In addition $p \parallel q \in G$, and therefore $O \cup (\mathcal{A} - O') = \mathcal{A}$. But then $O \supseteq O'$. Hence $O = O'$.

For the reverse direction, assume $O = O'$. Let $q = (I_q, O_q)$ be an agent. If $p' \parallel q$ is defined, then $O' \cap O_q = \emptyset$. But then also $O \cap O_q = \emptyset$, and therefore also $p \parallel q$ is defined. In addition, if $p' \parallel q \in G$ then it must be $O' \cap O_q = \emptyset$ (for the composition to be defined) and $O' \cup O_q = \mathcal{A}$, and therefore $O_q = \mathcal{A} - O'$. Hence also $p \parallel q \in G$. Therefore $p \preceq_G^{||} p'$. □

Example 3.58 (Typed IO Agent Algebra). Consider the Typed IO agent algebra $\mathcal{Q}$ defined in example 2.12 with the order defined in example 2.34. We would now like to characterize the order in terms of a conformance set. This can be done if we choose G to be the set of agents p such that $\mathit{inputs}(p) = \emptyset$.

Theorem 3.59. Let p and p' be Typed IO agents. Then the following three statements are equivalent:

1. $p \preceq p'$.
2. $p \preceq_G p'$.
3. $p \preceq_G^{||} p'$.

Proof: We already know that $1 \Rightarrow 2$ (by theorem 3.34, since G is downward closed) and that $2 \Rightarrow 3$ (by lemma 3.41). The remaining implication is proved below.

Lemma 3.60. ($3 \Rightarrow 1$): Let p and p' be agents such that for all agents q, if $p' \parallel q \in G$ then $p \parallel q \in G$. Then $p \preceq p'$.

Proof: It is easy to adapt the proof of lemma 3.51 to show that $\mathit{inputs}(p) \subseteq \mathit{inputs}(p')$ and that $\mathit{outputs}(p) = \mathit{outputs}(p')$. To prove the rest of the theorem, let $q = f_q$

be the agent such that for all $a \in \mathcal{Q}.\mathcal{A}$

$$f_q(a) = \begin{cases} (c_O, v) & \text{if } f'(a) = (c_I, v) \\ (c_I, v) & \text{if } f'(a) = (c_O, v) \\ c_U & \text{otherwise} \end{cases}$$

so that $\mathit{inputs}(q) = \mathit{outputs}(p')$ and $\mathit{outputs}(q) = \mathit{inputs}(p')$. Then clearly $p' \parallel q$ is defined, and by definition of $\parallel$, $p' \parallel q \in G$. Thus, by hypothesis, also $p \parallel q \in G$. Let now $a \in \mathcal{Q}.\mathcal{A}$. If $a \in \mathit{inputs}(p)$, then $a \in \mathit{inputs}(p')$ and $a \in \mathit{outputs}(q)$. Since $p \| q$ is defined, then $f_q(a).v \subseteq f(a).v$, and thus $f'(a).v \subseteq f(a).v$. Similarly, if $a \in \mathit{outputs}(p)$, then $a \in \mathit{outputs}(p')$ and $a \in \mathit{inputs}(q)$. Since $p \parallel q$ is defined, then $f(a).v \subseteq f_q(a).v$, and thus $f(a).v \subseteq f'(a).v$. Thus $p \preceq p'$. □

□

3.3 Mirrors

In this section we address the problem of checking in an ordered agent algebra whether two agents are related by the order. If the algebra has a G-conformance order, then the problem reduces to verifying the condition for conformance. This problem however is rather expensive, since it requires considering all possible contexts. When conformance corresponds to conformance relative to composition then we need only check contexts that consist of parallel compositions with other agents. We define an *environment* of an agent to be a composition context. In this section we show how, in certain cases, it is possible to construct for each agent a single environment that determines the order. We call this environment the *mirror* of an agent.

Definition 3.61 (Mirror Function). Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents of $\mathcal{Q}$. Then, *$\mathcal{Q}$ has a mirror function relative to G* if and only if

1. $\mathcal{Q}.\mathit{mirror}$ (which we may simply write as "*mirror*" when there is no ambiguity about what agent algebra is being considered) is a partial function from D to D,

2. $\mathit{mirror}(p)$ is defined if and only if there exists q such that $p \parallel q \in G$,

3. $p \preceq p'$ if and only if either $\mathit{mirror}(p')$ is undefined or $p \parallel \mathit{mirror}(p') \in G$.

When an ordered agent algebra $\mathcal{Q}$ has a mirror function relative to some set of agents G, then we can verify that $p \preceq p'$ by simply looking at the composition $p \parallel \mathit{mirror}(p')$. Often, computing the mirror and the composition, and verifying the membership in G, is computationally less expensive than checking that $p \preceq p'$ directly.

In the rest of this section we will explore the consequences of having a mirror function. Later, we will explore necessary and sufficient conditions for an ordered agent algebra to have a mirror function.

Lemma 3.62. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. For all agents p, if $\mathit{mirror}(p)$ is defined, then $p \parallel \mathit{mirror}(p) \in G$.

Proof: Since $\preceq$ is reflexive, $p \preceq p$. By definition 3.61, this implies $\mathit{mirror}(p)$ is undefined or $p \parallel \mathit{mirror}(p) \in G$. □

Theorem 3.63. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. For all agents p, if $\mathit{mirror}(p)$ is defined, then $\mathit{mirror}^2(p)$ is also defined.

Proof: Assume $\mathit{mirror}(p)$ is defined. By lemma 3.62, $p \parallel \mathit{mirror}(p) \in G$. This implies that there exists a p' (namely p) such that $p' \parallel \mathit{mirror}(p) \in G$. By definition 3.61, this implies that $\mathit{mirror}^2(p)$ is defined. □

Corollary 3.64. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. For all agents p, if $\mathit{mirror}(p)$ is defined, then $\mathit{mirror}^n(p)$ is also defined, for any positive integer n.

Proof: By induction on n. □

Lemma 3.65. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. Let p and q be agents such that $\mathit{mirror}(p)$ and $\mathit{mirror}(q)$ are both defined. Then,

$$\mathit{mirror}(p) \preceq q \Leftrightarrow \mathit{mirror}(q) \preceq p.$$

Proof: The proof is composed of the following series of double implications:

$\mathit{mirror}(p) \preceq q$

by definition 3.61, since $\mathit{mirror}(q)$ is defined

$\Leftrightarrow \quad \mathit{mirror}(p) \parallel \mathit{mirror}(q) \in G$

since $\parallel$ is commutative by A7

$\Leftrightarrow \quad \mathit{mirror}(q) \parallel \mathit{mirror}(p) \in G$

by definition 3.61

$\Leftrightarrow \quad \mathit{mirror}(q) \preceq p$

□

The mirror is a reflection of an agent with respect to the order and the conformance set. As expected, two such reflections will bring us back to the original starting point.

Theorem 3.66. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. Let p be an agent and assume $\mathit{mirror}(p)$ is defined. Then $\mathit{mirror}^2(p)$ is defined and

$$p \approx \mathit{mirror}^2(p).$$

Proof: It follows from corollary 3.64 that $\mathit{mirror}^2(p)$ is defined and $\mathit{mirror}^3(p)$ is defined. By definition 2.22, it is sufficient to show that $\mathit{mirror}^2(p) \preceq p$ and $p \preceq \mathit{mirror}^2(p)$.

Lemma 3.67. $\mathit{mirror}^2(p) \preceq p$.

Proof: $\mathit{mirror}(p) \preceq \mathit{mirror}(p)$, since $\preceq$ is reflexive. Thus, by lemma 3.65, $\mathit{mirror}^2(p) \preceq p$. □

Lemma 3.68. $p \preceq \mathit{mirror}^2(p)$.

Proof: $p \preceq p$, since $\preceq$ is reflexive. We complete the proof with the following chain of implications.

$p \preceq p$

by definition 3.61, since $\mathit{mirror}(p)$ is defined

$\Leftrightarrow \quad p \parallel \mathit{mirror}(p) \in G$

by lemma 3.67 and corollary 3.33

$\Rightarrow \quad p \parallel \mathit{mirror}^3(p) \in G$

by definition 3.61

$\Leftrightarrow \quad p \preceq \mathit{mirror}^2(p).$

□

□

A mirror function inverts the order relationships that exists between its arguments.

Theorem 3.69. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. Let p and q be agents such that $mirror(p)$ and $mirror(q)$ are defined. Then,

$$p \preceq q \Leftrightarrow mirror(q) \preceq mirror(p).$$

Proof: By corollary 3.64 we know that $mirror^2(q)$ is defined. By applying lemma 3.65 to q and $mirror(p)$, we get

$$mirror^2(p) \preceq q. \Leftrightarrow mirror(q) \preceq mirror(p)$$

By theorem 3.66, we know that $q \approx mirror^2(q)$. Thus

$$mirror^2(p) \preceq q \Leftrightarrow p \preceq q.$$

Together, these two facts imply the desired result. □

The next result shows that order equivalence is preserved by the application of the mirror function, and, at the same time, that the mirror function is one-to-one on the equivalence classes induced by the preorder.

Corollary 3.70. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. Let p and q be agents such that $mirror(p)$ and $mirror(q)$ are defined. Then,

$$p \approx q \Leftrightarrow mirror(q) \approx mirror(p).$$

If the agent algebra is partially ordered, then the mirror function is one-to-one on the agents themselves.

Corollary 3.71. Let $\mathcal{Q}$ be a partially ordered agent algebra with a mirror function relative to G. Let p and q be agents such that $mirror(p)$ and $mirror(q)$ are defined. Then,

$$p = q \;\Leftrightarrow\; mirror(p) = mirror(q).$$

Since mirrors reduce the problem of verifying conformance to a single composition environment, it is not surprising that their existence is related to G-conformance relative to composition. In fact, the mirror of an agent has an exact characterization in terms of the G-conformance order relative to composition and the greatest element of a certain set of agents.

Let G be a conformance set and let p and q be agents. If $p \parallel q \in G$ then we say that q is compatible (or G-compatible if we want to emphasize the conformance set) with p. We call the set of agents that are compatible with p the *compatibility set* of p.

Definition 3.72 (Compatibility Set). Let $\mathcal{Q}$ be an ordered agent algebra and G a downward closed set of agents. The *G-compatibility set* of an agent p, written $cmp(p)$, is defined as follows:

$$cmp(p) = \{\, q : p \parallel q \in G\}$$

If two agents are order equivalent, then their compatibility set is the same.

Lemma 3.73. Let $\mathcal{Q}$ be an ordered agent algebra and G a downward closed set of agents. Let p_1 and p_2 be agents such that $p_1 \approx p_2$. Then $cmp(p_1) = cmp(p_2)$.

Proof: We show that the compatibility sets are contained into each other. To show that $cmp(p_1) \subseteq cmp(p_2)$, let $q \in cmp(p_1)$ be an agent compatible with p_1. Then,

$$q \in cmp(p_1)$$

by definition 3.72,

$$\Leftrightarrow \quad p_1 \parallel q \in G$$

since $p_1 \approx p_2$, by corollary 3.33,

$$\Rightarrow \quad p_2 \parallel q \in G$$

by definition 3.72,

$$\Leftrightarrow \quad q \in cmp(p_2).$$

Consequently, $cmp(p_1) \subseteq cmp(p_2)$. The proof that $cmp(p_2) \subseteq cmp(p_1)$ is analogous. $\square$

The compatibility set gets larger as the agents are more refined according to the order of the algebra, as shown in the next theorem.

Lemma 3.74. Let $\mathcal{Q}$ be an ordered agent algebra and G a downward closed set of agents. Let p and p' be agents such that $p \preceq p'$. Then

$$cmp(p') \subseteq cmp(p).$$

Proof: We show that if $q \in cmp(p')$, then $q \in cmp(p)$. The proof consists of the following series of implications.

$q \in \mathit{cmp}(p')$

by definition 3.72

$\Leftrightarrow \quad p' \parallel q \in G$

since $\parallel$ is $\top$-monotonic and $p \preceq p'$

$\Rightarrow \quad p \parallel q \preceq p' \parallel q$

since G is downward closed

$\Rightarrow \quad p \parallel q \in G$

by definition 3.72

$\Leftrightarrow \quad q \in \mathit{cmp}(p).$

□

When an agent algebra has a G-conformance order relative to composition, the order is determined by the compatibility set of each agent.

Lemma 3.75. Let $\mathcal{Q}$ be an ordered agent algebra with a G-conformance order relative to composition. Then for all agents p and p',

$$p \preceq p' \Leftrightarrow p \parallel \mathit{cmp}(p') \subseteq G,$$

where $\parallel$ has been naturally extended to sets.

Proof: The result follows directly from definition 3.40. □

Since the operators of an ordered agent algebra are $\top$-monotonic, the maximal elements of the compatibility set are sufficient to completely determine the order.

Lemma 3.76. Let $\mathcal{Q}$ be an ordered agent algebra with a G-conformance order relative to composition. Then for all agents p and p',

$$p \preceq p' \Leftrightarrow \text{for all } q \text{ such that } q \text{ is maximal in } \mathit{cmp}(p), p \parallel q \in G.$$

Proof: The forward implication is simply a special case of lemma 3.75.

For the reverse implication, let $q \in \mathit{cmp}(p')$ be an agent. Then there exists $q' \in \mathit{cmp}(p')$ such that q' is maximal and $q \preceq q'$. By hypothesis, $p \parallel q' \in G$. Note that G is downward closed, since $\mathcal{Q}$ has a G-conformance order relative to composition. Hence, since $\|$ is $\top$-monotonic and G is downward closed, also $p \parallel q \in G$. Therefore $p \parallel \mathit{cmp}(p') \subseteq G$. The desired result then follows from lemma 3.75. □

We often denote the set of maximal elements of $\mathit{cmp}(p)$ as $\mathit{maxcmp}(p)$.

Lemma 3.76 suggests that the mirror of an agent should be found among the maximal elements of the compatibility set. In fact, since the mirror alone is sufficient to determine the order, it suggests that the mirror should be the greatest element of the compatibility set. In the following we will make the relationship between the mirror and the compatibility set more precise.

Theorem 3.77. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. If $p \parallel q \in G$, then $q \preceq \mathit{mirror}(p)$.

Proof: The proof is composed of the following implications:

$p \parallel q \in G$

by definition 3.61

$\Leftrightarrow \quad \mathit{mirror}(p)$ is defined

by lemma 3.67

$\Rightarrow \quad \mathit{mirror}^2(p) \preceq p$

by corollary 3.33

$\Rightarrow \quad \mathit{mirror}^2(p) \parallel q \in G$

by definition 2.6 (commutativity)

$\Leftrightarrow \quad q \parallel \mathit{mirror}^2(p) \in G$

by definition 3.61

$\Leftrightarrow \quad q \preceq \mathit{mirror}(p)$

□

When an agent algebra has a mirror function relative to a conformance set G, then it has a G-conformance order relative to composition and a G-conformance order, as shown by the next results.

Theorem 3.78. Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents. If $\mathcal{Q}$ has a mirror function relative to G, then $\mathcal{Q}$ has a G-conformance order relative to composition.

Proof: We must show that for all agents p and p', $p \preceq p'$ if and only $p \preceq_G^{\|} p'$.

The forward implication follows from theorem 3.34 and lemma 3.41 since G is downward closed.

For the reverse implication we consider two cases. Assume $\mathit{mirror}(p')$ is not defined. Then, by definition 3.61, $p \preceq p'$.

Assume $\mathit{mirror}(p')$ is defined. Then, by lemma 3.62, $p' \| \mathit{mirror}(p') \in G$. Then, since $p \preceq_G^{\|} p'$, also $p \| \mathit{mirror}(p') \in G$. Therefore, by definition 3.61, $p \preceq p'$. □

Corollary 3.79. Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents. If $\mathcal{Q}$ has a mirror function relative to G, $\mathcal{Q}$ has a G-conformance order.

Proof: The result follows from theorem 3.78 and theorem 3.42. □

To put it another way, when an algebra $\mathcal{Q}$ has a mirror function relative to G, both G-conformance and G-conformance relative to composition characterize the order.

We can now completely characterize the mirror function in terms of conformance and the compatibility sets.

Theorem 3.80 (Mirror Characterization). Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents. Then the following two statement are equivalent:

1. $\mathcal{Q}$ has a mirror function relative to G.

2. $\mathcal{Q}$ has a G-conformance order relative to composition, and for all agents p', $\mathit{cmp}(p')$ is either empty or if it is not empty it has a greatest element.

Proof: Assume $\mathcal{Q}$ has a mirror function relative to the set G. Then, by theorem 3.78, $\mathcal{Q}$ has a G-conformance order relative to composition. In addition, let p' be an agent. If $\mathit{mirror}(p')$ is undefined, then, by definition 3.61, $\mathit{cmp}(p')$ is empty. Otherwise, if $\mathit{mirror}(p')$ is defined, then, by definition 3.61, $\mathit{cmp}(p')$ is not empty, and, by theorem 3.77, $\mathit{mirror}(p')$ is its greatest element.

Conversely, assume $\mathcal{Q}$ has a G-conformance order relative to composition, and for all agents p', $\mathit{cmp}(p')$ is either empty or if it is not empty it has a greatest element. We show that the function

$$\mathit{mirror}(p') = \begin{cases} \max(\mathit{cmp}(p')) & \text{if } \mathit{cmp}(p') \neq \emptyset \\ \text{undefined} & \text{if } \mathit{cmp}(p') = \emptyset \end{cases}$$

is a mirror function relative to G.

Clearly *mirror* is a partial function, and $\mathit{mirror}(p')$ is defined if and only if there exists an agent q such that $p' \parallel q \in G$. It remains to be shown that for all agents p and p', $p \preceq p'$ if and only if $\mathit{mirror}(p')$ is undefined or $p \parallel \mathit{mirror}(p') \in G$.

Assume $p \preceq p'$. If $\mathit{mirror}(p')$ is undefined we are done. Assume $\mathit{mirror}(p')$ is defined. Since $\mathit{mirror}(p') \in \mathit{cmp}(p')$, $p' \parallel \mathit{mirror}(p') \in G$. But $\mathcal{Q}$ has a G-conformance order relative to composition, hence $p \preceq p'$ if and only if for all q, if $p' \parallel q \in G$ then $p \parallel q \in G$. Therefore $p \parallel \mathit{mirror}(p') \in G$, since by hypothesis $p \preceq p'$.

Conversely assume $\mathit{mirror}(p')$ is undefined or $p \parallel \mathit{mirror}(p') \in G$. If $\mathit{mirror}(p')$ is undefined, then $\mathit{cmp}(p') = \emptyset$, and therefore for all agents q, if $p' \parallel q \in G$ then $p \parallel q \in G$ vacuously. Hence $p \preceq_G^{\parallel} p'$, and since $\mathcal{Q}$ has a G-conformance order relative to composition, also $p \preceq p'$.

On the other hand, assume $\mathit{mirror}(p')$ is defined and $p \parallel \mathit{mirror}(p') \in G$. Let q be an agent such that $p' \parallel q \in G$. Then, by our definition of $\mathit{mirror}(p')$, $q \preceq \mathit{mirror}(p')$, since $\mathit{mirror}(p')$ is the greatest compatible agent. Then $p \parallel q \in G$, since $p \parallel \mathit{mirror}(p') \in G$, $q \preceq \mathit{mirror}(p')$, $\parallel$ is $\top$-monotonic and G is downward closed. Hence $p \preceq_G^{\parallel} p'$, and since $\mathcal{Q}$ has a G-conformance order relative to composition, also $p \preceq p'$. □

These results show that the mirror of an agent corresponds to the greatest element of the compatibility set. For general preordered agent algebra, the compatibility set may have several different greatest elements. In that case there is some flexibility in the choice of the mirror function.

Corollary 3.81. Let $\mathcal{Q}$ be an ordered agent algebra, and let mirror_1 and mirror_2 be partial functions from $\mathcal{Q}.D$ to $\mathcal{Q}.D$ such that for all agents $p \in \mathcal{Q}.D$,

$$\mathit{mirror}_1(p) \approx \mathit{mirror}_2(p)$$

(in particular, $\mathit{mirror}_1(p)$ is defined if and only if $\mathit{mirror}_2(p)$ is defined). Then, mirror_1 is a mirror function for $\mathcal{Q}$ relative to some conformance set G if and only if mirror_2 is a mirror function for $\mathcal{Q}$ relative to G.

Proof: The result follows directly from the definition of mirror function (def. 3.61) and corollary 3.33. □

If the algebra is partially ordered (i.e., the order is antisymmetric), the greatest element is unique. Hence, if a mirror function exists, it is uniquely determined.

Theorem 3.82. Let $\mathcal{Q}$ be a partially ordered agent algebra. If $\mathcal{Q}$ has a mirror function relative to G, then the mirror function is uniquely determined.

Proof: Assume $\mathcal{Q}$ has two mirrors functions $\mathcal{Q}.\mathit{mirror}_1$ and $\mathcal{Q}.\mathit{mirror}_2$. Let p be an agent. By definition 3.61, $\mathit{mirror}_1(p)$ and $\mathit{mirror}_2(p)$ are either both defined or both undefined. If they are both defined, then

$$\begin{aligned}
& p \parallel \mathit{mirror}_1(p) \in G \wedge p \parallel \mathit{mirror}_2(p) \in G \\
& \quad \text{By theorem 3.77} \\
\Rightarrow \quad & \mathit{mirror}_1(p) \preceq \mathit{mirror}_2(p) \wedge \mathit{mirror}_2(p) \preceq \mathit{mirror}_1(p) \\
& \quad \text{by corollary 2.23} \\
\Rightarrow \quad & \mathit{mirror}_1(p) = \mathit{mirror}_2(p)
\end{aligned}$$

Since p was arbitrary, then $\mathcal{Q}.\mathit{mirror}_1 = \mathcal{Q}.\mathit{mirror}_2$. ☐

Perfectly reasonable agent algebras may fail to have a mirror function. The characterization of theorem 3.80 tells us that this may occur for the following two reasons:

- the parallel composition operator is unable to characterize the order of the algebra, i.e., the algebra does not have a conformance order relative to composition, or
- the compatibility set fails to have a greatest element.

In both cases the lack of a mirror function is due to insufficient information in the agent model. The following examples show that by extending the model it is possible to recover a mirror function and a conformance order.

Example 3.83 (Alphabet Algebra). Consider the agent algebra $\mathcal{Q}$ described in example 3.44, and let $G = 2^{\mathcal{A}} - \mathcal{A}$. Recall that $\mathcal{Q}$ has a G-conformance order relative to composition. We now show that $\mathcal{Q}$ has no mirror function relative to G. To do so, we consider the compatibility set of each agent, and then apply theorem 3.80.

Let p be an agent. It is easy to see that the set of agents compatible with p is

$$\mathit{cmp}(p) = \{\, q : \exists a [a \notin q \wedge a \notin p] \}.$$

The maximal elements of the compatibility set are therefore

$$\mathit{maxcmp}(p) = \{\, \mathcal{A} - \{\, a\} : a \notin p \}.$$

Observe that $\mathit{maxcmp}(p)$ is a set of incomparable agents. Thus $\mathit{cmp}(p)$ does not have a greatest element, and therefore, by theorem 3.80, $\mathcal{Q}$ does not have a mirror function relative to G. Because G is the only set of agents that characterizes the order relative to composition, $\mathcal{Q}$ has no mirror function relative to any G.

Example 3.84 (Locked Alphabet Algebra). In this example we present an extension of example 3.83 and we show that by adding extra information to the model it is possible to characterize the order with a mirror function.

The locked alphabet algebra $\mathcal{Q}$ is defined as follows:

- Agents are of the form $p = (A, L)$ where A and L are disjoint subsets of $\mathcal{Q}.\mathcal{A}$. The alphabet of p is $\alpha(p) = A \cup L$.
- $\mathit{rename}(r)(p)$ is defined whenever $\alpha(p) \subseteq \mathit{dom}(r)$. In that case $\mathit{rename}(r)(p) = (r(A), r(L))$, where r is naturally extended to sets.
- $\mathit{proj}(B)(p) = (A \cap B, L \cap B)$.
- $p_1 \parallel p_2$ is defined whenever $L_1 \cap L_2 = \emptyset$, $A_1 \cap L_2 = \emptyset$ and $A_2 \cap L_1 = \emptyset$. In that case
$$p_1 \parallel p_2 = (A_1 \cup A_2, L_1 \cup L_2).$$

The additional set of signals L is used by an agent p to indicate that no agent q can compose with p if q uses signals in L.

Theorem 3.85. Let $\preceq$ be an order for $\mathcal{Q}$ such that rename, proj and $\parallel$ are $\top$-monotonic. Let $p = (A, L)$ and $p' = (A', L')$ be two agents. Then $p \preceq p'$ only if $A \subseteq A' \cup L'$ and $L \subseteq L'$.

Proof: Consider the agent $q = (A', \mathcal{A} - (A' \cup L'))$. Clearly, $p' \parallel q$ is defined, since $L' \cap \mathcal{A} - (A' \cup L') = \emptyset$ and $A' \cap L' = \emptyset$. Therefore, since $\parallel$ is $\top$-monotonic and $p \preceq p'$, also $p \parallel q$ is defined. Hence:

$$\begin{aligned} & L \cap \mathcal{A} - (A' \cup L') = \emptyset \ \wedge\ L \cap A' = \emptyset \ \wedge\ A \cap \mathcal{A} - (A' \cup L') = \emptyset \\ \Rightarrow\ \ & L \subseteq A' \cup L' \ \wedge\ L \cap A' = \emptyset \ \wedge\ A \subseteq A' \cup L' \\ \Rightarrow\ \ & A \subseteq A' \cup L' \ \wedge\ L \subseteq L'. \end{aligned}$$

The requirements of rename and proj are subsumed by those of $\parallel$. $\square$

We will consider the order such that $p \preceq p'$ if and only if $A \subseteq A' \cup L'$ and $L \subseteq L'$. The proof that the operators are $\top$-monotonic is left to the reader.

Note that the subset of agents $P = \{ (A, L) : L = \emptyset \}$ is closed under the operations and thus constitutes a subalgebra $\mathcal{P}$ of $\mathcal{Q}$. It is easy to show that $\mathcal{P}$ is isomorphic to the Alphabet Algebra of example 3.83. By extension, we consider the Locked Alphabet Algebra a superalgebra of the Alphabet Algebra.

The order can be characterized as a G-conformance order relative to composition where $G = \mathcal{D}$ includes all the agents of the algebra. Clearly G is downward closed relative to $\preceq$.

Let now $p' = (A', L')$ be an agent, and consider the set of agents $q = (A, L)$ that are compatible with p'. Since G is the set of all agents, an agent q is compatible with p' if and only if $q \parallel p'$ is defined, that is

$$L \cap L' = \emptyset \;\wedge\; A \cap L' = \emptyset \;\wedge\; A' \cap L = \emptyset$$

which translates to

$$\mathit{cmp}(p') = \{ (A, L) : A \cap L = \emptyset \wedge A \subseteq \mathcal{A} - L' \wedge L \subseteq \mathcal{A} - (A' \cup L') \}.$$

Note that if $A \subseteq \mathcal{A} - L'$ and $L \subseteq \mathcal{A} - (A' \cup L')$, then $A \subseteq A' \cup (\mathcal{A} - (A' \cup L'))$ and $L \subseteq \mathcal{A} - (A' \cup L')$. Therefore, the agent $q = (A', \mathcal{A} - (A' \cup L')$ is the greatest element of $\mathit{cmp}(p')$.

Theorem 3.86. Let $p = (A, L)$ and $p' = (A', L')$ be two agents. Then $p \preceq p'$ if and only if $p \parallel (A', \mathcal{A} - (A' \cup L'))$ is defined.

Therefore $\mathit{mirror}(p') = (A', \mathcal{A} - (A' \cup L'))$ is a mirror function relative to G, and $\mathcal{Q}$ has a G-conformance order relative to composition.

Example 3.87 (IO Agent Algebra). Consider the IO agent algebra $\mathcal{Q}$ described in example 3.48 and let G be the set of agents that have no inputs. Then $\mathcal{Q}$ has a G-conformance order relative to composition. We now show that $\mathcal{Q}$ has no mirror function relative to G. Let $p' = (I', O')$ be an agent. The set of agents compatible with p' is

$$\mathit{cmp}(p') = \{ q = (I_q, O_q) : I_q \subseteq O' \wedge I' \subseteq O_q \subseteq \mathcal{A} - O' \}.$$

The maximal elements of the compatibility set are therefore

$$\mathit{maxcmp}(p') = \{ q = (I_q, O_q) : I_q = O' \wedge I' \subseteq O_q \subseteq \mathcal{A} - O' \}.$$

Since the agents in $\mathit{maxcmp}(p')$ are incomparable, $\mathit{cmp}(p')$ does not have a greatest element, and therefore, by theorem 3.80, $\mathcal{Q}$ does not have a mirror function relative to G.

Notice how every maximal element imposes a particular constraint for an agent to refine another. Let $p = (I, O)$ and $p' = (I', O')$ be two agents. Then the maximal element $q_1 = (O', I')$ characterizes an order (which is not $\top$-monotonic) such that

$$p \preceq p' \Leftrightarrow I \subseteq I' \wedge O \supseteq O' \wedge O \cap I' = \emptyset.$$

On the other hand, the maximal element $q_2 = (O', \mathcal{A} - O')$ characterizes the different order (again not $\top$-monotonic) such that

$$p \preceq p' \Leftrightarrow I \subseteq \mathcal{A} - O' \wedge O \supseteq O' \wedge O \subseteq O'.$$

In other words, q_1 provides the constraint on the inputs, while q_2 constrains the outputs. Note that in this case these two maximal elements are sufficient to characterize the order, which is equal to the intersection of the two orders described.

Example 3.88 (Locked IO Agent Algebra). In this example we present an extension of example 3.87 and we show that by adding extra information to the model it is possible to characterize the order with a mirror function.

The locked IO Agent algebra $\mathcal{Q}$ is defined as follows:

- Agents are of the form $p = (I, O, L)$ where I, O and L are disjoint subsets of $\mathcal{Q}.\mathcal{A}$. The alphabet of p is $\alpha(p) = I \cup O \cup L$.
- $\mathit{rename}(r)(p)$ is defined whenever $\alpha(p) \subseteq \mathit{dom}(r)$. In that case $\mathit{rename}(r)(p) = (r(I), r(O), r(L))$, where r is naturally extended to sets.
- $\mathit{proj}(B)(p)$ is defined whenever $I \subseteq B$. In that case, $\mathit{proj}(B)(p) = (I, O \cap B, L \cap B)$.
- $p_1 \parallel p_2$ is defined whenever $(O_1 \cup L_1) \cap (O_2 \cup L_2) = \emptyset$, $I_1 \cap L_2 = \emptyset$ and $I_2 \cap L_1 = \emptyset$. In that case

$$p_1 \parallel p_2 = ((I_1 \cup I_2) - (O_1 \cup O_2), O_1 \cup O_2, L_1 \cup L_2).$$

The additional set of signals L is used by an agent p to indicate that no agent q can compose with p if q uses signals in L.

Theorem 3.89. Let $\preceq$ be an order for $\mathcal{Q}$ such that *rename*, *proj* and $\|$ are $\top$-monotonic. Then $p \preceq p'$ only if $I \subseteq I'$, $O' \subseteq O \subseteq O' \cup L'$ and $L \subseteq L'$.

Proof: The proof is similar to the proof of theorem 2.30. Let $p = (I, O, L)$ and $p' = (I', O', L')$ be two agents such that $p \preceq p'$. Then consider the agent $q = (O', I', \mathcal{A} - (I' \cup O' \cup L'))$ and deduce the conditions for which $\mathit{rename}(r)(p)$, $\mathit{proj}(B)(p)$ and $p \,\|\, q$ are all defined. □

We will consider the order such that $p \preceq p'$ exactly when $I \subseteq I'$, $O' \subseteq O \subseteq O' \cup L'$ and $L \subseteq L'$.

Theorem 3.90. The functions *rename*, *proj* and $\|$ are $\top$-monotonic with respect to $\preceq$.

Proof: The proof is similar to the proof of theorem 2.31. □

Note that the subset of agents $P = \{\, (I, O, L) : L = \emptyset\}$ is closed under the operations and thus constitutes a subalgebra $\mathcal{P}$ of $\mathcal{Q}$. It is easy to show that $\mathcal{P}$ is isomorphic to the IO Agent Algebra of example 3.87. By extension, we consider the Locked IO Agent Algebra a superalgebra of the IO Agent Algebra.

The order can be characterized as a G-conformance order relative to composition, where $G = \{\, (\emptyset, O, L)\}$ includes all the only the agents with no inputs. Clearly G is downward closed relative to $\preceq$.

Let now $p' = (I', O', L')$ be an agent, and consider the set of agents $q = (I, O, L)$ compatible with p'. We have

$$q \,\|\, p' = ((I \cup I') - (O \cup O'), O \cup O', L \cup L'),$$

with the following conditions for membership in G and for definedness:

$$I \cup I \subseteq O \cup O',$$
$$(O \cup L) \cap (O' \cup L') = \emptyset,$$
$$I \cap L' = \emptyset \wedge L \cap I' = \emptyset \wedge L \cap L' = \emptyset.$$

These conditions imply (since also for each agent, I, O and L must be disjoint) that

$$\begin{array}{ccccc} \emptyset & \subseteq & I & \subseteq & O' \\ I' & \subseteq & O & \subseteq & \mathcal{A} - (O' \cup L') \\ \emptyset & \subseteq & L & \subseteq & \mathcal{A} - (O \cup O' \cup L') \end{array}$$

Notice that the two agents $q_1 = (I, O \cup \{ a\}, L)$, and $q_2 = (I, O, L \cup \{ a\})$ are comparable and $q_1 \preceq q_2$. Therefore the set of compatible agents of p' has a greatest element. It is easy to show that the greatest element is also a mirror function, so that

$$\mathit{mirror}(p') = (O', I', \mathcal{A} - (I' \cup O' \cup L')).$$

Hence, the algebra also has a G-conformance order relative to composition.

In the particular case of the simple IO agents, p' is of the form $p' = (I', O', \emptyset)$. Hence $\mathit{mirror}(p') = (O', I', \mathcal{A} - (I' \cup O'))$. Note how all the maximal elements found in example 3.87 are contained in $\mathit{mirror}(p')$ in the superalgebra. In the superalgebra, however, the compatibility set is extended upwards by agents that converge to a unique greatest element.

Example 3.91 (Dill's IO Agent Algebra). We have seen in example 3.54 that the Dill's IO Agent Algebra does not have a characterization in terms of conformance relative to composition. It is therefore impossible to find a mirror function in this case. We will however reconsider this example when we restrict the order to agents that share the same alphabet, below.

In this section we have seen examples of agent algebras that don't have a mirror function, despite having a G-conformance order relative to composition (see example 3.83 and example 3.87). The solution adopted in those cases consists of augmenting the model with enough information to let a single environment characterize the order. In the next two sections we explore alternative solutions that consist of adding some extra condition to the definition of the mirror function in order to restrict the size of the compatibility set.

3.3.1 Mirrors with Predicates

Let $\mathcal{Q}$ be an ordered agent algebra, and let p' be an agent. If $\mathcal{Q}$ has a G-conformance order relative to composition, then the compatibility set $\mathit{cmp}(p')$ of p' completely characterizes the set of agents p such that $p \preceq p'$ (see lemma 3.75). Each individual agent q in the compatibility set contributes to the characterization of the order by discriminating among two sets: the set of agents p that are compatible with q *do not conform* to p'; and the set of agents p that are compatible with q that *potentially* conform to p'. In other words, each compatible agent has a particular view of the conformance order.

When a mirror function exists, one agent (the greatest element) has an exact view of the conformance order. In that case, the compatibility set of p' is equal to the set of agents that conform to $\mathit{mirror}(p')$, and, vice-versa, the compatibility set of $\mathit{mirror}(p')$ is equal to the set of agents that conform to p'. For an arbitrary element of the compatibility set we can only establish a containment relationship.

Definition 3.92 (Refinement Set). Let $\mathcal{Q}$ be an ordered agent algebra, and let $p' \in \mathcal{Q}.D$. The *refinement set* of p', written $\mathit{ref}(p')$, is the set of agents p such that $p \preceq p'$:

$$\mathit{ref}(p') = \{\, p : p \preceq p' \}.$$

Lemma 3.93. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. Let p' be an agent such that $\mathit{mirror}(p')$ is defined. Then

$$\mathit{ref}(p') = \mathit{cmp}(\mathit{mirror}(p')).$$

Proof: The proof consists of the following series of double implications:

$p \in \mathit{ref}(p')$

by definition 3.92

$\Leftrightarrow \quad p \preceq p'$

by definition 3.61, since $\mathit{mirror}(p')$ is defined

$\Leftrightarrow \quad p \parallel \mathit{mirror}(p') \in G$

by definition 3.72

$\Leftrightarrow \quad p \in \mathit{cmp}(\mathit{mirror}(p')).$

□

Lemma 3.94. Let $\mathcal{Q}$ be an ordered agent algebra with a G-conformance order relative to composition. Let p' be an agent and let $q \in \mathit{cmp}(p')$ be a compatible agent. Then

$$\mathit{ref}(p') \subseteq \mathit{cmp}(q).$$

Proof: The proof consists of the following series of implications:

$p \in \mathit{ref}(p')$

by definition 3.92

$\Leftrightarrow \quad p \preceq p'$

since $\mathcal{Q}$ has a G-conformance order relative to composition

$\Leftrightarrow \quad \forall q, p' \parallel q \in G \Rightarrow p \parallel q \in G$

since $q \in cmp(p')$, $q \parallel p' \in G$, therefore

$\Rightarrow \quad p \parallel q \in G$

by definition 3.72

$\Leftrightarrow \quad p \in cmp(q)$.

$\square$

These two results are represented graphically in figure 3.1 and figure 3.2. Given an agent q

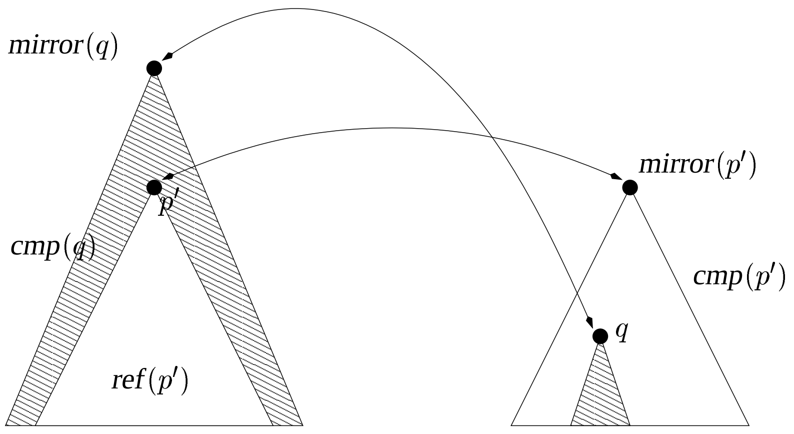

Figure 3.1: Refinement sets and compatibility sets with mirrors

in the compatibility set of p', we call the *discrimination set* of q the set of agents that q discriminates exactly for the purpose of conformance to p'.

Definition 3.95 (Discrimination Set). Let $\mathcal{Q}$ be an ordered agent algebra with a G-conformance order relative to composition. Let p' be an agent and let $q \in cmp(p')$ be a compatible agent. The *discrimination set* of q over p' is the set

$$dis_{p'}(q) = \{ p : p \preceq p' \Leftrightarrow p \parallel q \in G \}.$$

Lemma 3.96 (Discrimination). Let $\mathcal{Q}$ be an ordered agent algebra with a G-conformance order relative to composition. Let p' be an agent and let $q \in cmp(p')$ be a compatible agent. Then

$$dis_{p'}(q) = (\mathcal{Q}.D - cmp(q)) \cup ref(p').$$

Proof: We must show that $p \in (\mathcal{Q}.D - cmp(q)) \cup ref(p')$ if and only if $p \preceq p' \Leftrightarrow p \parallel q \in G$. For the forward direction we consider the following two cases:

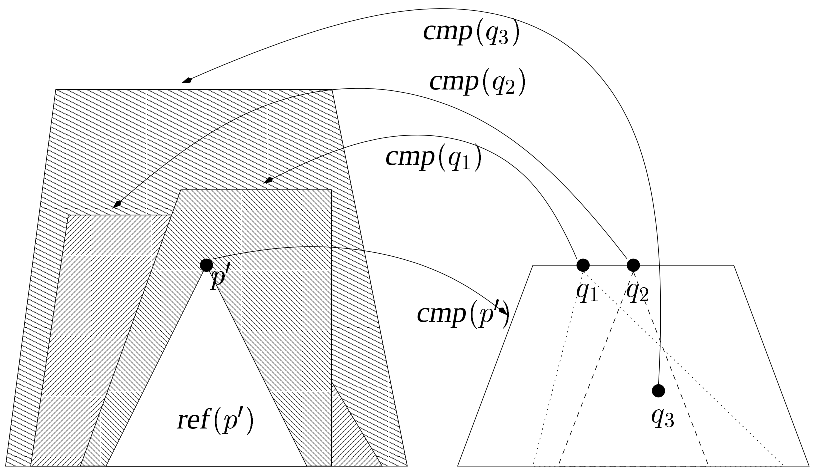

Figure 3.2: Compatibility sets of compatible agents

- If $p \in \mathcal{Q}.D - cmp(q)$, then $p \notin cmp(q)$ and $p \parallel q \notin G$. By lemma 3.94, $ref(p') \subseteq cmp(q)$, therefore $p \notin ref(p')$. Therefore $p \not\preceq p'$. Hence $p \preceq p' \Leftrightarrow p \parallel q \in G$.

- If $p \in ref(p')$, then $p \preceq p'$. By lemma 3.94, $ref(p') \subseteq cmp(q)$, therefore $p \in cmp(q)$. Therefore $p \parallel q \in G$. Hence $p \preceq p' \Leftrightarrow p \parallel q \in G$.

For the reverse direction, let p be an agent such that $p \preceq p' \Leftrightarrow p \parallel q \in G$. We then consider the following two cases:

- If $p \preceq p'$, then $p \in ref(p')$ and therefore $p \in (\mathcal{Q}.D - cmp(q)) \cup ref(p')$.

- If $p \not\preceq p'$, then, by hypothesis, $p \parallel q \notin G$. Hence $p \notin cmp(q)$. Therefore $p \in \mathcal{Q}.D - cmp(q)$, and consequently $p \in (\mathcal{Q}.D - cmp(q)) \cup ref(p')$.

□

Corollary 3.97. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function relative to G. Let p' be an agent such that $mirror(p')$ is defined. Then

$$dis_{p'}(mirror(p')) = \mathcal{Q}.D.$$

Proof: The proof consists of the following equalities:

$$\begin{aligned}
&\mathit{dis}_{p'}(\mathit{mirror}(p')) = \{\, p : p \preceq p' \Leftrightarrow p \parallel \mathit{mirror}(p') \in G\} \\
&\quad \text{by lemma 3.96} \\
&= \quad (\mathcal{Q}.D - \mathit{cmp}(\mathit{mirror}(p'))) \cup \mathit{ref}(p') \\
&\quad \text{by lemma 3.93} \\
&= \quad (\mathcal{Q}.D - \mathit{ref}(p')) \cup \mathit{ref}(p') \\
&= \quad \mathcal{Q}.D.
\end{aligned}$$

□

The above results show that every compatible agent can be used as a "mirror" if the characterization is restricted to its discrimination set. This suggests an extended notion of mirror function, whose applicability is subject to the satisfaction of a predicate.

Definition 3.98 (Mirror Function with Predicate). Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents of $\mathcal{Q}$. For each agent p', let $\mathit{pred}(p') \subseteq \mathcal{Q}.D$ be a predicate over $\mathcal{Q}.D$ such that $\mathit{ref}(p') \subseteq \mathit{pred}(p')$. Then, *$\mathcal{Q}$ has a mirror function with predicate relative to G* if and only if

1. $\mathcal{Q}.\mathit{mirror}$ (which we may simply write as "*mirror*" when there is no ambiguity about what agent algebra is being considered) is a partial function from D to D,

2. $\mathit{mirror}(p)$ is defined if and only if there exists q such that $p \parallel q \in G$,

3. If $p \in \mathit{pred}(p')$, then $p \preceq p'$ if and only if either $\mathit{mirror}(p')$ is undefined or $p \parallel \mathit{mirror}(p') \in G$.

Corollary 3.99. Let $\mathcal{Q}$ be an ordered agent algebra with a mirror function with predicate relative to G. Then, for all agents p and p', the following two statements are equivalent

1. $p \preceq p'$

2. $p \in \mathit{pred}(p')$ and either $\mathit{mirror}(p')$ is undefined or $p \parallel \mathit{mirror}(p') \in G$.

The regular mirror function can be interpreted as a mirror function with predicate by simply setting for all agents p'

$$\mathit{pred}(p') = \mathcal{Q}.D.$$

Hence mirror functions with predicate are more general than the regular mirror functions.

Unfortunately mirror functions with predicate do not enjoy the same characterization in terms of G-conformance relative to composition and greatest elements of the compatibility set (see theorem 3.80). A simple counterexample is obtained by considering the predicate

$$\mathit{pred}(p') = \{\, p : p \preceq p' \}.$$

In this case, any agent of the compatibility set can function as the mirror. This extreme case is, of course, useless, since the complexity of checking membership with the predicate is the same as the complexity of checking conformance. Mirror functions with predicate are therefore most useful when the predicate is relatively easy to check.

The choice of the predicate is guided by the following result.

Theorem 3.100. Let $\mathcal{Q}$ be an ordered agent algebra with a G-conformance order relative to composition. For all agents p', let $\mathit{mirror}(p') \in \mathit{cmp}(p')$ be a compatible agent ($\mathit{mirror}(p')$ is undefined if $\mathit{cmp}(p') = \emptyset$), and let $\mathit{pred}(p') \subseteq \mathcal{Q}.D$ be a predicate. Then the following two conditions are equivalent:

1. mirror is a mirror function with predicate pred relative to G.
2. For all agents p', $\mathit{ref}(p') \subseteq \mathit{pred}(p')$ and if $\mathit{mirror}(p')$ is defined, then $\mathit{pred}(p') \subseteq \mathit{dis}_{p'}(\mathit{mirror}(p'))$.

Proof: For the forward direction, by definition 3.98, for all agents p', $\mathit{ref}(p') \subseteq \mathit{pred}(p')$. Let now p' be an agent such that $\mathit{mirror}(p')$ is defined and let $p \in \mathit{pred}(p')$ be an agent. Then,

$p \in \mathit{pred}(p')$

by definition 3.98, since $\mathcal{Q}$ has a mirror function with predicate relative to G

$\Rightarrow \quad p \preceq p' \Leftrightarrow p \parallel \mathit{mirror}(p') \in G$

by definition 3.95

$\Rightarrow \quad p \in \mathit{dis}_{p'}(\mathit{mirror}(p'))$.

Hence, $\mathit{pred}(p') \subseteq \mathit{dis}_{p'}(\mathit{mirror}(p'))$.

For the reverse direction, assume $\mathit{ref}(p') \subseteq \mathit{pred}(p') \subseteq \mathit{dis}_{p'}(\mathit{mirror}(p'))$. Clearly mirror is a partial function, and $\mathit{mirror}(p)$ is defined if and only if there exists q such that $p \parallel q \in G$. Let now p be an agent such that $p \in \mathit{pred}(p')$. We must show that $p \preceq p'$ if and only if either $\mathit{mirror}(p')$ is undefined or $p \parallel \mathit{mirror}(p') \in G$. We consider two cases.

Assume $\mathit{mirror}(p')$ is undefined. Then, by hypothesis, $\mathit{cmp}(p') = \emptyset$, and therefore, since $\mathcal{Q}$ has a G-conformance order relative to composition, for all agents p, $p \preceq p'$. Therefore, $p \preceq p'$ if and only if $\mathit{mirror}(p')$ is undefined.

Conversely, assume $\mathit{mirror}(p')$ is defined. Then,

$$p \in \mathit{pred}(p')$$

since by hypothesis $\mathit{pred}(p') \subseteq \mathit{dis}_{p'}(\mathit{mirror}(p'))$

$$\Rightarrow \quad p \in \mathit{dis}_{p'}(\mathit{mirror}(p'))$$

by definition 3.95

$$\Rightarrow \quad p \preceq p \Leftrightarrow p \parallel \mathit{mirror}(p') \in G.$$

Hence *mirror* is a mirror function with predicate *pred* relative to G. □

The greater the element in the compatibility set, the larger the discrimination set.

Lemma 3.101. Let $\mathcal{Q}$ be an ordered agent algebra with a G-conformance order relative to composition. Let p' be an agent and let q_1 and q_2 be compatible agents such that $q_1 \preceq q_2$. Then

$$\mathit{dis}_{p'}(q_1) \subseteq \mathit{dis}_{p'}(q_2).$$

Since greater elements have larger discrimination sets, and since $\mathit{pred}(p')$ must be a subset of the discrimination set of $\mathit{mirror}(p')$, it is convenient to choose a maximal element of the compatibility set for $\mathit{mirror}(p')$. In this way, we have the maximum flexibility in choosing a predicate that is computationally easy to check. However, unlike regular mirror functions, the mirror of an agent with predicate is not necessarily a maximal element of the compatibility set.

The following examples show the use of mirror functions with predicate in the cases where a regular mirror function does not exist.

Example 3.102 (IO Agent Algebra). Example 3.87 shows that the IO agent algebra does not have a mirror function relative to G, despite having a G-conformance order relative to composition. In this example we show how to derive a mirror function with predicate.

Let $p' = (I', O')$ be an agent. As shown in example 3.87, the agent $q_1 = (O', I')$ is a maximal element of the compatibility set of p'. We now wish to use q_1 as a mirror of p'

with the use of a predicate. To do so, we compute the discrimination set of q_1:

$$\begin{aligned} \mathit{dis}_{p'}(q_1) &= (\mathcal{Q}.D - \mathit{cmp}(q_1)) \cup \mathit{ref}(p') \\ &= (\mathcal{Q}.D - \{\,(I,O) : I \subseteq I' \wedge O' \subseteq O \subseteq \mathcal{A} - I'\}) \cup \{\,(I,O) : I \subseteq I' \wedge O = O'\} \\ &= \{\,(I,O) : I' \subseteq I \vee O \subseteq O' \vee \mathcal{A} - I' \subseteq O\} \cup \{\,(I,O) : I \subseteq I' \wedge O = O'\} \\ &= \{\,(I,O) : I' \subseteq I \vee O \subseteq O' \vee \mathcal{A} - I' \subseteq O \vee (I \subseteq I' \wedge O = O')\} \end{aligned}$$

Recall that $\mathit{pred}(p')$ must include $\mathit{ref}(p')$. A reasonable choice for $\mathit{pred}(p')$ in this case is the following:

$$\mathit{pred}(p') = \{\,p : \alpha(p) \subseteq \alpha(p')\}.$$

This predicate is easy to check for finite alphabets, and satisfies the condition $\mathit{ref}(p') \subseteq \mathit{pred}(p') \subseteq \mathit{dis}_{p'}(q_1)$. Therefore, by theorem 3.100, $\mathcal{Q}$ has a mirror function with predicate relative to G, where

$$\mathit{mirror}((I,O)) = (O,I), \quad \mathit{pred}(p') = \{\,p : \alpha(p) \subseteq \alpha(p')\}.$$

Note that q_1 is not the only compatible agent that can be used as a mirror with the above predicate. For example, the maximally compatible agent $q_2 = (O', \mathcal{A} - O')$ has the following discrimination set:

$$\begin{aligned} \mathit{dis}_{p'}(q_2) &= (\mathcal{Q}.D - \mathit{cmp}(q_2)) \cup \mathit{ref}(p') \\ &= (\mathcal{Q}.D - \{\,(I,O) : I \subseteq \mathcal{A} - O' \wedge O = O'\}) \cup \{\,(I,O) : I \subseteq I' \wedge O = O'\} \\ &= \{\,(I,O) : \mathcal{A} - O' \subset I \vee O \neq O'\} \cup \{\,(I,O) : I \subseteq I' \wedge O = O'\} \\ &= \{\,(I,O) : \mathcal{A} - O' \subset I \vee O \neq O' \vee (I \subseteq I' \wedge O = O')\} \end{aligned}$$

It is easy to check that the condition $\mathit{ref}(p') \subseteq \mathit{pred}(p') \subseteq \mathit{dis}_{p'}(q_2)$ is satisfied. Hence, $\mathcal{Q}$ has also the following mirror function with predicate:

$$\mathit{mirror}((I,O)) = (O, \mathcal{A} - O), \quad \mathit{pred}(p') = \{\,p : \alpha(p) \subseteq \alpha(p')\}.$$

We have noted how mirror functions with predicate lose the characterization in terms of conformance order that simple mirror functions have. For a restricted case, however, we can reduce a mirror function with predicate to a regular mirror function by extending the model as in example 3.84 and example 3.88. The construction consists of augmenting the model by providing each agent with the information conveyed by the predicate of their mirror. This construction is still somewhat preliminary, as it doesn't guarantee a downward closed conformance set. In addition, the conformance set does not necessarily enjoy the properties that are necessary for applying theorem 3.119 below.

Theorem 3.103. Let $\mathcal{Q}$ be an agent algebra with a mirror function with predicate relative to G, such that $\mathit{pred}(p) = \mathit{pred}(\mathit{mirror}(p))$. If $\overline{G}$ is downward closed, then the agent algebra $\overline{\mathcal{Q}}$ has a mirror function relative to $\overline{G}$, where

- $\overline{\mathcal{Q}}.D = \{\, (p, set) : p \in \mathcal{Q}.D \wedge set \subseteq \mathcal{Q}.D \wedge \mathit{pred}(p) \cap set = \emptyset\}$

- $(p, set) \preceq (p', set')$ if and only if either $\mathcal{Q}.\mathit{mirror}(p')$ is not defined, or, if defined,

  $set \subseteq set'$,
  $\mathit{pred}(p) \subseteq set' \cup \mathit{pred}(p')$ and
  $p \parallel \mathcal{Q}.\mathit{mirror}(p') \in G$.

- $\mathit{proj}(B)((p, set_p)) = (\mathit{proj}(B)(p), \mathit{proj}(B)(set_p))$ if all quantities are defined.

- $\mathit{rename}(r)((p, set_p)) = (\mathit{rename}(r)(p), \mathit{rename}(r)(set_p))$ if all quantities are defined.

- $(p, set_p) \parallel (q, set_q)$ is defined if and only if

  $p \parallel q$ is defined,
  $\mathit{pred}(p) \cap set_q = \emptyset$,
  $\mathit{pred}(q) \cap set_p = \emptyset$ and
  $set_p \cap set_q = \emptyset$.

  In that case

  $$(p, set_p) \parallel (q, set_q) = (p \parallel q, set_p \cup set_q)$$

- $\overline{\mathcal{Q}}.\mathit{mirror}((p, set))$ is defined if and only if $\mathcal{Q}.\mathit{mirror}(p)$ is defined. In that case,

  $$\overline{\mathcal{Q}}.\mathit{mirror}((p, set)) = (\mathcal{Q}.\mathit{mirror}(p), \mathcal{Q}.D - (set \cup \mathit{pred}(p))).$$

- $\overline{G} = G \times 2^{\mathcal{Q}.D}$

Proof: We must show that $\overline{\mathcal{Q}}.\mathit{mirror}$ is a mirror function relative to $\overline{G}$.

Clearly $\overline{\mathcal{Q}}.\mathit{mirror}$ is a partial function from $\overline{\mathcal{Q}}.D$ to $\overline{\mathcal{Q}}.D$. Also, if, for an agent (p, set), $\overline{\mathcal{Q}}.\mathit{mirror}((p, set))$ is defined, then $\overline{\mathcal{Q}}.\mathit{mirror}((p, set)) \parallel (p, set) \in \overline{G}$. Conversely, if $(q, set_q) \parallel (p, set) \in \overline{G}$, then $q \parallel p \in G$, hence $\mathcal{Q}.\mathit{mirror}(p)$ is defined, and therefore $\overline{\mathcal{Q}}.\mathit{mirror}((p, set))$ is defined.

Assume now that $(p, set) \preceq (p', set')$, and assume that $\overline{\mathcal{Q}}.\mathit{mirror}((p', set')) = (\mathcal{Q}.\mathit{mirror}(p'), \mathcal{Q}.D - (set' \cup \mathit{pred}(p')))$ is defined. Then

- By hypothesis $p \parallel \mathcal{Q}.\mathit{mirror}(p') \in G$.
- $\mathit{pred}(p) \cap (\mathcal{Q}.D - (set' \cup \mathit{pred}(p'))) = \emptyset$, since $\mathit{pred}(p) \subseteq set' \cup \mathit{pred}(p')$.
- $\mathit{pred}(\mathcal{Q}.\mathit{mirror}(p')) \cap set = \emptyset$, since by hypothesis $\mathit{pred}(\mathcal{Q}.\mathit{mirror}(p')) = \mathit{pred}(p')$, $\mathit{pred}(p') \cap set' = \emptyset$ and $set \subseteq set'$.
- $set \cap (\mathcal{Q}.D - (set' \cup \mathit{pred}(p'))) = \emptyset$, since $set \subseteq set'$.

Therefore $(p, set) \parallel (\mathcal{Q}.\mathit{mirror}(p'), \mathcal{Q}.D - (set' \cup \mathit{pred}(p'))) \in \overline{G}$.

Conversely, assume $(p, set) \parallel (\mathcal{Q}.\mathit{mirror}(p'), \mathcal{Q}.D - (set' \cup \mathit{pred}(p'))) \in \overline{G}$. Then

- $set \subseteq set' \cup \mathit{pred}(p')$, since $set \cap (\mathcal{Q}.D - (set' \cup \mathit{pred}(p'))) = \emptyset$. In addition, $set \cap \mathit{pred}(p') = \emptyset$, since $set \cap \mathit{pred}(\mathcal{Q}.\mathit{mirror}(p')) = \emptyset$ and $\mathit{pred}(\mathcal{Q}.\mathit{mirror}(p')) = \mathit{pred}(p')$. Therefore, $set \subseteq set'$.
- $\mathit{pred}(p) \subseteq set' \cup \mathit{pred}(p')$, since $\mathit{pred}(p) \cap (\mathcal{Q}.D - (set' \cup \mathit{pred}(p'))) = \emptyset$.
- By hypothesis $p \parallel \mathcal{Q}.\mathit{mirror}(p') \in G$.

Therefore $(p, set) \preceq (p', set')$.

Similarly, if $\overline{\mathcal{Q}}.\mathit{mirror}((p', set'))$ is not defined, then $(p, set) \preceq (p', set')$ if and only if $\overline{\mathcal{Q}}.\mathit{mirror}((p', set'))$ is not defined.

Therefore, by definition 3.61, $\overline{\mathcal{Q}}.\mathit{mirror}$ is a mirror function for $\overline{\mathcal{Q}}$ relative to $\overline{G}$. $\square$

Theorem 3.104. Let $\mathcal{Q}$ and $\overline{\mathcal{Q}}$ be as in theorem 3.103. Then the function $e : \mathcal{Q}.D \mapsto \overline{\mathcal{Q}}.D$ such that for all agents p

$$e(p) = (p, \emptyset)$$

is an embedding.

Proof: It is easy to show that e commutes with the operators of the algebra, that is, for example, that

$$\mathit{proj}(B)(e(p)) = e(\mathit{proj}(B)(p)).$$

To complete the proof we must show that $p \preceq p'$ if and only if $(p, \emptyset) \preceq (p', \emptyset)$.

Let p and p' be such that $p \preceq p'$. If $\mathcal{Q}.\mathit{mirror}(p')$ is not defined, then $(p, \emptyset) \preceq (p', \emptyset)$. Alternatively, assume $\mathcal{Q}.\mathit{mirror}(p')$ is defined. Then, since $p \preceq p'$, $p \in \mathit{pred}(p')$ and $p \parallel \mathcal{Q}.\mathit{mirror}(p') \in G$. Since *pred* is monotonic relative to $\preceq$, $\mathit{pred}(p) \subseteq \mathit{pred}(p')$. Therefore, by definition of $\overline{\mathcal{Q}}$, $(p, \emptyset) \preceq (p', \emptyset)$.

Conversely, assume $(p, \emptyset) \preceq (p', \emptyset)$. If $\mathcal{Q}.\mathit{mirror}(p')$ is not defined then $p \preceq p'$. Alternatively, assume $\mathcal{Q}.\mathit{mirror}(p')$ is defined. Then, by definition of $\overline{\mathcal{Q}}$, $\mathit{pred}(p) \subseteq \mathit{pred}(p')$, and therefore, since $p \in \mathit{pred}(p)$, $p \in \mathit{pred}(p')$. In addition, $p \parallel \mathcal{Q}.\mathit{mirror}(p') \in G$. Therefore, by definition 3.98, also $p \preceq p'$.

Hence, $p \preceq p'$ if and only if $(p, \emptyset) \preceq (p', \emptyset)$. □

Corollary 3.105. Let $\mathcal{Q}$ and $\overline{\mathcal{Q}}$ be as in theorem 3.103 such that $\overline{G}$ is downward closed, and let $(p', \emptyset)$ be an agent of $\overline{\mathcal{Q}}$. If $(p, set) \preceq (p', \emptyset)$, then $set = \emptyset$.

The above results show that if p' is an agent in $\mathcal{Q}$, then the mirror of $(p', \emptyset)$ in $\overline{\mathcal{Q}}$ characterizes exactly the agents p such that $p \preceq p'$.

3.3.2 Mirrors and Subalgebras

In the previous section we have employed a predicate to focus the application of the mirror function to only those agents that the mirror can discriminate. Here we use an alternative approach, and consider only a subset of the agents to reduce the size of the compatibility sets. We choose the subset so that it is downward closed, and closed under parallel composition, thus effectively constructing a subalgebra when the operators of projection and renaming are removed from the signature. Since the compatibility sets are smaller, subalgebras have a greater chance to have a G-conformance order relative to composition and a mirror function.

An example that is particularly useful in practice is the subset of agents that have the same alphabet. In particular, we are interested in studying the conformance order and the corresponding mirror function for algebras whose order satisfies the constraint

$$p \preceq p' \Rightarrow \alpha(p) = \alpha(p').$$

Note that if $\alpha(p) = \alpha(q)$, then $\alpha(p \| q) = \alpha(p) = \alpha(q)$. Therefore the subset of agents with a certain alphabet that satisfy the above constraint is closed under parallel composition and thus constitute a subalgebra of the original agent algebra (restricted to parallel composition only). Note also that the projection and renaming operators have no effect in determining conformance relative to composition and mirror functions. Therefore, the results of the previous sections apply to this restricted case, provided that the necessary restrictions on the alphabet are enforced throughout. Projection and renaming can be used, instead, to transition from one subalgebra to another subalgebra with a different alphabet. This will be useful in solving the local specification synthesis problem under these specific assumptions.

Let $\mathcal{Q}$ be an agent algebra and assume that for all alphabets A, the algebra $\mathcal{P}$ such that $\mathcal{P}.D = \{ p : \alpha(p) = A\}$ is a subalgebra of $\mathcal{Q}$ and is closed under $\preceq$. Assume also that each subalgebra has a G-conformance order relative to composition and a mirror function relative to G. Note that since $\mathcal{P}.D$ must be downward closed for all alphabets, $p \preceq p'$ only if $\alpha(p) = \alpha(p')$. The results obtained in the subalgebras can be rephrased in terms of the original algebra by restricting the definitions of conformance order and mirror function to apply only when the alphabets of the agents involved are the same. Note that we are *not* changing the definition of conformance, but we are simply reflecting the restrictions of the subalgebra in the superalgebra.

Definition 3.106. Let $\mathcal{Q}$ be an agent algebra and let G be a downward closed set of agents of $\mathcal{Q}$. $\mathcal{Q}$ has a *same alphabet G-conformance order relative to composition* if and only if for all agents p and p', $p \preceq p'$ if and only if $\alpha(p) = \alpha(p')$ and for all agents q such that $\alpha(q) = \alpha(p')$, if $p' \parallel q \in G$ then $p \parallel q \in G$.

Definition 3.107. Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents of $\mathcal{Q}$. Then, *$\mathcal{Q}$ has a same alphabet mirror function relative to G* if and only if

1. $\mathcal{Q}.\mathit{mirror}$ is a partial function from D to D,
2. $\mathit{mirror}(p)$ is defined if and only if there exists q such that $\alpha(q) = \alpha(p)$ and $p \parallel q \in G$,
3. $p \preceq p'$ if and only if $\alpha(p) = \alpha(p')$ and either $\mathit{mirror}(p')$ is undefined or $p \parallel \mathit{mirror}(p') \in G$.

The additional conditions in these definitions consistently restrict the alphabets of the agents involved to the alphabet of the agent for which we are considering the mirror.

In particular we are interested in the characterization of the mirror in terms of the greatest element of the compatibility set and of conformance relative to composition.

Definition 3.108 (Compatibility Set). Let $\mathcal{Q}$ be an ordered agent algebra and G a downward closed set of agents. The *alphabet invariant G-compatibility set* of an agent p, written $\mathit{cmp}(p)$, is defined as follows:

$$\mathit{cmp}(p) = \{\, q : \alpha(q) = \alpha(p) \wedge p \parallel q \in G\}$$

Theorem 3.109. Let $\mathcal{Q}$ be an ordered agent algebra and let G be a downward closed set of agents. Then the following two statement are equivalent:

1. $\mathcal{Q}$ has an alphabet invariant mirror function relative to G.

2. $\mathcal{Q}$ has an alphabet invariant G-conformance order relative to composition, and for all agents p', $\mathit{cmp}(p')$ is either empty or if it is not empty it has a greatest element.

These definitions and results apply, for example, to Dill's trace structure algebra [34] and can be applied to any model in which substitutability in defined only for agents that share the same interface. These notion can however be generalized to any arbitrary equivalence relation that partitions the sets of agents into equivalence classes that are closed under composition and under the agent ordering. We omit the details of this generalization.

3.3.3 Construction of Algebras

In this section we explore conformance and mirrors for the direct product of algebras and for subalgebras. We begin by showing that if two agent algebras have a conformance order, then the agent order in their product is weaker than the corresponding conformance order. We also show that the product does have a conformance order, in case the algebras have a mirror function, and the mirror is defined for all agents.

Theorem 3.110. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be agent algebras with a G_1 and G_2-conformance order, respectively. Let $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$ be the direct product (definition 2.13) of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ and let $G = G_1 \times G_2$. Then for all $p, p' \in \mathcal{Q}.D$, if $p \preceq_{\mathcal{Q}} p'$ then for all expression contexts E, if $E[p'] \in G$ then $E[p] \in G$.

Proof: Let $p = \langle p_1, p_2\rangle$ and $p' = \langle p'_1, p'_2\rangle$ be agents such that $p \preceq p'$. The proof consists of the following series of implications:

$p \preceq p'$

by definition 2.13

$\Leftrightarrow \quad p_1 \preceq_{\mathcal{Q}_1} p_1' \wedge p_2 \preceq_{\mathcal{Q}_2} p_2'$

by hypothesis

$\Leftrightarrow \quad (\forall E, E[p_1'] \in G_1 \Rightarrow E[p_1] \in G_1) \wedge (\forall E, E[p_2'] \in G_2 \Rightarrow E[p_2] \in G_2)$

by definition 2.13

$\Rightarrow \quad \forall E, E[p'] \in G \Rightarrow E[p] \in G.$

□

Unfortunately the reverse of the last implication in the proof above does not hold, that is $\mathcal{Q}$ does not necessarily have a G-conformance order. This is because a context E may be defined for an agent p_1, while it may not be defined for the pair $\langle p_1, p_2 \rangle$. However, the result holds in the presence of mirror functions, when the mirror function is always defined. In that case, in fact, the expression contexts can be reduced to a single environment, and the difficulty above disappears.

Theorem 3.111. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be agent algebras with a mirror function relative to G_1 and G_2, respectively, such that for all agents p, $\mathit{mirror}(p)$ is defined. Let $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$ be the direct product (definition 2.13) of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ and let $G = G_1 \times G_2$. Then for all agents $(p_1, p_2) \in \mathcal{Q}.D$, $\mathit{mirror}((p_1, p_2)) = (\mathit{mirror}(p_1), \mathit{mirror}(p_2))$ is a mirror function for $\mathcal{Q}$ relative to G.

Proof: Clearly $\mathcal{Q}.\mathit{mirror}$ is a partial (in fact, total) function. Since mirror is always defined, we must show that for all $p = \langle p_1, p_2 \rangle$ there exists q such that $p \parallel q \in G$.

$\mathit{mirror}(\langle p_1, p_2 \rangle)\downarrow$

by hypothesis

$\Leftrightarrow \quad \mathit{mirror}(p_1)\downarrow \wedge \mathit{mirror}(p_2)\downarrow$

by definition 3.61

$\Leftrightarrow \quad (\exists q_1, p_1 \parallel q_1 \in G_1) \wedge (\exists q_2, p_2 \parallel q_2 \in G_2)$

by definition 2.13

$\Leftrightarrow \quad \exists \langle q_1, q_2 \rangle, \langle p_1, p_2 \rangle \parallel \langle q_1, q_2 \rangle \in G$

$\Leftrightarrow \quad \exists q, p \parallel q \in G$

It remains to show that for all $p, p' \in \mathcal{Q}.D$, $p \preceq p'$ if and only if $p \parallel \mathit{mirror}(p') \in G$. Let $p = \langle p_1, p_2 \rangle$ and $p' = \langle p'_1, p'_2 \rangle$.

$p \preceq p'$

$\Leftrightarrow \quad \langle p_1, p_2 \rangle \preceq \langle p'_1, p'_2 \rangle$

by definition 2.13

$\Leftrightarrow \quad p_1 \preceq p'_1 \wedge p_2 \preceq p'_2$

by definition 3.61, since $\mathit{mirror}(p'_1)$ and $\mathit{mirror}(p'_2)$ are both defined

$\Leftrightarrow \quad p_1 \parallel \mathit{mirror}(p'_1) \in G_1 \wedge p_2 \parallel \mathit{mirror}(p'_2) \in G_2$

by definition 2.13

$\Leftrightarrow \quad \langle p_1, p_2 \rangle \parallel \langle \mathit{mirror}(p'_1), \mathit{mirror}(p'_2) \rangle \in G$

by hypothesis

$\Leftrightarrow \quad \langle p_1, p_2 \rangle \parallel \mathit{mirror}(\langle p'_1, p'_2 \rangle) \in G$

$\Leftrightarrow \quad p \parallel \mathit{mirror}(p') \in G$

□

In the rest of the section we consider subalgebras. We distinguish between two cases. For the first case, we consider subalgebras that preserve the agent ordering, as described in definition 2.41. We show that, in that case, the conformance order in the subalgebra becomes stronger, since fewer contexts may contribute to the notion of conformance. Consequently, the subalgebra may fail to have a conformance order. If however the superalgebra has a mirror function, and if the subalgebra is closed under that mirror function, then the subalgebra has the same mirror function and therefore a conformance order.

Theorem 3.112. Let $\mathcal{Q}'$ be an ordered agent algebra with a G'-conformance order and let $\mathcal{Q}$ be a subalgebra of $\mathcal{Q}'$. Let $G = G' \cap \mathcal{Q}.D$. Then for all agents p and p' in $\mathcal{Q}$, if $p \preceq_{\mathcal{Q}} p'$ then for all expression contexts E over $\mathcal{Q}$, if $E[p'] \in G$, then $E[p] \in G$.

Proof: Let $\mathcal{E}'$ be the set of expressions over $\mathcal{Q}'$, and let $\mathcal{E}$ be the set of expressions over $\mathcal{Q}$. Note that since $\mathcal{Q}$ is a subalgebra of $\mathcal{Q}'$, an expression over $\mathcal{Q}$ is also an expression over $\mathcal{Q}'$, and therefore $\mathcal{E} \subseteq \mathcal{E}'$.

Let now p and p' be elements of $\mathcal{Q}.D$. The proof consists of the following series of implications.

$p \preceq_{\mathcal{Q}} p'$

by definition 2.41

$\Leftrightarrow \quad p \preceq_{\mathcal{Q}'} p'$

since $\mathcal{Q}'$ has a G'-conformance order, by definition 3.26

$\Leftrightarrow \quad \forall E \in \mathcal{E}', E[p'] \in G' \Rightarrow E[p] \in G'$

since $\mathcal{Q}.D$ is closed in $\mathcal{Q}'.D$ under the operators, $G = G' \cap \mathcal{Q}.D$, $\mathcal{E} \subseteq \mathcal{E}'$,

and since for all $p \in \mathcal{Q}.D$, $E[p'] \in G' \Leftrightarrow E[p'] \in G$,

$\Rightarrow \quad \forall E \in \mathcal{E}, E[p'] \in G \Rightarrow E[p] \in G$

□

The reverse of the last implication does not hold. In fact, while it is true that if $E[p] \in G$, then $E[p] \in G'$, the subalgebra can only consider a subset of the contexts, and may therefore be unable to completely characterize the order.

Theorem 3.113. Let $\mathcal{Q}'$ be an ordered agent algebra with a mirror function *mirror* relative to G' and let $\mathcal{Q}$ be a subalgebra of $\mathcal{Q}'$ closed under *mirror*. Let $G = G' \cap \mathcal{Q}.D$. Then $\mathcal{Q}$ has a mirror function relative to G.

Proof: We show that that *mirror* is a mirror function for $\mathcal{Q}$ relative to G. Clearly *mirror* is a partial function.

Let now $p \in \mathcal{Q}.D$ be an agent. If $\mathit{mirror}(p)$ is defined, then, since $\mathcal{Q}$ is closed under mirror, $\mathit{mirror}(p) \in \mathcal{Q}.D$. Since *mirror* is a mirror function relative to G' for $\mathcal{Q}'$, by lemma 3.62, $p \parallel \mathit{mirror}(p) \in G'$. Since $p \in \mathcal{Q}.D$, $\mathit{mirror}(p) \in \mathcal{Q}.D$, and since $\mathcal{Q}$ is closed under parallel composition, $p \parallel \mathit{mirror}(p) \in \mathcal{Q}.D$. Therefore, since $G = G' \cap \mathcal{Q}.D$, $p \parallel \mathit{mirror}(p) \in G$. Hence, if $\mathit{mirror}(p)$ is defined, then there exists q (i.e., $\mathit{mirror}(p)$) such that $p \parallel q \in G$.

Conversely, if there exists $q \in \mathcal{Q}.D$ such that $p \parallel q \in G$, then, since $\mathcal{Q} \subseteq \mathcal{Q}'$, also $p \parallel q \in G'$. Therefore, by definition 3.61, $\mathit{mirror}(p)$ is defined.

Let now $p \in \mathcal{Q}.D$ and $p' \in \mathcal{Q}.D$. It remains to show that $p \preceq p'$ if and only if either $\mathit{mirror}(p')$ is undefined or $p \parallel \mathit{mirror}(p') \in G$.

$p \preceq_{\mathcal{Q}} p'$

since $\mathcal{Q} \subseteq \mathcal{Q}'$

$\Leftrightarrow \quad p \preceq_{\mathcal{Q}'} p'$

since $\mathcal{Q}'$ has a mirror function relative to G', by definition 3.61

$\Leftrightarrow \quad \mathit{mirror}(p')\!\uparrow \vee\, p \parallel \mathit{mirror}(p') \in G'$

since $\mathcal{Q}$ is closed under mirror and $G = G' \cap \mathcal{Q}.D$

$\Leftrightarrow \quad \mathit{mirror}(p')\!\uparrow \vee\, p \parallel \mathit{mirror}(p') \in G$

□

For the second case, we consider a subalgebra that does not preserve the order, but that is simply closed under the operators. This notion of subalgebra corresponds to the one described in definition 2.18. We assume that both the superalgebra and the subalgebra have a conformance order. We show that the order in the subalgebra is stronger than the order in the superalgebra, since in the subalgebra case there are fewer context to be satisfied in the definition of conformance.

Theorem 3.114. Let $\mathcal{Q}'$ be an ordered agent algebra with a G'-conformance order and let $\mathcal{Q}$ be a subalgebra of $\mathcal{Q}'$. Let $G = G' \cap \mathcal{Q}.D$ and assume $\mathcal{Q}$ has a G-conformance order. Then for all agents p and p' in $\mathcal{Q}$,

$$p \preceq_{\mathcal{Q}'} p' \Rightarrow p \preceq_{\mathcal{Q}} p'.$$

Proof: Let p and p' be elements of $\mathcal{Q}.D$. The proof consists of the following series of implications.

$p \preceq_{\mathcal{Q}'} p'$

since $\mathcal{Q}'$ has a G'-conformance order, by definition 3.26

$\Leftrightarrow \quad \forall E \in \mathcal{E}', E[p'] \in G' \Rightarrow E[p] \in G'$

since $\mathcal{Q}.D$ is closed in $\mathcal{Q}'.D$ under the operators, $G = G' \cap \mathcal{Q}.D$, $\mathcal{E} \subseteq \mathcal{E}'$,

and since for all $p \in \mathcal{Q}.D$, $E[p'] \in G' \Leftrightarrow E[p'] \in G$,

$\Rightarrow \quad \forall E \in \mathcal{E}, E[p'] \in G \Rightarrow E[p] \in G$

since $\mathcal{Q}$ has a G-conformance order, by definition 3.26

$\Leftrightarrow \quad p \preceq_{\mathcal{Q}} p'$

□

3.4 Local Specification Synthesis

With conformance we have addressed the problem of characterizing substitutability under all possible contexts. Relative conformance has been introduced to reduce (whenever possible) *the complexity* of the problem by considering only a limited set of contexts. Relative conformance, however, when applicable, does not change the notion of substitutability, since, in that case, relative and general conformance coincide (see theorem 3.42).

In this section we address the problem of deriving the local specification for an agent in a context, such that when an agent that satisfies the local specification is substituted in the context, the resulting system satisfies a global specification. Instances of this problem include supervisory-control synthesis [4], the rectification and optimization problem [13], and protocol conversion [73]. We will show that, under certain conditions, a mirror function provides us with a closed form solution.

Definition 3.115 (Local Specification). Let $\mathcal{Q}$ be an ordered agent algebra, E an expression context, and let p' be an agent. A *local specification for p' in E* is an agent q such that for all agents p,

$$p \preceq q \Leftrightarrow E[p] \preceq p'.$$

In the rest of this section we address the problem of deriving the local specification q, given the expression context E and the global specification p'. The solution involves the use of the mirror function. However, to solve the equation for the local specification, the conformance set must have some additional closure properties. We call a conformance set with these additional properties a *rectification set*.

Definition 3.116 (Rectification Set). Let $\mathcal{Q}$ be an agent algebra. A set $G \subset \mathcal{Q}.D$ is a *rectification set* if it satisfies the following requirements:

Downward closure If $p' \in G$ and $p \preceq p'$, then $p \in G$.

Closure under projection If $p \in G$, then for all alphabets B, *proj*$(B)(p)$ is defined and *proj*$(B)(p) \in G$.

Closure under inverse projection If $p \in G$, then for all alphabets B and all agents p', if *proj*$(B)(p') = p$ then $p' \in G$.

Closure under renaming If $p \in G$, then for all bijections r, if $\mathit{rename}(r)(p)$ is defined then $\mathit{rename}(r)(p) \in G$.

An agent algebra must be normalizable to synthesize a local specification. In fact, we need two additional properties to make sure that certain operations are well defined.

Definition 3.117 (Rectifiable Algebra). Let $\mathcal{Q}$ be a normalizable agent algebra. Then $\mathcal{Q}$ is rectifiable if it satisfies the following axioms, where p is an agent:

A26. $\mathit{rename}(r)(p)$ is defined if and only if $\alpha(p) \subseteq \mathit{dom}(r)$.

A27. For all alphabets A such that $\alpha(p) \subseteq A$, $\mathit{rename}(\mathit{id}_A)(p) = p$.

In order to find an algebraic solution to the problem of finding a local specification it is convenient to first transform the expression context into an equivalent expression in RCP normal form.

Lemma 3.118. Let $\mathcal{Q}$ be a normalizable agent algebra. Let $E[\beta]$ be an expression context that contains only one instance of the free variable. Then E is equivalent to an expression

$$E' = \mathit{proj}(B)(\mathit{rename}(\beta) \parallel q)$$

in RCP normal form.

Proof: By theorem 3.16, and since parallel composition is associative (definition 2.6, axiom A6), E is equivalent to an expression

$$E_1 = \mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel \mathit{rename}(r_1)(p_1) \parallel \cdots \parallel \mathit{rename}(r_n)(E_n)),$$

where β is the free variable and p_1 through p_n are the constant agents that appear in the expression E. Note that theorem 3.16 also ensures that β appears only once in E_1, since it appears only once in E.

Let now q be the agent such that

$$q = [\![\mathit{rename}(r_1)(p_1) \parallel \cdots \parallel \mathit{rename}(r_n)(E_n)]\!].$$

Then, by theorem 3.4,

$$E_1 \equiv \mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q).$$

□

We can now state and prove the main result of this section.

Theorem 3.119 (Local Specification Synthesis). Let $\mathcal{Q}$ be an ordered rectifiable algebra and let G be a rectification set, such that $\mathcal{Q}$ has a G-conformance order relative to composition. Assume $\mathcal{Q}$ has a mirror function relative to G.

Let $E[\beta]$ be an expression context that contains only one instance of the free variable, and let p be an agent such that $\mathit{mirror}(p)$ is defined. Let

$$\mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q)$$

be an expression in RCP normal form equivalent to E. The existence of this expression is guaranteed by lemma 3.118 above. Let $A_1 = \mathit{codom}(r) \cup \alpha(q) \cup B$ and let $\hat{r}^{-1}$ be an extension of r^{-1} to A_1 such that $\hat{r}^{-1}$ is a bijection. Then

$$E[\beta] \preceq p$$

if and only if

$$\beta \preceq \ \mathit{mirror}(\mathit{rename}(\hat{r}^{-1})(q \parallel \mathit{proj}(B)(\mathit{mirror}(p)))) \text{ and } \alpha(\beta) \subseteq \mathit{dom}(r).$$

Proof: The proof is composed of the following series of double implications.

$\mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q) \preceq p$

by the characterization of "$\preceq$" in terms of G, since $\mathit{mirror}(p)$ exists

$\Leftrightarrow \quad \mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q) \parallel \mathit{mirror}(p) \in G$

since G is closed under projection and inverse projection

$\Leftrightarrow \quad \mathit{proj}(B)(\mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q) \parallel \mathit{mirror}(p)) \in G$

since, by A1, $\alpha(\mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q)) \subseteq B$ and

since $\alpha(\mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q) \cap \alpha(\mathit{mirror}(p)) \subseteq B$,

therefore by A25

$\Leftrightarrow \quad \mathit{proj}(B)(\mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q)) \parallel \mathit{proj}(B)(\mathit{mirror}(p)) \in G$

by A20

$\Leftrightarrow \quad \mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q) \parallel \mathit{proj}(B)(\mathit{mirror}(p)) \in G$

by A20

$\Leftrightarrow \quad \mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q) \parallel \mathit{proj}(B)(\mathit{proj}(B)(\mathit{mirror}(p))) \in G$

since, by A1, $\alpha(\mathit{proj}(B)(\mathit{mirror}(p))) \subseteq B$ and

since $\alpha(\mathit{rename}(r)(\beta) \parallel q) \cap \alpha(\mathit{proj}(B)(\mathit{mirror}(p))) \subseteq B$,

therefore by A25

$\Leftrightarrow \quad \mathit{proj}(B)(\mathit{rename}(r)(\beta) \parallel q \parallel \mathit{proj}(B)(\mathit{mirror}(p))) \in G$

since G is closed under projection and inverse projection

$\Leftrightarrow \quad \mathit{rename}(r)(\beta) \parallel q \parallel \mathit{proj}(B)(\mathit{mirror}(p)) \in G$

by A27

$\Leftrightarrow \quad \mathit{rename}(\mathit{id}_{A_1})(\mathit{rename}(r)(\beta) \parallel q \parallel \mathit{proj}(B)(\mathit{mirror}(p))) \in G$

since $\hat{r}^{-1}$ is a bijection over A_1

$\Leftrightarrow \quad \mathit{rename}(\hat{r} \circ \hat{r}^{-1})(\mathit{rename}(r)(\beta) \parallel q \parallel \mathit{proj}(B)(\mathit{mirror}(p))) \in G$

by A21

$\Leftrightarrow \quad \mathit{rename}(\hat{r})(\mathit{rename}(\hat{r}^{-1})(\mathit{rename}(r)(\beta) \parallel q \parallel \mathit{proj}(B)(\mathit{mirror}(p)))) \in G$

since G is closed under rename (and consequently under inverse rename)

$\Leftrightarrow \quad \mathit{rename}(\hat{r}^{-1})(\mathit{rename}(r)(\beta) \parallel q \parallel \mathit{proj}(B)(\mathit{mirror}(p))) \in G$

by A24

$\Leftrightarrow \quad \mathit{rename}(\hat{r}^{-1})(\mathit{rename}(r)(\beta)) \parallel \mathit{rename}(\hat{r}^{-1})(q \parallel \mathit{proj}(B)(\mathit{mirror}(p))) \in G$

by A21

$\Leftrightarrow \quad \mathit{rename}(\hat{r}^{-1} \circ r)(\beta) \parallel \mathit{rename}(\hat{r}^{-1})(q \parallel \mathit{proj}(B)(\mathit{mirror}(p))) \in G$

since $\hat{r}^{-1}$ is an extension of r^{-1}

$\Leftrightarrow \quad \mathit{rename}(\mathit{id}_{\mathit{dom}(r)})(\beta) \parallel \mathit{rename}(\hat{r}^{-1})(q \parallel \mathit{proj}(B)(\mathit{mirror}(p))) \in G$

by A27

$\Leftrightarrow \quad \beta \parallel \mathit{rename}(\hat{r}^{-1})(q \parallel \mathit{proj}(B)(\mathit{mirror}(p))) \in G$ and $\alpha(\beta) \subseteq \mathit{dom}(r)$

by the characterization of "$\preceq$" in terms of G

$\Leftrightarrow \quad \beta \preceq \mathit{mirror}(\mathit{rename}(\hat{r}^{-1})(q \parallel \mathit{proj}(B)(\mathit{mirror}(p))))$ and $\alpha(\beta) \subseteq \mathit{dom}(r)$

$\square$

Theorem 3.119 applies in general to an agent algebra with the required properties. In subsection 3.3.2 we have explored a notion of mirror that applies to individual partitions of an agent algebra when the equivalence classes are closed under parallel composition and under the agent ordering. There we have specifically considered the case where $p \preceq p'$ implies that p and p' have the same alphabet. In that case, a simplified version of the theorem can be derived. Specifically, we assume we are looking for the local specification of an agent p_1 with a specific alphabet A_1, under a global specification p with alphabet A. The context is represented by another agent p_2. Note that the projection in the normal form expression is required to retain alphabet A, since it must match the alphabet of the specification. Also, we know that the mirror function preserves the alphabet. Under this assumption, we may therefore prove the following result.

Theorem 3.120. Let $\mathcal{Q}$ be an ordered agent algebra with a same alphabet mirror function relative to G (def. 3.107). Let p_1, p_2 and p be agents with alphabet A_1, A_2 and A respectively. Assume that $\mathit{mirror}(p)$ is defined, that $A \subseteq A_1 \cup A_2$, $A_1 \subseteq A_2 \cup A$. Assume further that $\mathit{mirror}(\mathit{proj}(A_1)(p_2 \parallel \mathit{mirror}(p)))$ exists. Then

$$\mathit{proj}(A)(p_1 \parallel p_2) \preceq p$$

if and only if

$$p_1 \preceq \mathit{mirror}(\mathit{proj}(A_1)(p_2 \parallel \mathit{mirror}(p))).$$

Proof: Note that since $A \subseteq A_1 \cup A_2$, then

$$\alpha(\mathit{proj}(A)(p_1 \parallel p_2)) = A \cap (A_1 \cup A_2) = A.$$

The proof is given by the following series of double implications.

$\mathit{proj}(A)(p_1 \parallel p_2) \preceq p$

by the characterization of "$\preceq$" in terms of G, since $\mathit{mirror}(p)$ exists and since $\alpha(\mathit{proj}(A)(p_1 \parallel p_2)) = \alpha(p)$

$\Leftrightarrow \quad \mathit{proj}(A)(p_1 \parallel p_2) \parallel \mathit{mirror}(p) \in G$

By A2 since $\alpha(\mathit{mirror}(p)) = A$

$\Leftrightarrow \quad \mathit{proj}(A)(p_1 \parallel p_2) \parallel \mathit{proj}(A)(\mathit{mirror}(p)) \in G$

By A5

$\Leftrightarrow \quad \mathit{proj}(A \cap (A_1 \cup A_2))(p_1 \parallel p_2) \parallel \mathit{proj}(A \cap A)(\mathit{mirror}(p)) \in G$

by A15 since $(A_1 \cup A_2) \cap A \subseteq A$

$\Leftrightarrow \quad \mathit{proj}(A)(p_1 \parallel p_2 \parallel \mathit{mirror}(p)) \in G$

Since G is closed under projection and inverse projection

$\Leftrightarrow \quad p_1 \parallel p_2 \parallel \mathit{mirror}(p) \in G$

Since G is closed under projection and inverse projection

$\Leftrightarrow \quad \mathit{proj}(A_1)(p_1 \parallel p_2 \parallel \mathit{mirror}(p)) \in G$

by A15 since $A_1 \cap (A_2 \cup A) \subseteq A_1$

$\Leftrightarrow \quad \mathit{proj}(A_1)(p_1) \parallel \mathit{proj}(A_1 \cap (A_2 \cup A))(p_2 \parallel \mathit{mirror}(p)) \in G$

By theorem 3.66 and theorem 3.63 since

$\mathit{mirror}(\mathit{proj}(A_1)(p_2 \parallel \mathit{mirror}(p)))$ exists

$\Leftrightarrow \quad \mathit{proj}(A_1)(p_1) \parallel \mathit{mirror}^2(\mathit{proj}(A_1 \cap (A_2 \cup A))(p_2 \parallel \mathit{mirror}(p))) \in G$

by the characterization of "$\preceq$" in terms of G, since

$A_1 \subseteq A_2 \cup A \Rightarrow A_1 \cap (A_2 \cup A) = A_1$ and

$\mathit{mirror}(\mathit{proj}(A_1)(p_2 \parallel \mathit{mirror}(p)))$ exists

and has alphabet A_1

$\Leftrightarrow \quad p_1 \preceq \mathit{mirror}(\mathit{proj}(A_1)(p_2 \parallel \mathit{mirror}(p)))$

□

Note that the requirements on G are unchanged, and it still must be closed under projection and inverse projection. This essentially allows us to switch from one equivalence class to another during the proof of the theorem (see the steps that require closure of the conformance set under projection and inverse projection in the proof above). Note also that certain of the assumptions of the theorem are not really restrictions on its applicability. For example, since we are considering the problem $\mathit{proj}(A)(p_1 \| p_2) \preceq p$, it follows that $A \cap (A_1 \cup A_2) = A$ (since $p \preceq p' \Rightarrow \alpha(p) = \alpha(p')$) which implies $A \subseteq A_1 \cup A_2$. Likewise, we can derive the following series of equalities from the form of the result:

$$\begin{aligned}
A_1 &= \alpha(\mathit{mirror}(\mathit{proj}(A_1 \cap (A_2 \cup A))(p_2 \parallel \mathit{mirror}(p)))) \\
&= \alpha(\mathit{proj}(A_1 \cap (A_2 \cup A))(p_2 \parallel \mathit{mirror}(p))) \\
&= A_1 \cap (A_2 \cup A) \cap (A_2 \cup A) \\
&= A_1 \cap (A_2 \cup A),
\end{aligned}$$

which implies that $A_1 \subseteq A_2 \cup A$.

In the following example we show how to use the local specification synthesis technique in the IO Agent Algebra.

Example 3.121 ((Locked) IO Agent Algebra). Consider the IO agent algebra described in example 3.87. Figure 3.3 shows an intuitive graphical representation of the system

$$\mathit{proj}(\{\, a, b, c, d, e, f\,\})(\beta \parallel p_1 \parallel p_2),$$

where

$$\begin{aligned} p_1 &= (\{\, a, g, h\}, \{\, d\}), \\ p_2 &= (\{\, d\}, \{\, g, j\,\}) \end{aligned}$$

and β is an agent variable. Suppose we would like to solve the system for β so that it satisfies

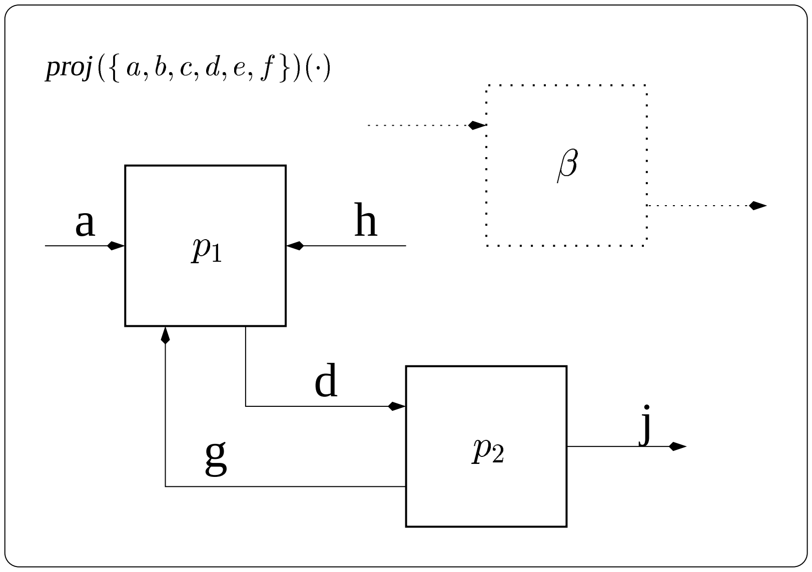

Figure 3.3: IO agent system

the specification

$$p' = (\{\, a, b\}, \{\, c, d\}).$$

As discussed in example 3.87, this algebra does not have mirrors, and therefore we are unable to apply our solution to the local specification synthesis. However we can embed the model in the Locked IO agent algebra described in example 3.88 as follows:

$$\begin{aligned} p_1 &\to (\{a,g,h\},\{d\},\emptyset), \\ p_2 &\to (\{d\},\{g,j\},\emptyset) \\ p' &\to (\{a,b\},\{c,d\},\emptyset). \end{aligned}$$

Because of the embedding, the system expression is unchanged. Thus, by applying theorem 3.119 we obtain

$$\mathit{proj}(\{a,b,c,d,e,f\})(\beta \parallel p_1 \parallel p_2) \preceq p'$$

if and only if

$$\beta \preceq \mathit{mirror}(p_1 \parallel p_2 \parallel \mathit{proj}(\{a,b,c,d,e,f\})(\mathit{mirror}(p')))$$

Substituting the real quantities for the symbols:

$$\begin{aligned} &\mathit{mirror}(p_1 \parallel p_2 \parallel \mathit{proj}(\{a,b,c,d,e,f\})(p')) = \\ &\quad = \mathit{mirror}((\{a,g,h\},\{d\},\emptyset) \parallel (\{d\},\{g,j\},\emptyset) \parallel \\ &\qquad \parallel \mathit{proj}(\{a,b,c,d,e,f\})(\mathit{mirror}((\{a,b\},\{c,d\},\emptyset)))) \\ &\quad = \mathit{mirror}((\{a,h\},\{d,g,j\},\emptyset) \parallel \\ &\qquad \parallel \mathit{proj}(\{a,b,c,d,e,f\})(\{c,d\},\{a,b\},\mathcal{A}-\{a,b,c,d\})) \\ &\quad = \mathit{mirror}((\{a,h\},\{d,g,j\},\emptyset) \parallel \mathit{proj}(\{a,b,c,d,e,f\})(\{c,d\},\{a,b\},\{e,f\})) \\ &\quad = \mathit{mirror}((\{h,c\},\{a,b,d,g,j\},\{e,f\})) \\ &\quad = (\{a,b,d,g,j\},\{h,c\},\mathcal{A}-\{a,b,c,d,e,f,g,h,j\}) \end{aligned}$$

Recall that $p \preceq p'$ if and only if $I \subseteq I'$, $O' \subseteq O \subseteq O' \cup L'$ and $L \subseteq L'$. If we only consider agents that have $L = \emptyset$, the agents $p = (I, O, L)$ that can be assigned to β must be such that

$$\begin{aligned} \emptyset &\subseteq I \subseteq \{a,b,d,g,j\} \\ \{c,h\} &\subseteq O \subseteq \mathcal{A}-\{a,b,d,e,f,g,j\} \end{aligned}$$

We interpret this result as follows. Agent p can have as input any of the inputs allowed by the specification (i.e., a and b) and any of the outputs that are already present in the system (d, g and j), whether they are retained (d) or not (g and j). It cannot have any additional input, since they would be left "unconnected" and hidden, a situation that is not allowed by the definition of the algebra. Note that p is not *required* to have any input, even though b (which is in the specification) is not already present. That is because the order only requires that the set of inputs of the implementation be *contained* in the set of inputs of the specification.

On the other hand, p *must* have outputs c and h in order for the system to satisfy the specification. In fact, c is required by the specification and is not already present in the rest of the system, while h is an input to p_1, and it must be converted to an output in order to project it away. Agent p can also have additional outputs, but not a and b which are inputs to the system (having them as outputs would make them outputs, contrary to the specification), d, g and j, which are already outputs in the system (and thus would collide and make the parallel composition undefined), and e and f, which are retained in the projection but are not allowed by the specification.

3.5 Conservative Approximations and Mirrors

In section 4.4 we show for trace-based agent algebras how a relation between models of individual behaviors of different concurrent systems can be used to induce a conservative approximation between the corresponding agent models. The relation is used to derive a Galois connection between the powersets of the behaviors. A second Galois connection, in the opposite direction, is obtained by computing the complement of sets of behaviors relative to the universe of behaviors. This technique is interesting because it simplifies the construction of a conservative approximation by considering functions or relations on the simpler models of individual behaviors.

It is impossible to apply the same result in the framework of agent algebra, where agents are not necessarily described as sets of behaviors. However, a similar technique of complementing the components of a conservative approximation can be used when a mirror function exists. In this case, the mirror takes the place of set complementation.

In the following, we start by defining the dual of a function relative to a mirror. We then show that the duals of a Galois connection between two partially ordered domains of agents is again a Galois connection, in the reverse direction. Finally we derive the necessary and sufficient conditions for the pair of Galois connection to form a conservative approximation.

Definition 3.122 (Dual). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be ordered agent algebras with a mirror function relative to G and G', respectively. Let $f : \mathcal{Q}.D \mapsto \mathcal{Q}'.D$ be a function from agents in $\mathcal{Q}$ to agents in $\mathcal{Q}'$.

The *dual of f*, written $\widetilde{f}$, is a function from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$ such that for all agents $p \in \mathcal{Q}.D$,

$$\widetilde{f}(p) = \mathit{mirror}(f(\mathit{mirror}(p)).$$

Theorem 3.123 (Dual Connection). Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras with a mirror function relative to G and G', respectively. Let $\langle \alpha, \gamma \rangle$ be a Galois connection from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$. Then $\langle \widetilde{\gamma}, \widetilde{\alpha} \rangle$ is a Galois connection from $\mathcal{Q}'.D$ to $\mathcal{Q}.D$.

Proof: Let $p \in \mathcal{Q}.D$ and $p \in \mathcal{Q}'.D$ be agents. We prove that $\widetilde{\gamma}(p') \preceq p$ if and only if $p' \preceq \widetilde{\alpha}(p)$. The result can be derived as follows.

$\widetilde{\gamma}(p') \preceq p$

by definition 3.122

$\Leftrightarrow \quad \mathit{mirror}(\gamma(\mathit{mirror}(p'))) \preceq p$

by lemma 3.65

$\Leftrightarrow \quad \mathit{mirror}(p) \preceq \gamma(\mathit{mirror}(p'))$

since $\langle \alpha, \gamma \rangle$ is a Galois connection, by definition 2.74

$\Leftrightarrow \quad \alpha(\mathit{mirror}(p)) \preceq \mathit{mirror}(p')$

by lemma 3.65

$\Leftrightarrow \quad \mathit{mirror}^2(p') \preceq \mathit{mirror}(\alpha(\mathit{mirror}(p)))$

by theorem 3.66, since $\mathcal{Q}'$ is partially ordered

$\Leftrightarrow \quad p' \preceq \mathit{mirror}(\alpha(\mathit{mirror}(p)))$

by definition 3.122

$\Leftrightarrow \quad p' \preceq \widetilde{\alpha}(p).$

□

Theorem 3.124. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras with a mirror function relative to G and G', respectively. Let $\langle \alpha, \gamma \rangle$ be a Galois connection from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$, and let $\langle \widetilde{\gamma}, \widetilde{\alpha} \rangle$ be the dual Galois connection from $\mathcal{Q}'$ to $\mathcal{Q}$ (by thm. 3.123). Then the following two statements are equivalent.

1. For all agents $p' \in \mathcal{Q}'.D$, $\gamma(p') \preceq \widetilde{\gamma}(p')$.
2. For all agents $p' \in \mathcal{Q}'.D$, $\gamma(\mathit{mirror}(p')) \preceq \mathit{mirror}(\gamma(p'))$.

Proof: Let $p' \in \mathcal{Q}'.D$ be an agent. Then

$$\gamma(p') \preceq \widetilde{\gamma}(p')$$

by definition 3.122

$$\Leftrightarrow \quad \gamma(p') \preceq \mathit{mirror}(\gamma(\mathit{mirror}(p')))$$

by lemma 3.65

$$\Leftrightarrow \quad \mathit{mirror}^2(\gamma(\mathit{mirror}(p'))) \preceq \mathit{mirror}(\gamma(p'))$$

by theorem 3.66, since $\mathcal{Q}$ is partially ordered

$$\Leftrightarrow \quad \gamma(\mathit{mirror}(p')) \preceq \mathit{mirror}(\gamma(p')).$$

□

Corollary 3.125. Let $\mathcal{Q}$ and $\mathcal{Q}'$ be partially ordered agent algebras with a mirror function relative to G and G', respectively. Let $\langle \alpha, \gamma \rangle$ be a Galois connection from $\mathcal{Q}.D$ to $\mathcal{Q}'.D$, and let $\langle \widetilde{\gamma}, \widetilde{\alpha} \rangle$ be the dual Galois connection from $\mathcal{Q}'$ to $\mathcal{Q}$ (by thm. 3.123). Then $(\widetilde{\alpha}, \alpha)$ is a conservative approximation if and only if for all agents $p' \in \mathcal{Q}'.D$, $\gamma(\mathit{mirror}(p')) \preceq \mathit{mirror}(\gamma(p'))$.

Proof: The result follows directly from theorem 3.124 and corollary 2.101. □

Chapter 4

Trace-Based Agent Algebras

A trace-based agent algebra is a particular kind of agent algebra whose agents are composed of sets of elementary elements that we call traces. Trace-based agent algebras are based on trace algebras and trace structure algebras [12] and can be used to construct different models of concurrent systems. In this chapter we introduce the concept of a trace-based agent algebra, show its construction, and we then present several examples of trace-based algebras that span different levels of abstraction and application areas of interest in the design of embedded systems.

4.1 Introduction

The models of computation in use for embedded concurrent systems represent a design by a collection of agents (processes, actors, modules) that interact to perform a function. For any particular input to the system, the agents react with some particular execution, or behavior. In the trace-based agent algebras framework we maintain a clear distinction between models of agents and models of individual executions. In different models of computation, individual executions can be modeled by very different kinds of mathematical objects. We always call these objects *traces*. A model of an agent, which we call a *trace structure*, consists primarily of a set of traces. This is analogous to verification methods based on language containment, where individual executions are modeled by strings and agents are modeled by sets of strings. However, our notion of trace is more general and so is not limited to strings.

Traces often refer to the externally visible features of agents: their actions, signals, state variables, etc. We do not distinguish among the different types, and we refer to them collectively using the set of signals in the master alphabet $\mathcal{A}$. Each trace and each trace structure is then associated with an alphabet $A \subseteq \mathcal{A}$ of the signals it uses.

The first step in defining a model of computation as a trace-based agent algebra is to construct a *trace algebra*. The carrier of a trace algebra contains the universe of traces for the model of computation. The algebra also includes operations on traces such as *projection* and *renaming*. Intuitively, these operations correspond to encapsulation and instantiation, respectively. Any mathematical object that satisfies certain minimum requirements can be used as a trace. These requirements are formalized as the axioms of *trace algebra*.

The second step is to construct a *trace structure algebra*. Here each element of the algebra is a trace structure, which consists primarily of a set of traces from the trace algebra constructed in the first step. Given a trace algebra, and the set of trace structures to be used as the universe of agent models, a trace structure algebra is constructed in a fixed way. The construction ensures that the trace structure algebra is also an agent algebra. Thus, constructing a trace algebra is the creative part of defining a model of computation. Constructing the corresponding trace structure algebra is much easier. A trace structure algebra includes the operations of *projection*, *renaming* and *parallel composition* on agents.

Each trace structure algebra has a refinement order that is based on trace containment. We say that an agent p_1 refines an agent p_2, written $p_1 \preceq p_2$, if the set of traces of p_1 is a subset of the set of traces of p_2. Intuitively, this means that the implementation p_1 can be substituted for the specification p_2. It is easy to show that the refinement relationships constitutes a preorder on the set of trace structures. The definitions and the construction given in this section make sure that the operators of a trace structure algebra are $\top$-monotonic relative to the refinement order, and that therefore the trace structure algebra is an ordered agent algebra.

Conservative approximations can also be defined between trace structure algebras. Defining a conservative approximations and proving that it satisfies the definition can sometimes be difficult. However, a conservative approximation between trace structure algebras can be derived from a homomorphism between the underlying trace algebras, which is often easier to define.

The relationships between trace algebras and trace structure algebras is depicted in figure 4.1. This figure also shows the relationships between different algebras in terms of conservative approximations.

It is often convenient to make a distinction between two different kinds of behaviors: *complete* behaviors and *partial* behaviors. A complete behavior has no endpoint. Since a complete behavior goes on forever, it does not make sense to talk about something happening “after” a complete behavior. A partial behavior has an endpoint; it can be a prefix of a complete behavior or of

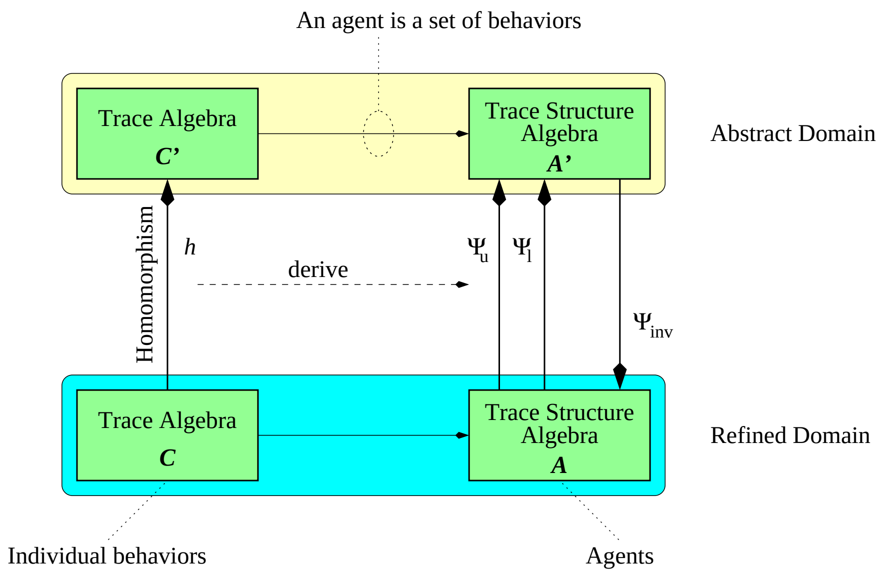

Figure 4.1: Algebras and their relationships

another partial behavior. Every complete behavior has partial behaviors that are prefixes of it; every partial behavior is a prefix of some complete behavior. The distinction between a complete behavior and a partial behavior has only to do with the length of the behavior (that is, whether or not it has an endpoint), not with what is happening during the behavior; whether an agent does anything, or what it does, is irrelevant. *Complete traces* and *partial traces* are used to model complete and partial behaviors, respectively.

Trace algebras that include complete and partial traces can be enriched with the additional operation of *concatenation*, which intuitively corresponds to sequential composition. Concatenation can be used to define the notion of a prefix of a trace. We say that a trace x is a prefix of a trace z if there exists a trace y such that z is equal to x concatenated with y. Likewise, the corresponding trace structure algebra includes an operation of *sequential composition*, which complements that of parallel composition, and is particularly useful for modeling programmable embedded systems.

In summary, a *trace algebra* has a set of traces as its domain, and each trace is interpreted as an abstraction of a physical behavior. A sequence of actions is a standard example of a trace, but in trace algebra any mathematical object can used as a trace as long as certain axioms are satisfied. An agent is modeled by a *trace structure*, which contains a set of traces from some trace algebra, representing the set of possible behaviors of the agent.

The operations of parallel composition, projection and renaming are defined over a domain of trace structures, forming a *trace structure algebra*. These operations satisfy the axioms of agent algebra, so a trace structure algebra is a special case of an agent algebra.

4.2 Trace Algebras and Trace Structure Algebras

We begin with the definition of trace algebra and of trace structure. We then construct trace structure algebra and show that trace structure algebras are agent algebras. In the following we assume that $\mathcal{A}$ is the master alphabet.

Definition 4.1 (Trace Algebra). Let $\mathcal{A}$ be a master alphabet. A *trace algebra* $\mathcal{C}$ over $\mathcal{A}$ is a triple $(\mathcal{B}, \mathit{proj}, \mathit{rename})$ such that

- For every alphabet A over $\mathcal{A}$, $\mathcal{B}(A)$ is a non-empty set, called the set of traces over A. Slightly abusing notation, we also write $\mathcal{B}$ as an abbreviation for

$$\bigcup \{\mathcal{B}(A) : A \text{ is an alphabet over } \mathcal{A}\}.$$

  Note that for two alphabets A_1 and A_2, $\mathcal{B}(A_1)$ and $\mathcal{B}(A_2)$ need not be disjoint.

- For every alphabet B over $\mathcal{A}$, $\mathit{proj}(B)$ is a partial function from $\mathcal{B}$ to $\mathcal{B}$.

- For every renaming function r over $\mathcal{A}$, $\mathit{rename}(r)$ is a partial functions from $\mathcal{B}$ to $\mathcal{B}$.

The following axioms must also be satisfied. For each axiom, we assume that A and B (and their decorated versions) are alphabets over $\mathcal{A}$, that r (and its decorated versions) is a renaming function over $\mathcal{A}$, and that $x \in \mathcal{B}(A)$.

T1. $\mathit{proj}(B)(x)$ is always defined and $\mathit{proj}(B)(x) \in \mathcal{B}(A \cap B)$.

T2. $\mathit{proj}(A)(x) = x$.

T3. $\mathit{proj}(B)(x) = \mathit{proj}(B \cap A)(x)$.

T4. $\mathit{proj}(B)(\mathit{proj}(B')(x)) = \mathit{proj}(B \cap B')(x)$.

T5. $\mathit{rename}(r)(x)$ is defined whenever $A \subseteq \mathit{dom}(r)$ and in that case $\mathit{rename}(r)(x) \in \mathcal{B}(r(A))$.

T6. $\mathit{rename}(\mathit{id}_A)(x) = x$.

T7. If $\mathit{rename}(r)(x)$ is defined, then $\mathit{rename}(r)(x) = \mathit{rename}(r\,|_{A \to r(A)})(x)$.

T8. Let $x_1 \in \mathcal{B}(A_1)$ and $x_2 \in \mathcal{B}(A_2)$ be such that $\mathit{proj}(A_1 \cap A_2)(x_1) = \mathit{proj}(A_1 \cap A_2)(x_2)$. For all alphabets A such that $A_1 \cup A_2 \subseteq A$, there exists $x \in \mathcal{B}(A)$ such that $x_1 = \mathit{proj}(A_1)(x)$ and $x_2 = \mathit{proj}(A_2)(x)$.

T9. If $A' \cap A = \emptyset$, then $\mathit{proj}(B \cup A')(x) = \mathit{proj}(B)(x)$.

T10. $\mathit{rename}(r)(x_1) = \mathit{rename}(r)(x_2) \Rightarrow x_1 = x_2$.

T11. $\mathit{rename}(r_1)(\mathit{rename}(r_2)(x)) = \mathit{rename}(r_1 \circ r_2)(x)$.

T12. Assume $r = \hat{r}\,|_{\mathit{dom}(r)}$, $r' = \hat{r}\,|_{\mathit{dom}(r')}$, $r'' = \hat{r}\,|_{\mathit{dom}(r'')}$, and that $\mathit{dom}(r) \subseteq \mathit{dom}(r')$ and $\mathit{dom}(r) \subseteq \mathit{dom}(r'')$. Then

$$\mathit{rename}(r)(\mathit{proj}(B)(x)) = \mathit{proj}(r'(B))(\mathit{rename}(r'')(x))$$

if both sides of the equation are defined.

T13. Assume $r\,|_B = \mathit{id}_{\mathit{dom}(r) \cap B}$. Then

$$\mathit{proj}(B)(x) = \mathit{proj}(B)(\mathit{rename}(r)(x))$$

if both sides of the equation are defined.

Definition 4.2 (Trace Structure). Let $\mathcal{C} = (\mathcal{B}, \mathit{proj}, \mathit{rename})$ be a trace algebra over $\mathcal{A}$. The set of *trace structures* over $\mathcal{C}$ is the set of ordered pairs (A, P), where

- A is an alphabet over $\mathcal{A}$, and
- P is a subset of $\mathcal{B}(A)$.

We call A the alphabet and P the set of possible traces of a trace structure $p = (A, P)$.

Note that it is necessary to make the alphabet explicit in the definition of a trace structure, since the sets $\mathcal{B}(A_1)$ and $\mathcal{B}(A_2)$ are not necessarily disjoint for distinct alphabets A_1 and A_2. Consequently, if $P \subseteq \mathcal{B}(A_1)$ and $P \subseteq \mathcal{B}(A_2)$, it would be impossible to associate a unique alphabet to the trace structure that has P as its set of traces, unless the alphabet is itself part of the structure.

Definition 4.3 (Trace Structure Algebra). Let $\mathcal{C} = (\mathcal{B}, \mathit{proj}, \mathit{rename})$ be a trace algebra over $\mathcal{A}$ and let $\mathcal{T}$ be a subset of the trace structures over $\mathcal{C}$. Then $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ is a *trace structure algebra* over $\mathcal{C}$ if and only if the domain $\mathcal{T}$ is closed under the following operations on trace structures: parallel composition (definition 4.4), projection (definition 4.5) and renaming (definition 4.6).

Definition 4.4 (Parallel Composition). $p = p_1 \parallel p_2$ is always defined and

$$\begin{aligned} A &= A_1 \cup A_2 \\ P &= \{\, x \in \mathcal{B}(A) : \mathit{proj}(A_1)(x) \in P_1 \wedge \mathit{proj}(A_2)(x) \in P_2 \}. \end{aligned}$$

Definition 4.5 (Projection). $p = \mathit{proj}(B)(p')$ is always defined and

$$\begin{aligned} A &= B \cap A' \\ P &= \mathit{proj}(B)(P'), \end{aligned}$$

where *proj* is naturally extended to sets.

Definition 4.6 (Renaming). $p = \mathit{rename}(r)(p')$ is defined whenever $A' \subseteq \mathit{dom}(r)$. In that case

$$\begin{aligned} A &= r(A') \\ P &= \mathit{rename}(r)(P'), \end{aligned}$$

where *rename* is naturally extended to sets.

Note that in definition 4.5 and definition 4.6, the operations effectively yield a trace structure, since by T1, $\mathit{proj}(B)(P') \subseteq \mathcal{B}(A)$, and by T5, $\mathit{rename}(r)(P') \subseteq \mathcal{B}(A)$.

Theorem 4.7. Trace structure algebras are agent algebras.

Proof: We need to show that A1 to A7 are satisfied. A1 follows from T1 and A2 follows from T2. Also, A3 follows from T5. A5, A6 and A7 all follow easily from definition 4.4. □

Theorem 4.8. Trace structure algebras are normalizable agent algebras.

Proof: It is easy to show, by simply extending to sets the corresponding axiom, that A18 follows from T6 and T7, that A19 follows from T9, A20 follows from T4 and that A21 follows from T11. The following lemmas prove the validity of the remaining axioms.

Lemma 4.9. Trace structure algebras satisfy A22.

Proof: Let r be a renaming function and let B be an alphabet over $\mathcal{A}$. We wish to show that there exist renaming functions r' and r'' such that for all trace structures p,

$$\mathit{rename}(r)(\mathit{proj}(B)(p)) = \mathit{proj}(r'(B))(\mathit{rename}(r'')(p)).$$

Let $\hat{r}$ be an extension of r to $\mathcal{A}$. We construct r' and r'' as follows:

$$\begin{aligned} r' &= \hat{r}, \\ r'' &= \hat{r}\,|_{\mathcal{A}-(B-\mathit{dom}(r))}. \end{aligned}$$

Let p be an agent with alphabet A. The following series of double implications shows that $\mathit{rename}(r)(\mathit{proj}(B)(p))$ is defined if and only if $\mathit{proj}(r'(B))(\mathit{rename}(r'')(p))$ is defined.

$$\begin{aligned}
&\mathit{rename}(r)(\mathit{proj}(B)(p))\downarrow \\
&\quad \text{by T1 and T5} \\
&\quad \Leftrightarrow \quad A \cap B \subseteq \mathit{dom}(r) \\
&\quad \Leftrightarrow \quad A \cap B \cap \overline{\mathit{dom}(r)} = \emptyset \\
&\quad \Leftrightarrow \quad A \cap (B - \mathit{dom}(r)) = \emptyset \\
&\quad \Leftrightarrow \quad A \cap \overline{(\mathcal{A} - (B - \mathit{dom}(r)))} = \emptyset \\
&\quad \Leftrightarrow \quad A \subseteq \mathcal{A} - (B - \mathit{dom}(r)) \\
&\quad \Leftrightarrow \quad A \subseteq \mathit{dom}(r'') \\
&\quad \text{by T1 and T5} \\
&\quad \Leftrightarrow \quad \mathit{proj}(r'(B))(\mathit{rename}(r'')(p))\downarrow.
\end{aligned}$$

The desired result then follows from T12, since $r = \hat{r}\,|_{\mathit{dom}(r)}$, $r' = \hat{r}\,|_{\mathit{dom}(r')}$, $r'' = \hat{r}\,|_{\mathit{dom}(r'')}$, and $\mathit{dom}(r) \subseteq \mathit{dom}(r')$ and $\mathit{dom}(r) \subseteq \mathit{dom}(r'')$. □

Lemma 4.10. Trace structure algebras satisfy A23.

Proof: Let B, A and A' be alphabets such that $|(\mathcal{A} - A') - B| \geq |A - B|$. We need to show that there exists a renaming function r such that $r(A) \cap A' \subseteq B$ and for all trace structures p such that $\alpha(p) \subseteq A$,

$$\mathit{proj}(B)(p) = \mathit{proj}(B)(\mathit{rename}(r)(p)).$$

Let r be a renaming function such that $\mathit{dom}(r) = A$ and such that for all $a \in A$,

$$r(a) = \begin{cases} a & \text{if } a \in B \\ c & \text{where } c \notin A' \cup B\text{, otherwise} \end{cases}$$

The renaming function r, which must be a bijection, exists since the size of $A - B$ is smaller than the size of the available signals $\mathcal{A} - (A' \cup B)$. In addition, since the range of r does not include any element of A' unless it is also in B, $r(A) \cap A' \subseteq B$. Let now p be a trace structure such that $\alpha(p) \subseteq A$. Then, since $\mathit{dom}(r) = A$, $\alpha(p) \subseteq \mathit{dom}(r)$. Therefore, $\mathit{proj}(B)(\mathit{rename}(r)(p))$ is always defined, as is $\mathit{proj}(B)(p)$. The equality then follows from T13, since $r\,|_B = \mathit{id}_{\mathit{dom}(r) \cap B}$. □

Lemma 4.11. Trace structure algebras satisfy A24.

Proof: Let r be a renaming function. We wish to show that for all trace structures $p_1 = (A_1, P_1)$ and $p_2 = (A_2, P_2)$,

$$\mathit{rename}(r)(p_1 \parallel p_2) = \mathit{rename}(r)(p_1) \parallel \mathit{rename}(r)(p_2).$$

The following series of double implications shows that $\mathit{rename}(r)(p_1 \parallel p_2)$ is defined if and only if $\mathit{rename}(r)(p_1) \parallel \mathit{rename}(r)(p_2)$ is defined.

$$\begin{aligned}
& \mathit{rename}(r)(p_1 \parallel p_2){\downarrow} \\
& \quad \text{by definition 4.4 and T5} \\
& \Leftrightarrow \quad A_1 \cup A_2 \subseteq \mathit{dom}(r) \\
& \Leftrightarrow \quad A_1 \subseteq \mathit{dom}(r) \land A_2 \subseteq \mathit{dom}(r) \\
& \quad \text{by definition 4.4 and T5} \\
& \Leftrightarrow \quad \mathit{rename}(r)(p_1) \parallel \mathit{rename}(r)(p_2){\downarrow}.
\end{aligned}$$

We now prove that the two sides of the equation are equal. Clearly, since r is a bijection, $r(A_1 \cup A_2) = r(A_1) \cup r(A_2)$, so that

$$\begin{aligned}
p &= (r(A_1 \cup A_2), P) = \mathit{rename}(r)(p_1 \parallel p_2) \\
p' &= (r(A_1) \cup r(A_2), P') = \mathit{rename}(r)(p_1) \parallel \mathit{rename}(r)(p_2).
\end{aligned}$$

have the same alphabet. In addition,

$$\begin{aligned}
P &= \{y \in \mathcal{B}(r(A_1 \cup A_2)) : \exists x \in \mathcal{B}(A_1 \cup A_2)\ [y = \mathit{rename}(r)(x) \\
&\qquad \land \mathit{proj}(A_1)(x) \in P_1 \land \mathit{proj}(A_2)(x) \in P_2]\} \\
P' &= \{y \in \mathcal{B}(r(A_1) \cup r(A_2)) : \mathit{proj}(A_1)(y) \in \mathit{rename}(r)(P_1) \\
&\qquad \land \mathit{proj}(A_2)(y) \in \mathit{rename}(r)(P_2)\}
\end{aligned}$$

We wish to show that $y \in P$ if and only if $y \in P'$. To do so, we will show that for every $y \in \mathcal{B}(r(A_1 \cup A_2))$ there is $x \in \mathcal{B}(A_1 \cup A_2)$ such that $y = \mathit{rename}(r)(x)$ and $\mathit{proj}(A_i)(x) \in P_i$ if and only if $\mathit{proj}(r(A_i))(y) \in \mathit{rename}(r)(P_i)$, for $i = 1, 2$.

Let $y \in \mathcal{B}(r(A_1 \cup A_2))$. By T5, $x = \mathit{rename}(r^{-1})(y)$ is defined, and $x \in \mathcal{B}(A_1 \cup A_2)$. In addition,

$$\begin{aligned}
\mathit{rename}(r)(x) &= \mathit{rename}(r)(\mathit{rename}(r^{-1})(y)) \\
&\quad \text{by T11} \\
&= \mathit{rename}(r \circ r^{-1})(y) \\
&\quad \text{since } r \text{ is a bijection} \\
&= \mathit{rename}(\mathit{id}_{\mathit{codom}(r)})(y) \\
&\quad \text{by T6} \\
&= y.
\end{aligned}$$

Then,

$$\begin{aligned}
&\mathit{proj}(A_1)(x) \in P_1 \\
&\Leftrightarrow \quad \exists z_1 \in P_1\ [\mathit{proj}(A_1)(x) = z_1] \\
&\quad \text{by T10} \\
&\Leftrightarrow \quad \exists z_1 \in P_1\ [\mathit{rename}(r)(\mathit{proj}(A_1)(x)) = \mathit{rename}(r)(z_1)] \\
&\Leftrightarrow \quad \mathit{rename}(r)(\mathit{proj}(A_1)(x)) \in \mathit{rename}(r)(P_1) \\
&\quad \text{by T12} \\
&\Leftrightarrow \quad \mathit{proj}(r(A_1))(\mathit{rename}(r)(x)) \in \mathit{rename}(r)(P_1) \\
&\quad \text{since } y = \mathit{rename}(r)(x) \\
&\Leftrightarrow \quad \mathit{proj}(r(A_1))(y) \in \mathit{rename}(r)(P_1).
\end{aligned}$$

Similarly, $\mathit{proj}(A_2)(x) \in P_2$ if and only if $\mathit{proj}(r(A_2))(y) \in \mathit{rename}(r)(P_2)$. Therefore, $y \in P$ if and only if $y \in P'$. □

Lemma 4.12. Trace structure algebras satisfy A25.

Proof: Let B be an alphabet. We wish to show that for all trace structures $p_1 = (A_1, P_1)$ and $p_2 = (A_2, P_2)$ such that $A_1 \cap A_2 \subseteq B$,

$$\mathit{proj}(B)(p_1 \parallel p_2) = \mathit{proj}(B)(p_1) \parallel \mathit{proj}(B)(p_2).$$

By T1 and definition 4.4, both sides of the equation are always defined. We now prove that if $A_1 \cap A_2 \subseteq B$, then they are also equal. Clearly $B \cap (A_1 \cup A_2) = (B \cap A_1) \cup (B \cap A_2)$, so that

$$\begin{aligned} p &= (B \cap (A_1 \cup A_2), P) = \textit{proj}(B)(p_1 \parallel p_2) \\ p' &= ((B \cap A_1) \cup (B \cap A_2), P') = \textit{proj}(B)(p_1) \parallel \textit{proj}(B)(p_2). \end{aligned}$$

have the same alphabet. In addition,

$$\begin{aligned} P &= \{y \in \mathcal{B}(B \cap (A_1 \cup A_2)) : \exists x \in \mathcal{B}(A_1 \cup A_2)\ [y = \textit{proj}(B \cap (A_1 \cup A_2))(x) \\ &\quad \wedge \textit{proj}(A_1)(x) \in P_1 \wedge \textit{proj}(A_2)(x) \in P_2]\} \\ P' &= \{y \in \mathcal{B}(B \cap (A_1 \cup A_2)) : \textit{proj}(B \cap A_1)(y) \in \textit{proj}(B)(P_1) \\ &\quad \wedge \textit{proj}(B \cap A_2)(y) \in \textit{proj}(B)(P_2)\} \end{aligned}$$

We wish to show that $y \in P$ if and only if $y \in P'$.

For the forward direction, let $y \in \mathcal{B}(B \cap (A_1 \cup A_2))$ be such that $y \in P$. Then there exists $x \in \mathcal{B}(A_1 \cup A_2)$ such that

$$y = \textit{proj}(B \cap (A_1 \cup A_2))(x)$$

and

$$\begin{aligned} \textit{proj}(A_1)(x) &\in P_1 \\ \textit{proj}(A_2)(x) &\in P_2 \end{aligned}$$

Then

$$\begin{aligned} &\textit{proj}(B \cap A_1)(y) \\ &\quad = \textit{proj}(B \cap A_1)(\textit{proj}(B \cap (A_1 \cup A_2))(x)) \\ &\quad\ \text{by T4} \\ &\quad = \textit{proj}(B \cap A_1 \cap B \cap (A_1 \cup A_2))(x) \\ &\quad = \textit{proj}(B \cap A_1)(x). \end{aligned}$$

But since $\textit{proj}(A_1)(x) \in P_1$, clearly by T4, $\textit{proj}(B \cap A_1)(y) \in \textit{proj}(B)(P_1)$. Similarly, $\textit{proj}(B \cap A_2)(y) \in \textit{proj}(B)(P_2)$. Therefore, $y \in P'$.

For the reverse direction, refer to figure 4.2. Let $y \in \mathcal{B}(B \cap (A_1 \cup A_2))$ be

$$\begin{tikzcd}
 & & x \in \mathcal{B}(A_1 \cup A_2) \arrow[dl] \arrow[dr] & & \\
 & z_1' \in \mathcal{B}((B \cap (A_1 \cup A_2)) \cup A_1) \arrow[dl] \arrow[dr] & & z_2' \in \mathcal{B}((B \cap (A_1 \cup A_2)) \cup A_2) \arrow[dl] \arrow[dr] & \\
z_1 \in \mathcal{B}(A_1) \arrow[dr] & & y \in \mathcal{B}(B \cap (A_1 \cup A_2)) \arrow[dl] \arrow[dr] & & z_2 \in \mathcal{B}(A_2) \arrow[dl] \\
 & \textit{proj}(B \cap A_1)(y) \in \mathcal{B}(B \cap A_1) & & \textit{proj}(B \cap A_2)(y) \in \mathcal{B}(B \cap A_2) &
\end{tikzcd}$$

Figure 4.2: Proving A25

such that $y \in P'$. Then

$\textit{proj}(B \cap A_1)(y) \in \textit{proj}(B)(P_1)$

by T3

$\Leftrightarrow \quad \textit{proj}(B \cap A_1)(y) \in \textit{proj}(B \cap A_1)(P_1)$

$\Leftrightarrow \quad \exists z_1 \in P_1[\textit{proj}(B \cap A_1)(y) = \textit{proj}(B \cap A_1)(z_1)]$

by T8, since $B \cap A_1 = (B \cap (A_1 \cup A_2)) \cap A_1$,

$\Rightarrow \quad \exists z_1' \in \mathcal{B}((B \cap (A_1 \cup A_2)) \cup A_1))[\textit{proj}(B \cap (A_1 \cup A_2))(z_1') = y$

$\wedge\, \textit{proj}(A_1)(z_1') = z_1]$

Similarly

$$\begin{aligned}
& \textit{proj}(B \cap A_2)(y) \in \textit{proj}(B)(P_2) \\
& \quad \Rightarrow \quad \exists z_2' \in \mathcal{B}((B \cap (A_1 \cup A_2)) \cup A_2))[\textit{proj}(B \cap (A_1 \cup A_2))(z_2') = y \\
& \qquad\quad \wedge \textit{proj}(A_2)(z_2') = z_2]
\end{aligned}$$

Since $A_1 \cap A_2 \subseteq B$, $((B \cap (A_1 \cup A_2)) \cup A_1) \cap ((B \cap (A_1 \cup A_2)) \cup A_2) = B \cap (A_1 \cup A_2)$. Therefore, by T8, since $((B \cap (A_1 \cup A_2)) \cup A_1) \cup ((B \cap (A_1 \cup A_2)) \cup A_2) \subseteq A_1 \cup A_2$, there exists $x \in \mathcal{B}(A_1 \cup A_2)$ such that

$$\begin{aligned}
z_1' &= \textit{proj}((B \cap (A_1 \cup A_2)) \cup A_1)(x) \\
z_2' &= \textit{proj}((B \cap (A_1 \cup A_2)) \cup A_2)(x)
\end{aligned}$$

Therefore, by T4,

$$\begin{aligned}
y &= \textit{proj}(B \cap (A_1 \cup A_2))(x) \\
z_1 &= \textit{proj}(A_1)(x) \\
z_2 &= \textit{proj}(A_2)(x)
\end{aligned}$$

But then, since $z_1 \in P_1$ and $z_2 \in P_2$, $y \in P$. □

□

4.2.1 Signatures and Behaviors

The simple model of agent and of the operations on agents introduced in definition 4.2 and definition 4.3 ignores any consideration regarding the "interface" of an agent. In other words, an agent is simply a container of behaviors. The set of signals used by an agent, i.e., its alphabet, is one example of what we call the *signature* of an agent. The signature is a representation of the interface of an agent in terms of its signals, their role and their properties. We consistently use the term signature for this purpose, since the term interface is interpreted differently by different communities.

Signatures can themselves form an agent algebra. For instance, the alphabet algebra introduced in example 2.7 is an example of a signature algebra where each signature consists solely of a set of signals. The IO agent algebra (example 2.10) extends the alphabet algebra by classifying each signal as either an input or an output. In this case, parallel composition and projection must be restricted to avoid intuitively inconsistent or problematic operations. Finally, typed IO agents (example 2.12) is a signature algebra that includes typing information for each signal in the agent.

We view the signature and the behavior of an agent as orthogonal, although sometimes related, representations. In other words, the signature and the behavior are two incomparable abstractions of an agent. It is therefore natural to construct a complete model of an agent algebra by combining a signature algebra with a trace structure algebra. In particular, the product of a signature algebra Γ and a trace structure algebra $\mathcal{Q}$ is used to construct agents of the form (γ, p), where γ is the signature and p is the behavior of the agent. For consistency, we require that the alphabet of the signature and the alphabet of the corresponding trace structure be the same. Theorem 2.19 proves that the subset of agents of a product algebra that have this property is again an agent algebra.

4.2.2 Concatenation and Sequential Composition

In the presentation so far we have emphasized a kind of composition of agents that corresponds to their parallel execution. Many models of computation, however, also include the ability to compose agents in "sequence". This could be seen as a parallel composition where control flows from one agent to another, thus making only one agent active at a time. Nevertheless, this situation is so common that it warrants the introduction of some special operations and notation.

For these models we introduce a third operation on traces called *concatenation*, which corresponds to the sequential composition of behaviors. Similarly to the other operations, concatenation must also satisfy certain properties that ensure that its behavior is consistent with its intuitive interpretation. Other than that, the definition of concatenation depends upon the particular model of computation. Concatenation is also used to define the notion of a prefix of a trace. We say that a trace x is a prefix of a trace z if there exists a trace y such that z is equal to x concatenated with y.

Our treatment of concatenation and sequential composition is consistent with the one introduced by Burch [12]. In particular, Burch introduces a set of axioms that formalize the intuitive notion of concatenation and its properties. For example, concatenation is required to be associative, but is not required to be commutative. We do not reconsider those axioms here, and reserve a complete treatment of concatenation, including sequential composition and its consequences on conformance orders and mirrors for our future work.

With concatenation, we distinguish between a complete behavior and a partial behavior. A complete behavior has no endpoint. Since a complete behavior goes on forever, it does not make sense to talk about something happening "after" a complete behavior. A partial behavior has an endpoint; it can be a prefix of a complete behavior or of another partial behavior. Every complete behavior has partial behaviors that are prefixes of it; every partial behavior is a prefix of some complete behavior. The distinction between a complete behavior and a partial behavior has only to do with the length of the behavior (that is, whether or not it has an endpoint), not with what is happening during the behavior; whether an agent does anything, or what it does, is irrelevant.

Complete traces and *partial traces* are used to model complete and partial behaviors, respectively. A given object can be both a complete trace and a partial trace; what is being represented in a given case is determined from context. For example, a finite string can represent a complete behavior with a finite number of actions, or it can represent a partial behavior. In the following, we will denote the set of partial traces with alphabet A as $\mathcal{B}_P(A)$, and the set of complete traces with alphabet A as $\mathcal{B}_C(A)$.

As discussed above, concatenation induces a corresponding operation on trace structures that we call *sequential composition*. Because of the different nature of complete and partial traces, the definition of trace structures must be extended to contain a set of complete traces $P_C \subseteq \mathcal{B}_C(A)$ and a set of partial traces $P_P \subseteq \mathcal{B}_P(A)$, where A is the alphabet of the agent. We also denote with $P = P_C \cup P_P$ the set of all traces (consistently with the previous formulation). The sequential composition $p'' = p \cdot p'$ is then defined when $A = A'$, and in that case:

$$\begin{aligned} A'' &= A = A', \\ P''_C &= P_C \cup (P_P \cdot P'_C), \\ P''_P &= P_P \cdot P'_P. \end{aligned}$$

where concatenation is naturally extended to sets of traces. Note that the concatenation of a partial trace with a complete trace is a complete trace, while the concatenation of two partial traces is again partial. Because complete traces have no endpoint, the concatenation of a complete trace with a partial trace is not defined. As for parallel composition, the definition of sequential composition is constructed from equivalent concepts in the trace algebra. Therefore, the trace structure algebra can still be constructed automatically.

4.3 Models of Computation

In this section we will present examples of agent models that use signatures and trace structures as their building blocks. In section 1.6 we have introduced our motivating example (see figure 1.3), and informally studied a model of computation for continuous time by first considering its natural semantic domain, and then formalizing it in terms of traces and trace structures. The examples in this section are formalizations of the semantic domain of the remaining models of computation for that same example. In all cases we follow the same pattern by first presenting the natural semantic domain, and then the formalization in terms of trace algebras. For each model of computation we also sketch an example of its typical applications in terms of a subsystem of the PicoRadio architecture shown before. Later we will show how we can derive relationships between these models within the framework.

The proof that the trace algebras defined below satisfy the axioms of trace algebra is usually straightforward and typically follows directly from the definitions with minimal manipulation, with the exception of T8 that requires exhibiting a witness to the existential quantifier. However, our examples are usually fairly simple and it is easy to construct the right trace given the common projection of two other traces. We therefore omit the details of these proofs.

4.3.1 Hybrid Systems

The example presented in section 1.7 is a simple formalization of a continuous time model. Here we make the formalization more precise, and we also extend the model to not only cover continuous time behavior, but also hybrid continuous and discrete behavior.

A typical semantics for hybrid systems includes continuous *flows* that represent the continuous dynamics of the system, and discrete *jumps* that represent instantaneous changes of the operating conditions. In our model we represent both flows and jumps with single piece-wise continuous functions over real-valued time. The flows are continuous segments, while the jumps are discontinuities between continuous segments. We assume that the variables of the system take only real or integer values and we defer the treatment of a complete type system for future work. The sets of real-valued and integer-valued variables for a given trace are called $V_{\mathbb{R}}$ and $V_{\mathbb{Z}}$, respectively.

Traces may also contain actions, which are discrete events that can occur at any time. Actions do not carry data values. For a given trace, the set of input actions is M_I and the set of output actions is M_O.

The signature γ of each agent is a 4-tuple of the above sets of signals:

$$\gamma = (V_{\mathbb{R}}, V_{\mathbb{Z}}, M_I, M_O).$$

The sets of signals may be empty, but we assume they are disjoint. The alphabet of γ, and therefore the alphabet of an agent with signature γ, is

$$A = V_{\mathbb{R}} \cup V_{\mathbb{Z}} \cup M_I \cup M_O.$$

Later we will define the operations on signatures, as well as those on traces and agents. The signature defined here, together with its operations, is an extension of both the alphabet algebra of example 2.7 (in that the set of signals $V_{\mathbb{R}}$ and $V_{\mathbb{Z}}$ have no direction) and the IO agent algebra of example 2.10.

The set of partial traces for a signature γ is $\mathcal{B}_P(\gamma)$. Each element of $\mathcal{B}_P(\gamma)$ is as a triple $x = (\gamma, \delta, f)$. The non-negative real number δ is the *duration* (in time) of the partial trace. The function f has domain A. For $v \in V_{\mathbb{R}}$, $f(v)$ is a function in $[0, \delta] \to \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers and the closed interval $[0, \delta]$ is the set of real numbers between 0 and δ, inclusive. This function must be piece-wise continuous and right-hand limits must exist at all points. Analogously, for $v \in V_{\mathbb{Z}}$, $f(v)$ is a piece-wise constant function in $[0, \delta] \to \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. For $a \in M_I \cup M_O$, $f(a)$ is a function in $[0, \delta] \to \{0, 1\}$, where $f(a)(t) = 1$ if and only if action a occurs at time t in the trace.

The set of complete traces for a signature γ is $\mathcal{B}_C(\gamma)$. Each element of $\mathcal{B}_C(\gamma)$ is as a pair $x = (\gamma, f)$. The function f is defined as for partial traces, except that each occurrence of $[0, \delta]$ in the definition is replaced by $\mathbb{R}^{\not-}$, the set of non-negative real numbers.

To complete the definition of this trace algebra, we must define the operations of projection, renaming and concatenation on traces. The projection operation *proj*$(B)(x)$ is always defined and the trace that results is the same as x except that the domain of f is restricted to the elements that are in B. The renaming operation $x' = \textit{rename}(r)(x)$ is defined if and only if $A \subseteq \textit{dom}(r)$. If x is a partial trace, then $x' = (\gamma', \delta, f')$ where γ' results from using r to rename the elements of γ and $f' = r \circ f$.

The definition of the concatenation operator $x_3 = x_1 \cdot x_2$, where x_1 is a partial trace and x_2 is either a partial or a complete trace, is more complicated. If x_2 is a partial trace, then x_3 is defined if and only if $\gamma_1 = \gamma_2$ and for all $a \in A$,

$$f_1(a)(\delta_1) = f_2(a)(0)$$

(note that δ_1, δ_2, etc., are components of x_1 and x_2 in the obvious way). When defined, $x_3 = (\gamma_1, \delta_3, f_3)$ is such that $\delta_3 = \delta_1 + \delta_2$ and for all $a \in A$

$$f_3(a)(\delta) = \begin{cases} f_1(a)(\delta) & \text{if } 0 \le \delta \le \delta_1 \\ f_2(a)(\delta - \delta_1) & \text{if } \delta_1 \le \delta \le \delta_3. \end{cases}$$

Note that concatenation is defined only when the end points of the two traces match. The concatenation of a partial trace with a complete trace yields a complete trace with a similar definition. If $x_3 = x_1 \cdot x_2$, then x_1 is a *prefix* of x_3.

Trace structures in this model have again signature γ and are constructed as usual as sets of partial and complete traces.

4.3.2 Non-metric Time

In the definition of this trace algebra we are concerned with the order in which events occur in the system, but not in their absolute distance or position. This is useful if we want to describe the semantics of a programming language for hybrid systems that abstracts from a particular real time implementation.

Although we want to remove real time, we want to retain the global ordering on events induced by time. In particular, in order to simplify the abstraction from metric time to non-metric time described below, we would like to support the case of an uncountable number of events[1]. Sequences are clearly inadequate given our requirements. Instead we use a more general notion of a partially ordered multiset to represent the trace. We repeat the definition found in [76], and due to Gischer, which begins with the definition of a labeled partial order.

Definition 4.13 (Labeled Partial Order). A *labeled partial order* (lpo) is a 4-tuple $(V, \Sigma, \leq, \mu)$ consisting of

1. a *vertex set* V, typically modeling *events*;
2. an *alphabet* Σ (for symbol set), typically modeling *actions* such as the arrival of integer 3 at port Q, the transition of pin 13 of IC-7 to 4.5 volts, or the disappearance of the 14.3 MHz component of a signal;
3. a *partial order* $\leq$ on V, with $e \leq f$ typically being interpreted as event e necessarily preceding event f in time; and
4. a *labeling function* $\mu : V \mapsto \Sigma$ assigning symbols to vertices, each labeled event representing an *occurrence* of the action labeling it, with the same action possibly having multiple occurrence, that is, μ need not be injective.

[1] In theory, such Zeno-like behavior is possible, for example, for an infinite loop whose execution time halves with every iteration.

A *pomset* (partially ordered multiset) is then the isomorphism class of an lpo, denoted $[V, \Sigma, \leq, \mu]$. By taking lpo's up to isomorphism we confer on pomsets a degree of abstractness equivalent to that enjoyed by strings (regarded as finite linearly ordered labeled sets up to isomorphism), ordinals (regarded as well-ordered sets up to isomorphism), and cardinals (regarded as sets up to isomorphism).

This representation is suitable for the above mentioned infinite behaviors: the underlying vertex set may be based on an uncountable total order that suits our needs. For our application, we do not need the full generality of pomsets. Instead, we restrict ourselves to pomsets where the partial order is total, which we call *tomsets*.

It is easy to define a non-metric trace algebra using tomsets. Traces have the same form of signature as in metric time model of the previous section:

$$\gamma = (V_{\mathbb{R}}, V_{\mathbb{Z}}, M_I, M_O).$$

Both partial and complete traces are of the form $x = (\gamma, L)$ where L is a tomset. When describing the tomset L of a trace, we will in fact describe a particular lpo, with the understanding that L is the isomorphism class of that lpo. An action $\sigma \in \Sigma$ of the lpo is a function with domain A such that for all $v \in V_{\mathbb{R}}$, $\sigma(v)$ is a real number (the value of variable v resulting from the action σ); for all $v \in V_{\mathbb{Z}}$, $\sigma(v)$ is an integer; and for all $a \in M_I \cup M_O$, $\sigma(v)$ is either 0 or 1. The underlying vertex set V, together with its total order, provides the notion of time, a space that need not contain a metric. For both partial and complete traces, there must exist a unique minimal element $\min(V)$. The action $\mu(\min(V))$ that labels $\min(V)$ should be thought of as giving the initial state of the variables in $V_{\mathbb{R}}$ and $V_{\mathbb{Z}}$. For each partial trace, there must exist a unique maximal element $\max(V)$ (which may be identical to $\min(V)$).

Notice that, as defined above, the set of partial traces and the set of complete traces are not disjoint. It is convenient, in fact, to extend the definitions so that traces are labeled with a bit that distinguishes partial traces from complete traces, although we omit the details.

According to this definition, it is possible for a trace to exhibit stutters. A *stutter* occurs in a trace when two consecutive vertices in the vertex set are mapped onto the same label. In the case of non-metric time we are interested in the first occurrence of an event, and not in its repetitions. For that reason, and for other technical reasons, we define an operation of *stutter removal* that takes a non-metric time trace that possibly contains stutters, and produces a non-metric time trace without stutters.

The key to defining this operation is a formalization of the intuitive notion of *consecutive* vertices in the vertex set. This notion fails in the case of dense or continuous vertex sets, where it is impossible to identify two vertices such that no other vertex is contained between them in the order relation. Instead, we define an equivalence relation, called *stutter equivalence*, that partitions the entire vertex set into its set of stutters.

Definition 4.14 (Stutter Equivalence). Let L be an lpo and V be its vertex set. Then $v_1, v_2 \in V$ are stutter equivalent if and only if for all $v \in V$ such that $v_1 \leq v \leq v_2$, $\sigma(v) = \sigma(v_1)$.

This relation is clearly reflexive, symmetric and transitive, and is therefore an equivalence relation on the vertex set.

In particular we can define an lpo based on the set of equivalence classes generated by the equivalence relation. For an equivalence class $[v]$, we define $\sigma([v]) = \sigma(v)$. The function is well defined. In fact, for all $v_1 \in [v]$ and $v_2 \in [v]$, $\sigma(v_1) = \sigma(v_2)$, therefore the definition is independent of the particular representative of the equivalence class. Analogously, if $[v_1]$ and $[v_2]$ are two equivalence classes, then we define $[v_1] \leq [v_2]$ if and only if $[v_1] = [v_2]$ or $v_1 \leq v_2$. This relation is also well defined, since each equivalence class is essentially an interval in the original lpo. The structure composed of the set $[V]$ of equivalence classes, the induced total order and the induced labeling function thus constitute an lpo without stutters. Note that this lpo is nothing more than the quotient structure of the lpo with stutters with respect to stutter equivalence.

It is easy now to define the operation of stutter removal simply as as the process of taking a tomset, an lpo representing the tomset, its quotient structure relative to stutter equivalence, and finally the isomorphism class. To prove that this operation is well defined, we must show that the result is independent of the particular choice of lpo taken as a representative of the tomset. This is a rather technical argument, and we omit the details.

A tomset is *stutter free* if and only if it has no stutters, i.e., the equivalence classes under stutter equivalence are all singletons. This is always the case after the application of stutter removal. Therefore we define a non-metric time trace as a stutter free tomset.

By analogy with the metric time case, it is straightforward to define projection and renaming on actions $\sigma \in \Sigma$. This definition can be easily extended to lpo's and, thereby, traces. Projection, however, must be followed by an additional operation of stutter removal, since hiding certain signals from the trace may expose stutters that were not present before.

The concatenation operation $x_3 = x_1 \cdot x_2$ is defined if and only if x_1 is a partial trace, $\gamma_1 = \gamma_2$ and $\mu_1(\max(V_1)) = \mu_2(\min(V_2))$. When defined, the vertex set V_3 of x_3 is a disjoint union:

$$V_3 = V_1 \uplus (V_2 - \min(V2))$$

ordered such that the orders of V_1 and V_2 are preserved and such that all elements of V_1 are less than all elements of V_2. The labeling function is such that for all $v \in V_3$

$$\begin{aligned} \mu_3(v) &= \mu_1(v) \text{ for } \min(V_1) \leq v \leq \max(V_1) \\ \mu_3(v) &= \mu_2(v) \text{ for } \max(V_1) \leq v. \end{aligned}$$

Analogously to the operation of stutter removal, the operations of projection, renaming and concatenation are well defined only if the result is independent of the particular choice of representative of the lpo's involved. Again, we omit the details of this proof.

Trace structures are constructed, as usual, as sets of traces. In particular, the operation of parallel composition is defined in terms of the projection operation. It is interesting to note that parallel composition need not be followed by stutter removal, since the composition of stutter free trace structures is again stutter free.

4.3.3 CSP

Communicating Sequential Processes were introduced by Hoare [50]. It consists of a collection of agents that interact through the exchange of actions. Actions are shared and must be synchronized: when an agent wishes to perform an action with another agent, it must wait until the other agent is ready to perform the same action.

CSP is particularly well suited to handle cases where a tight synchronization is required or to schedule access to a shared resource. In our example we can use CSP to model a manager subsystem that regulates access to a set of parameters and tables that can be set and read by the user and by the protocol stack. To do this, the manager initially waits to synchronize with either the protocol stack or the user input; once synchronized with one of the two parties, it reserves the shared resource and handles the communication by performing a set of actions (e.g., read, write, update). At the end of the transaction, the manager goes back to its initial state and waits to synchronize again. Figure 4.3 shows a diagram of this subsystem.

Constructing a trace algebra and a trace structure algebra for this model is particularly simple because the communication model fits very easily in our framework. A single execution of an agent (a trace) is simply a sequence of actions from the alphabet A of possible actions. Formally,

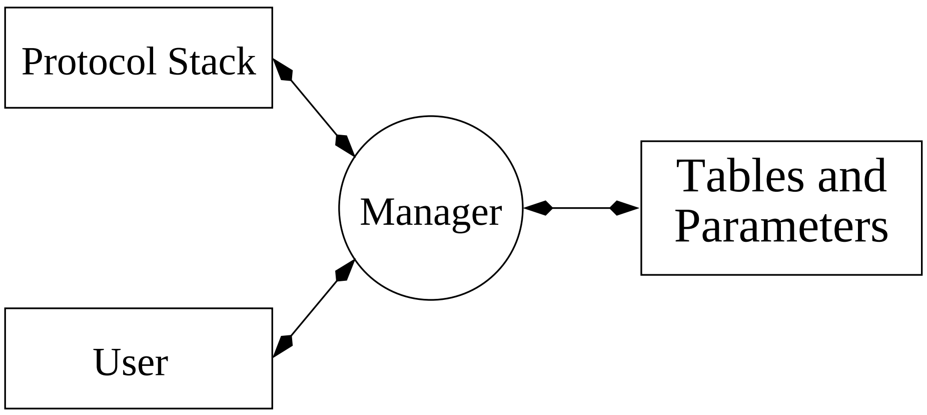

Figure 4.3: Table manager and UI interface

we define

$$\mathcal{B}(A) = A^{\infty},$$

where the notation A^{∞} includes both the finite and the infinite sequences over A. Projection and renaming are defined as expected: if $x \in \mathcal{B}(A)$, then $\mathit{proj}(B)(x)$ is the sequence formed from x by removing every symbol a not in B. More formally, if $x' = \mathit{proj}(B)(x)$, then the length of x' (written $\mathit{len}(x')$) is

$$\mathit{len}(x') = |\{j \in \mathbb{N} : 0 \leq j < \mathit{len}(x) \wedge x(j) \in B\}|$$

where $\mathit{len}(x') = \omega$ when the set is infinite. The k-th element of x' corresponds to the k-th element of x that belongs to B. Hence, if $x(n) \in B$, then $x'(k) = x(n)$ where

$$k = |\{j \in \mathbb{N} : 0 \leq j < n \wedge x(j) \in B\}|.$$

Note that any n and k combination is unique.

For renaming, assume without loss of generality that $x \in \mathcal{B}(A)$ is of the form

$$x = \langle a_0, a_1, a_2, \ldots \rangle;$$

then

$$\mathit{rename}(r)(x) = \langle r(a_0), r(a_1), r(a_2), \ldots \rangle.$$

The same can be restated more formally as

$$\mathit{rename}(r)(x) = \lambda n \in \mathbb{N}\, [r(x(n))].$$

Models of agents are obtained in the standard way, as a collection of sequences. For the CSP model we use the IO agent algebra of example 2.10 as the signature algebra, so that the signature γ of each agent includes a set of input actions I and a set of output actions O. Given the definition of projection, parallel composition (see definition 4.4) clearly requires that trace structures (agents) synchronize on the shared actions. Additionally, since we are taking the product with the IO agent algebra, the parallel composition is defined only if the agents that are being composed have disjoint sets of output actions.

This model is based solely on actions that bear no value. It is straightforward to extend the model to include a value for each action. We define:

$$\mathcal{B}(A) = (A \times V)^{\infty},$$

where V is the set of possible values. Projection and renaming are extended by having them act only on the first component of the pair. Formally, if $x \in \mathcal{B}(A)$ and $x' = \mathit{proj}(B)(x)$ then the length of x' (written $\mathit{len}(x')$) is

$$\mathit{len}(x') = |\{j \in \mathbb{N} : 0 \leq j < \mathit{len}(x) \wedge x(j) \in B \times V\}|$$

and $x'(k) = x(n)$ for all $k < \mathit{len}(x')$, where n is the unique integer such that $x(n) \in (B, V)$ and

$$k = |\{\, j \in \mathbb{N} : 0 \leq j < n \wedge x(j) \in B \times V\}|.$$

Likewise for renaming. Without loss of generality, assume

$$x = \langle (a_0, v_0), (a_1, v_1), \ldots \rangle;$$

then

$$\mathit{rename}(r)(x) = \langle (r(a_0), v_0), (r(a_1), v_1), \ldots \rangle,$$

or, equivalently

$$\mathit{rename}(r)(x) = \lambda n \in \mathbb{N}\, [(r(x_1(n)), x_2(n))],$$

With this definition we can construct a trace structure that represents the table manager depicted in figure 4.3. The signature γ includes inputs and outputs to and from both the protocol stack and the user interface, with actions that set and read the appropriate parameters. For example, the parameters could be a set of virtual connections, specified as pair of addresses (*vci* and *vpi*) and the packet length. Two typical traces for the manager deal with handling requests from the protocol and from the user, as in

$$\begin{aligned} P &= \{< \text{ps\_req}, \text{vci}(10), \text{vpi}(13), \text{ps\_release}, \ldots >, \\ &\quad < \text{user\_req}, \text{length}(1500), \text{vpi}(0), \ldots >, \ldots\}, \end{aligned}$$

where *ps* refers to the protocol stack, and *user* to the user interface. Note that while the manager can non-deterministically choose to serve the protocol stack or the user, it must continue to serve the party that was chosen until the shared resource is released.

Compared to the traditional CSP model, ours differ in some respects. For example, in our model it is possible for several agents to synchronize on the same action, thus making it possible for one agent to *broadcast* an event. In a more traditional model, only one of the listeners is able to react to the event. This is a consequence of our definition of parallel composition.

Another difference is that in our model (and in all other models constructed using trace algebras), the operation of parallel composition and renaming are clearly differentiated. In other words, parallel composition in our model does not *create* the connections, but is limited to constructing an agent whose projections are compatible with the ones being composed. Renaming must be invoked separately (and before the composition) to create the appropriate instances of the agents to be composed (see also the discussion on the operators in section 1.4).

4.3.4 Process Networks

Process networks are collections of agents that operate on infinite streams of data [53, 54, 32]. Streams are traditionally implemented as FIFO queues that connect processes that can produce (write) and consume (read) tokens. Process networks are particularly well suited to modeling digital signal processing applications, given the good match between the typical data model of signal processing and the communication model of process networks.

As an example we might consider a demodulator that uses a local reference to convert an incoming signal from high to base band. The decoder receives a stream of tokens that corresponds to, for instance, the output of the local oscillator described above in subsection 4.3.1. At the same time it receives a stream of data tokens to be demodulated. The demodulator combines the two streams and then applies a filter to retain only the component of interest. A diagram of this subsystem is shown in figure 4.4.

The important property of this model is that the exact time at which tokens arrive at the input is irrelevant, and that only their order within the same stream determines the output stream

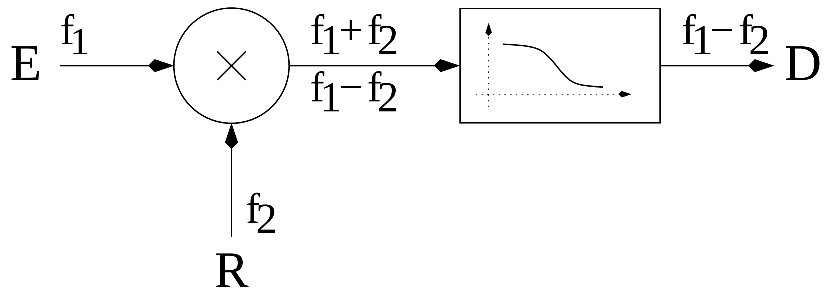

Figure 4.4: A signal demodulator

(together, of course, with their value). The natural domain for this kind of model is then clearly that of a function on streams, which can in turn be formalized as sequences. In the case of our demodulator, if we denote with R and E the reference and the modulated streams, and with D the demodulated stream, we can represent the decoder in the natural domain as a function f from the inputs to the output:

$$D = f(R, E).$$

Parallel composition of agents is defined by composing for each stream s the function whose range is s with the function whose domain is s. This definition becomes circular in the presence of loops in the structure of the parallel composition. In this case, the composition is defined by breaking the loop at some point, and then looking for the fixed points of the function that results. If we do not restrict the range of the possible functions f, the parallel composition may have several fixed points (or even no fixed points at all), and hence exhibit non-deterministic behavior. Because we ultimately want to model physical processes that *are* deterministic, we must impose some constraints on f. These are well known properties required of process networks (see, for example, the excellent presentation by Lee et al. [62]). Here we show how they impact the construction of the semantic domain in our framework.

We say that a stream v is a *prefix* of a stream u if v is equal to some initial segment of u. This relation can be extended to sets of streams by requiring that all streams in the first set be a prefix of the corresponding stream in the second set. This relation is easily proved to be a partial order on the streams.

To ensure that a composition of stream functions is determinate, the function f of each of the components must be *continuous* with respect to the prefix ordering on the streams. If that is case, then we are assured that there exists a unique least fixed point, and the parallel composition is defined in terms of that. In addition, continuity implies monotonicity, which in turn ensures that the response of the system to a specific input can be computed incrementally from progressively longer prefixes.

In the following we will show two ways of describing the process networks model in our framework. The first method is closer to the semantic domain based on functions on streams, but falls short in the definition of parallel composition. The second method fixes this problem, at the expense of modeling the traces at a more detailed level of abstraction.

In our initial attempt we follow the natural semantic rather closely. Similarly to the CSP model, we use the IO agent algebra of example 2.10 as the signature algebra, since process networks clearly distinguish between inputs and outputs. In the example above, we have

$$\begin{aligned} I &= \{R, E\}, \\ O &= \{D\}. \end{aligned}$$

Given a stream function, a trace is a single application from a set of input streams to a set of output streams. If we define the alphabet of a trace to be the set $A = I \cup O$, and formalize streams as the finite and infinite sequences over a value domain V, denoted by V^∞, then the set of all possible traces is

$$\mathcal{B}(A) = A \to V^\infty.$$

As usual, a trace structure is simply the signature together with a set of traces, i.e., $p = (\gamma, P)$ where $P \subseteq A \to V^\infty$. If we separate the contributions of the inputs and the outputs, the set P of traces can be seen as (is isomorphic to) a subset of

$$(I \to V^\infty) \times (O \to V^\infty),$$

that is, as a function on streams.

In order to comply with the process network model, we also insist that the functions so identified have the necessary continuity and monotonicity properties with respect to the prefix ordering defined on the sequences. In other words, not all sets of traces may form a trace structure.

We define a *functional* trace structure as one that associates at most one output stream to each input stream. More formally, the condition is equivalent to requiring that

$$\textit{proj}\,(I)(x) = \textit{proj}\,(I)(y) \Rightarrow x = y,$$

for all traces $x, y \in P$. To define monotonicity we first need a partial order on traces. We say that a trace $x \in \mathcal{B}(A)$ is a *prefix* of a trace $y \in \mathcal{B}(A)$, written $x \sqsubseteq y$, if $x(a)$ is a prefix of $y(a)$ for all $a \in A$. Let $p = (\gamma, P)$ be a trace structure. Then p is monotonic if for all $x, y \in P$,

$$\mathit{proj}(I)(x) \sqsubseteq \mathit{proj}(I)(y) \Rightarrow \mathit{proj}(O)(x) \sqsubseteq \mathit{proj}(O)(y).$$

Note that, in particular, this also implies

$$\mathit{proj}(I)(x) \sqsubseteq \mathit{proj}(I)(y) \Rightarrow x \sqsubseteq y.$$

Finally we define the process network trace structure algebra as the algebra that contains all and only the functional and monotonic trace structures.

The operations of projection and renaming on traces are easily defined. If $x \in \mathcal{B}(A)$, B is an alphabet and r is a renaming function, then

$$\begin{aligned} \mathit{proj}(B)(x) &= \lambda a \in B \cap A\, [x(a)], \\ \mathit{rename}(r)(x) &= \lambda a \in A\, [x(r(a))]. \end{aligned}$$

Parallel composition on trace structures is defined as usual in terms of the projection operation. Note that the trace structure obtained from a composition contains all the traces that are compatible with the agents being composed; in particular, it will contain all the fixed-points in a composition that involves a feedback loop. Figure 4.5 illustrates the point. Here two instances of the trace structure I are composed so that the input of one corresponds to the output of the other. The trace structures are also defined to be the identity function on streams, i.e., they contain all pairs of identical input and output streams. It is easy to show that also the composition contains all pairs of identical streams. This is a problem, as it doesn't faithfully represent the semantics of the original formulation of process networks, that in this case includes only empty streams, the least of the fixed-points in the composition. The problem with our model is that whether a trace is included in the composition or not depends exclusively on whether its projections are part of the individual components. In order to include only the least fixed-point, we would also need to check whether other traces (more specifically, prefixes) are also included in the composition.

Our solution to this problem avoids changing the definition of parallel composition (which is common to all trace structure algebras), but requires us to develop a new semantic domain at a more detailed level of abstraction. The additional information is sufficient to determine the result of the parallel composition exactly. Note that we could define the semantic domain in its exact natural

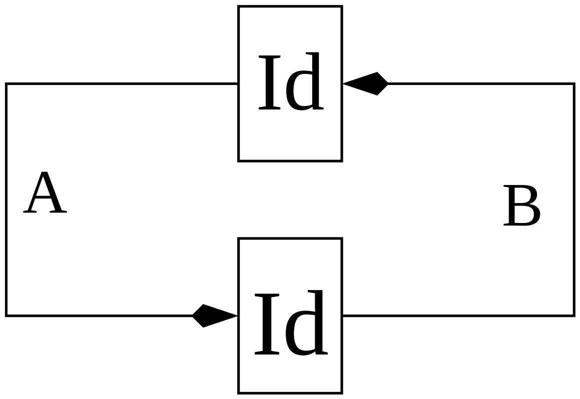

Figure 4.5: Parallel composition with feedback

form in the more general framework of agent algebra. The framework of trace algebra, however, provides us more flexibility in deriving relationships between agent models, as will be shown later.

In the new formalization, each trace is a totally ordered sequence of events. Formally we have:

$$\mathcal{B}(A) = (A \times V)^{\infty}.$$

Note that this is exactly the definition that we have for the semantic domain for communicating sequential processes. The definition of projection and renaming also parallels the definitions given in subsection 4.3.3, and will not be repeated here. The signature of the trace structures is again a pair of disjoint sets $\gamma = (I, O)$ as before. Despite the similarities with CSP, this formulation results in a different model of computation because the class of trace structures that we construct must satisfy some additional conditions, as was also the case in our initial formalization of process networks.

In the new formulation, the traces in the trace algebra carry order information for all events. This means that we can tell whether an input (or an output) event occurred before or after another input or output event. Because the semantics of process networks is independent of this ordering, a trace structure must contain traces that represent all orderings of inputs and outputs that are compatible with a particular stream function. The word "compatible" here has two meanings. First we must only include those orderings that result in monotonic functions. Second, inputs and outputs can not occur arbitrarily ordered in a trace: output tokens should never precede the input tokens that caused them. The rest of this section makes these two requirements more precise.

It is easy to construct a homomorphism h to the previous trace algebra that loses the ordering information. Given a trace x in the alphabet A, we isolate the sequence relative to a signal

a using a projection operation, and then construct the appropriate function. More formally:

$$h(x) = \lambda a \in A \; \lambda n \in \mathbb{N} \; [(\mathit{proj}(\{a\})(x))(n))_v],$$

where the subscript v denotes the second component of a pair in $A \times V$. This function is a homomorphism in that it commutes with the application of the other operations on traces, projection and renaming.

The functionality and monotonicity conditions are best expressed in the domain of stream functions, as we don't want the particular order of a trace to affect the prefix relation. A functional trace structure can be defined as follows. For all $x, y \in P$, the following condition must be satisfied:

$$h(\mathit{proj}(I)(x)) = h(\mathit{proj}(I)(y)) \Rightarrow h(x) = h(y).$$

Similarly for monotonicity. If $p = (\gamma, P)$, then p is monotonic if for all $x, y \in P$,

$$h(\mathit{proj}(I)(x)) \sqsubseteq h(\mathit{proj}(I)(y)) \Rightarrow h(\mathit{proj}(O)(x)) \sqsubseteq h(\mathit{proj}(O)(y)).$$

In order to include all orderings in the trace structures, we might be tempted to state that if $x \in P$, then any other trace y such that $h(y) = h(x)$ should be in P. Doing this would remove all information regarding the ordering of inputs and outputs. As a result, the composition would again suffer from the same problem (inclusion of all of the fixed-points) that we had with the previous model. Instead, we must strengthen this condition.

We do this in two steps. Given a trace structure $p = (\gamma, P)$, we first look for a subset $P_0 \subseteq P$ of only those traces that can be characterized as *quiescent*, in the sense that all the outputs relative to the inputs have been produced. In fact, we are looking for the set P_0 with the added property that the outputs occur in the sequence as soon as possible. This is similar to the *fundamental model* assumption in asynchronous design. In the formalization that follows, we will assume that tokens have no value to simplify the notation. Under this assumption, P_0 can be formalized as follows:

$$\mathcal{P}_0 = \{ z \in P : \forall x, y \in \mathcal{B}(A) \; \forall b \in I \; [z = x\langle b\rangle y \Rightarrow x \in P]\},$$

where the notation $\langle b\rangle$ denotes the sequence made of only the symbol b. The intuition behind this definition is as follows. Assume that a trace $z \in P$ can be written as the concatenation $x\langle b\rangle y$ with $x \in P$. Then, since p is functional, for any trace x' such that $h(\mathit{proj}(I)(x')) = h(\mathit{proj}(I)(x))$, we have $h(\mathit{proj}(O)(x')) = h(\mathit{proj}(O)(x))$. So, in particular, none of the output tokens that are contained in the suffix y ever occur before the input token b in any other trace in P with the same inputs as x. If y starts with an output token c, this condition tells us that c does not appear any sooner in any other trace, and therefore that z outputs c as soon as possible. The universal quantification on x, y and b extends the property to the entire trace z.

Since P_0 is the "fastest" subset of P, we can now construct a new set that includes all possible delays of the output. We construct this set by induction. Given a set X of traces, we define a function F that adds all traces where each output that precedes an input is delayed by one position. Formally:

$$F(X) = X \cup \{\, x\langle b, c\rangle y \in \mathcal{B}(A) : x\langle c, b\rangle y \in X \wedge b \in I \wedge c \in O\}. \tag{4.1}$$

Intuitively we would like to repeatedly apply this function starting from P_0 until we reach a fixed-point. This function is monotonic relative to set containment (given $X_1 \subseteq X_2$, $F(X_2)$ will add at least the traces that $F(X_1)$ adds, plus possibly some more). In addition, F creates progressively larger sets, i.e.,

$$\forall X \; [X \subseteq F(X)].$$

When this is the case, we say that F is *inflationary* at X. These two properties are enough to guarantee the existence of a fixed-point [91]. In fact, they guarantee the existence of a *least* fixed-point greater than or equal to P_0, the minimal set that contains P_0 and all the traces with delayed outputs[2]. Let's denote with $P_{\text{fp}}(P)$ the fixed-point obtained by starting the recursion with the P_0 associated to P. Then we define the trace structure algebra for process networks as the one that contains only those trace structures such that

$$P = P_{\text{fp}}(P).$$

The system shown in figure 4.5 now results in a correct composition. In fact, the bottom trace structure I will require that the input at A appear before any output on B in all its traces. Likewise, the top trace structure will require its input, which corresponds to B, to occur before the output A. This contradiction will rule out all traces except the empty one, as dictated by the least fixed-point semantics.

4.3.5 Discrete Event

A discrete event system consists of agents that interact by exchanging *events* on a set of signals. Each event is labeled with a *time stamp* that denotes the time at which the event occurred.

[2]Technically it is the greatest lower bound of the set of fixed-points of F that are greater than or equal to P_0.

The notion of time is global to the entire system, so that if any two events have the same time stamp then they are considered to occur at the same time. The set of time stamps is often taken to be the set of positive integers or real numbers, ordered by the usual order. The order is then extended to the events so that events with smaller time stamps precede events with higher time stamps. The model is called *discrete* because it is required that for each signal the set of time stamps is not dense in the reals.

Examples of discrete event systems abound, as both Verilog [93] and VHDL [2] use this model as their underlying simulation semantics. For our example, we might consider the subsystem that implements the protocol stack that handles the data stream after it has been demodulated. The stack includes functions that modify and depend on the tables and parameters managed by the subsystem described in the section on CSP (subsection 4.3.3). In addition, the protocol stack interacts with the physical layer at the lower levels, and then unpacks and delivers the raw data to the application. The non-recurring nature of these operations, their unpredictable timing and the dependency of the protocol behavior on their timing make a discrete event model more suitable than, say, a data-flow model. A typical protocol stack of four layers is shown in figure 4.6.

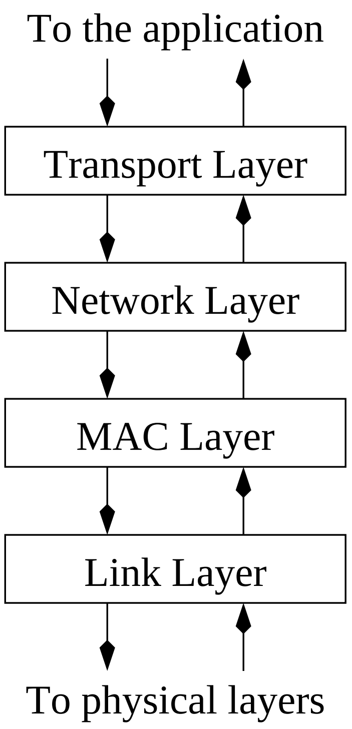

Figure 4.6: Protocol Stack

In the natural semantic domain, each behavior of an agent can be characterized as a sequence of events, each associated to an increasing time stamp. Note that events that occur in unrelated parts of the system are still ordered by their time stamps. Different events may occur with the same time stamp. In most cases, if two discrete event models differ at all, they differ in the way events with the same time stamp are handled. For the purpose of simulating such systems, some models define a notion of a *delta cycle* that orders the events with identical time stamps. Others don't define any specific way to handle this occurrence, leading to non-determinism.

It is natural to construct a semantic domain in our framework based on the interpretation of a behavior as a sequence of events with time stamps. If A is the set of signals, V the set of values and $\mathbb{R}^{+}$ the set of non-negative reals, we define the traces as follows:

$$\mathcal{B}(A) = (A \times V \times \mathbb{R}^{+})^{\infty}.$$

Two conditions must be imposed on the time stamps of a trace. First, the time stamps in the sequence must be non-decreasing, i.e., if x is a trace and n and m are two natural numbers such that $n, m < \mathit{len}(x)$, then

$$n < m \Rightarrow x(n)_t \leq x(m)_t,$$

where the subscript t denotes the time stamp of the event. Second, the time stamps of an infinite sequence x must be divergent, i.e., for all $t \in \mathbb{R}^{+}$, there is an event in x with time stamp greater than t. Discreteness can be enforced by requiring that for all non-negative reals $t \in \mathbb{R}^{+}$, there is only a finite number of events in x such that $x(n)_t < t$. Projection and renaming are defined similarly to the functions defined for CSP in subsection 4.3.3.

The signature γ of a trace structure is taken from the IO agent algebra, and therefore it distinguishes between the set of inputs I and the set of output O, that together form the alphabet A. Trace structures are then built as a signature with a set of traces in a way similar to the models that we have already presented. Constraints can be imposed on the set P of traces of a trace structure, analogous to the monotonicity and continuity requirements for process networks.

As an example from our protocol stack, one of the layers may include, among others, two traces, one for a successful operation, and the other for the occurrence of a timeout. The discrete event model is required in this case, as the process network model is unable to handle timeouts.

In some discrete event models, a new event occurs on a signal if and only if the corresponding value for that signal has changed since the previous occurrence. Traces that have this property are called *stutter free*. If this is the case, it is convenient in our framework to define the set of traces as the subset of stutter free traces. We can do this by defining a function that, given a trace, produces its unique stutter free equivalent by removing the unnecessary events, similarly to the stutter removal technique described for non-metric time models (see subsection 4.3.2). Note that discrete traces result in discrete traces after stutter removal.

4.3.6 Pre-Post

One of the fundamental features of embedded software is that it interacts with the physical world. Conventional axiomatic or denotational semantics of sequential programming languages only model initial and final states of terminating programs. Thus, these semantics are inadequate to fully model embedded software.

However, much of the code in an embedded application does computation or internal communication, rather than interacting with the physical world. Such code can be adequately modeled using conventional semantics, as long as the model can be integrated with the more detailed semantics necessary for modeling interactions. The pre-post model is quite similar to conventional semantics, in that we are concerned with modeling non-interactive constructs, such as the ones that occur in a programming language. Thus, in this case, we are interested only in an agents possible final states given an initial state. As described earlier, however, we can also embed the pre-post model into more detailed models. Thus, we can model the non-interactive parts of an embedded application at a high level of abstraction that is simpler and more natural, while also being able to integrate accurate models of interaction, real-time constraints and continuous dynamics.

In our example, this model may be appropriate for the higher levels of the protocol stack, and in particular for the application layer where most of the functionality can be described as non-interactive procedure calls. Note how this model of computation differs from those that were introduced in the previous sections, all of which included some notion of “evolution” of the system. Nonetheless, traces do not necessarily require that notion, and we can easily fit this model in our framework.

Traditionally, the semantics for this kind of models is constructed by first defining a *state* as a set of variables $S = \{ s_i \}$, and then indicating the rules according to which each construct in the programming language modifies this state. A natural semantic domain for describing the constructs is therefore a set of pairs of initial and final state, one for each possible initial state.

The formulation in the framework of trace algebra is almost identical to the natural domain. The signature γ of the agents is simply the set of variables A that the agent depends on and writes to (and is identical to the alphabet algebra of example 2.7). The signature may also be extended to distinguish between different types of variables, as already seen in our previous hybrid model. Each trace is made of pairs of states. A state s is a function with domain A that to each variable $a \in A$ associates a value $s(a)$ from a set of values V. We also define a degenerate, undefined state $\perp$. Given an alphabet A a trace is simply a pair of states

$$\mathcal{B}(A) = (s_i, s_f),$$

where $s_i, s_f : A \to V$ denote the initial and the final state, respectively. Here, the initial state must be non-degenerate. A degenerate final state denotes constructs whose final state is either undefined, or that fail to terminate.

If $s : A \to V$ is a state, we can define projection and renaming on states as follows:

$$\begin{aligned}
&\mathit{proj}(B)(s) = \lambda a \in B \cap A\ [s(a)], \\
&\mathit{rename}(r)(s) = \lambda a \in A\ [s(r(a))].
\end{aligned}$$

Then, if $x = (s_i, s_f)$ is a trace, we define projection and renaming by the obvious extension:

$$\begin{aligned}
\mathit{proj}(B)(x) &= (\mathit{proj}(B)(s_i), \mathit{proj}(B)(s_f)), \\
\mathit{rename}(r)(x) &= (\mathit{rename}(r)(s_i), \mathit{rename}(r)(s_f)).
\end{aligned}$$

A trace structure is easily constructed as a set of traces. As usual, the notion of parallel composition arises automatically given the definition of projection. However, in this particular model, parallel composition is not the main operation of interest, since we are modeling the behavior on non-interacting constructs. In fact, handling shared variables of concurrent programs is problematic with these definitions, and we define parallel composition to be undefined when the signatures of two agents overlap. Instead, we concentrate on the concatenation operation which is relevant to define the concept of sequential composition.

As mentioned in subsection 4.2.2, we must distinguish between complete and partial traces. The above definition of a trace can be interpreted either way, depending on whether we consider the behavior to be completed or not. A non-terminating trace could be considered as a partial trace, assuming that non-termination occurs within a bounded amount of time. This is quite unusual: it may occur, for example, if the duration of an infinite loop decreases exponentially from one iteration to the other.

If $x = (s_i, s_f)$ and $x' = (s'_i, s'_f)$ are traces, the concatenation operation $x'' = x \cdot x'$ is defined if and only if x is a partial trace, the signature A and A' are the same, and the final state of

x is identical to the initial state of x'. As expected, when defined, x'' has alphabet $A'' = A = A'$ and contains the initial state of x and the final state of x':

$$x'' = (s_i, s'_f).$$

Trace structures in this model have signature A which indicates the variables accessible in the scope where the statement appears. For a block that declares local variables, the trace structure for the statement in the block includes in its signature the local variables. The trace structure for the block is formed by projecting away the local variables from the trace structure of the statement.

The trace structures of this model are used to provide a semantics of the statements of a programming language. Clearly, the sequential composition of two statements is defined as the concatenation of the corresponding trace structures: the definition of concatenation ensures that the two statements agree on the intermediate state. In the rest of this section we discuss the semantics of several constructs that are commonly found in programming languages.

For example, the traces in the trace structure for an assignment to variable v are of the form (s_i, s_f), where s_i is an arbitrary initial state, and s_f is identical to s_i except that the value of v is equal to the value of the right-hand side of the assignment statement evaluated in state s_i (we assume the evaluation is side-effect free).

The semantics of a procedure definition is given by a trace structure with an alphabet $\{v_1, \ldots, v_r\}$ where v_k is the k-th argument of the procedure. The semantics of a procedure call `proc(a, b)` is the result of renaming $v_1 \to a$ and $v_2 \to b$ on the trace structure for the definition of `proc`. The parameter passing semantics that results is *value-result* (i.e., no aliasing or references) with the restriction that no parameter can be used for both a value and result. More realistic (and more complicated) parameter passing semantics can also be modeled.

To define the semantics of conditional constructs we introduce a function $\textit{init}(x, c)$ that is true if and only if the predicate c is true in the initial state of trace x. For the semantics of `if-then-else`, let c be the conditional expression and let P_T and P_E be the sets of possible traces of the `then` and `else` clauses, respectively. The set of possible traces of the `if-then-else` is

$$P = \{\, x \in P_T : \textit{init}(x, c)\} \cup \{\, x \in P_E : \neg\textit{init}(x, c)\},$$

that is, we choose the traces from one or the other clause according to the truth value of the condition. Notice that this definition can be used for any trace algebra where $\textit{init}(x, c)$ has been defined, and that it ignores any effects of the evaluation of c not being atomic.

In the case of `while` loops we first define a set of traces E such that for all $x \in E$ and traces y, if $x \cdot y$ is defined then $x \cdot y = y$. For pre-post traces, E is the set of all traces with identical initial and final states. If c is the condition of the loop, and P_B the set of possible traces of the body, we define $P_{T,k}$ and $P_{N,k}$ to be the set of terminating and non-terminating traces, respectively, for iteration k, as follows:

$$\begin{aligned} P_{T,0} &= \{x \in E : \neg \mathit{init}(x, c)\} \\ P_{N,0} &= \{x \in E : \mathit{init}(x, c)\} \\ P_{T,k+1} &= P_{N,k} \cdot P_B \cdot P_{T,0} \\ P_{N,k+1} &= P_{N,k} \cdot P_B \cdot P_{N,0} \end{aligned}$$

The concatenation of $P_{T,0}$ and $P_{N,0}$ at the end of the definition ensures that the final state of a terminating trace does not satisfy the condition c, while that of a non-terminating trace does. Clearly the semantics of the loop should include all the terminating traces. For non-terminating traces, we need to introduce some additional notation. A sequence $Z =< z_0, \ldots >$ is a non-terminating execution sequence of a loop if, for all k, $z_k \in P_{N,k}$ and $z_{k+1} \in z_k \cdot P_B$. This sequence is a chain in the prefix ordering. The initial state of Z is defined to be the initial state of z_0. For pre-post traces, we define $P_{N,\perp}$ to be all traces of the form $(s, \perp)$ where s is the initial state of some non-terminating execution sequence Z of the loop. The set of possible traces of the loop is therefore

$$P = (\bigcup_k P_{T,k}) \cup P_{N,\perp}.$$

4.4 Refinement and Conservative Approximations

In the previous section we have presented the formalization of several models of computation at different levels of abstraction, and how they can all be described in the framework of trace algebra. For each model we have suggested a particular application in the context of a system similar to the PicoRadio project. The whole system is depicted in figure 1.3. In order to understand the behavior and the properties of the whole system, we need to understand the interplay between the different subsystems. We can accomplish this by relating the semantic domains that we have developed in the previous section and by studying how the different notions of computation fit together.

4.4.1 Conservative Approximations Induced by Homomorphisms

As discussed in section 2.6, we can relate different agent algebras, and therefore different models of computation, through a conservative approximation. Burch shows that for trace based agent algebras it is possible to build a conservative approximation starting from a function (or more in general, from a relation) between the trace algebras [12]. To make the conservative approximation compositional, it is required that the function be a homomorphism. In this section we revisit these constructions, which are presented in more general terms. Our presentation also highlights the similarities between this technique and the work of Sifakis [65] and Negulescu [71].

Before introducing conservative approximations, we must define a refinement order for trace structures. This definition will be generalized in section 5.1 (definition 5.1).

Definition 4.15 (Refinement order). Let $p = (A, P)$ and $p' = (A', P')$ be two trace structures. Then

$$p \preceq p' \;\Leftrightarrow\; A = A' \wedge P \subseteq P'.$$

In other words, an agent p refines an agent p' if the possible behaviors of p are also possible behaviors of p'. In this case, we also say that the implementation p satisfies the specification p'. This definition is therefore similar to traditional notions of refinement using language containment. It is easy to show that the operators of projection, renaming and parallel composition are $\top$-monotonic relative to this order. Therefore, a trace structure algebra is a partially ordered agent algebra.

Conservative approximations can be derived from a function that relates the underlying trace algebras. In order to obtain compositional conservative approximations, the function must *commute* with the operators of the trace algebra. Such a function is called a *homomorphism*. The following definition is a specialization of the notion of homomorphism to trace algebras.

Definition 4.16 (Homomorphism). Let $\mathcal{C}$ and $\mathcal{C}'$ be trace algebras. Let $h : \mathcal{B} \mapsto \mathcal{B}'$ be a function such that for all alphabets A, $h(\mathcal{B}(A)) \subseteq \mathcal{B}'(A)$. Then h is a *homomorphism from $\mathcal{C}$ to $\mathcal{C}'$* if and only if

$$\begin{aligned} h(\mathit{rename}(r)(x)) &= \mathit{rename}(r)(h(x)), \\ h(\mathit{proj}(B)(x)) &= \mathit{proj}(B)(h(x)), \end{aligned}$$

where the right hand side of the equation is defined if the left hand side is defined.

A conservative approximation on the trace structures is essentially a pair of functions that operate on sets of traces. Given a function on traces, there are different ways one could derive a function on sets of traces. Here we use the notion of an axiality, that is of the one-to-one correspondence between relations between two sets and Galois connections between their powersets. Lemma 4.18 below, which makes this correspondence precise, is a well known results in the theory of Galois connections.

Definition 4.17 (Axiality). Let $\mathcal{B}$ and $\mathcal{B}'$ be sets and let $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ be a binary relation between $\mathcal{B}$ and $\mathcal{B}'$. The *axialities* of ρ are the two functions $\rho_\alpha : 2^{\mathcal{B}} \mapsto 2^{\mathcal{B}'}$ and $\rho_\gamma : 2^{\mathcal{B}'} \mapsto 2^{\mathcal{B}}$ defined as

$$\begin{aligned}
\rho_\alpha(X) &= \{\, y \in \mathcal{B}' : \exists x \in \mathcal{B}\, [x \in X \wedge (x,y) \in \rho]\},\\
\rho_\gamma(Y) &= \{\, x \in \mathcal{B} : \forall y \in \mathcal{B}'\, [(x,y) \in \rho \Rightarrow y \in Y]\}.
\end{aligned}$$

Lemma 4.18. Let $\mathcal{B}$ and $\mathcal{B}'$ be sets. Then $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ is a binary relation between $\mathcal{B}$ and $\mathcal{B}'$ if and only if $\langle \rho_\alpha, \rho_\gamma \rangle$ is a Galois connection between $2^{\mathcal{B}}$ and $2^{\mathcal{B}'}$.

Proof: For the forward direction, assume $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ is a binary relation between $\mathcal{B}$ and $\mathcal{B}'$. Let $X \subseteq \mathcal{B}$ and $Y \subseteq \mathcal{B}'$. We prove that $\rho_\alpha(X) \subseteq Y$ if and only if $X \subseteq \rho_\gamma(Y)$. The proof consists of the following series of double implications.

$$\begin{aligned}
&\rho_\alpha(X) \subseteq Y\\
&\quad\text{by definition 4.17,}\\
&\Leftrightarrow\quad \{\, y \in \mathcal{B}' : \exists x \in \mathcal{B}\, [x \in X \wedge (x,y) \in \rho]\} \subseteq Y\\
&\Leftrightarrow\quad \{\, y \in \mathcal{B}' : \exists x \in \mathcal{B}\, [x \in X \wedge (x,y) \in \rho]\} \cap \overline{Y} = \emptyset\\
&\Leftrightarrow\quad \forall y \in \mathcal{B}'\, \forall x \in \mathcal{B}\, \neg[x \in X \wedge (x,y) \in \rho \wedge y \notin Y]\\
&\Leftrightarrow\quad \forall x \in \mathcal{B}\, \forall y \in \mathcal{B}'\, \neg[x \in X \wedge (x,y) \in \rho \wedge y \notin Y]\\
&\Leftrightarrow\quad \{\, x \in \mathcal{B} : \exists y \in \mathcal{B}'\, [x \in X \wedge (x,y) \in \rho \wedge y \notin Y]\} = \emptyset\\
&\Leftrightarrow\quad \{\, x \in \mathcal{B} : x \in X\} \cap \overline{\{\, x \in \mathcal{B} : \forall y \in \mathcal{B}'\, \neg[(x,y) \in \rho \wedge y \notin Y]\}} = \emptyset\\
&\Leftrightarrow\quad X \subseteq \{\, x \in \mathcal{B} : \forall y \in \mathcal{B}'\, [(x,y) \in \rho \Rightarrow y \in Y]\}\\
&\quad\text{by definition 4.17,}\\
&\Leftrightarrow\quad X \subseteq \rho_\gamma(Y).
\end{aligned}$$

For the reverse direction, assume that $\langle \alpha, \gamma \rangle$ is a Galois connection and define, for all $x \in \mathcal{B}$ and $y \in \mathcal{B}'$,

$$(x,y) \in \rho \;\Leftrightarrow\; y \in \alpha(\{\, x\}). \tag{4.2}$$

We show that for all $X \subseteq \mathcal{B}$, $\rho_\alpha(X) = \alpha(X)$. We proceed by induction on the size of X.

For the base case, by corollary 2.77, $\alpha(\emptyset) = \emptyset$, and clearly, by equation 4.2, $\rho_\alpha(\emptyset) = \emptyset$. Therefore, $\rho_\alpha(\emptyset) = \alpha(\emptyset)$. Assume now $\rho_\alpha(X) = \alpha(X)$ and let $X' = X \cup \{ x' \}$. We show that $\rho_\alpha(X') = \alpha(X')$. The proof consists of the following series of implications, starting from definition 4.17.

$$
\begin{aligned}
\rho_\alpha(X') &= \{ y \in \mathcal{B}' : \exists x \in \mathcal{B}\, [x \in X' \wedge (x, y) \in \rho] \} \\
&\quad \text{by the definition of } \rho \text{ (equation 4.2),} \\
&= \{ y \in \mathcal{B}' : \exists x \in \mathcal{B}\, [x \in X' \wedge y \in \alpha(\{ x \})] \} \\
&= \{ y \in \mathcal{B}' : \exists x \in \mathcal{B}\, [x \in X \wedge y \in \alpha(\{ x \})] \} \cup \{ y \in \mathcal{B}' : y \in \alpha(\{ x' \}) \} \\
&\quad \text{by definition 4.17,} \\
&= \rho_\alpha(X) \cup \alpha(\{ x' \}) \\
&\quad \text{by hypothesis} \\
&= \alpha(X) \cup \alpha(\{ x' \}) \\
&\quad \text{since, by theorem 2.90, } \alpha \text{ distributes over } \cup, \\
&= \alpha(X \cup \{ x' \}) = \alpha(X').
\end{aligned}
$$

□

The following result is obtained by applying lemma 4.18 to the inverse relation ρ^{-1}.

Lemma 4.19. Let $\mathcal{B}$ and $\mathcal{B}'$ be sets. Then $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ is a binary relation between $\mathcal{B}$ and $\mathcal{B}'$ if and only if $\langle (\rho^{-1})_\alpha, (\rho^{-1})_\gamma \rangle$ is a Galois connection between $2^{\mathcal{B}'}$ and $2^{\mathcal{B}}$.

We are interested in relations on traces that preserve the alphabet. In other words, we are interested in relations ρ between $\mathcal{B}$ and $\mathcal{B}'$ such that if $(x, y) \in \rho$, then there exists an alphabet A such that $x \in \mathcal{B}(A)$ and $y \in \mathcal{B}'(A)$, (i.e., ρ relates only traces with the same alphabet). Then, for each alphabet A, the relation $\rho_A \subseteq \mathcal{B}(A) \times \mathcal{B}'(A)$, obtained by restricting ρ to the traces with alphabet A, determines two pairs of axialities between $2^{\mathcal{B}(A)}$ and $2^{\mathcal{B}'(A)}$, which form Galois connections. Since in the following the alphabet will always be clear from context, we will drop the subscript A from ρ_A, and refer to its axialities simply as ρ_α and ρ_γ.

Let now $\mathcal{C}$ and $\mathcal{C}'$ be trace algebras and let $\mathcal{A}$ and $\mathcal{A}'$ be trace structure algebras over $\mathcal{C}$ and $\mathcal{C}'$, respectively. If ρ preserves the alphabets, then its axialities can be extended to functions between the trace structures in $\mathcal{A}$ and $\mathcal{A}'$. With a little abuse of notation we will write for a trace structure $p = (A, P)$:

$$\rho_\alpha(p) = (A, \rho_\alpha(P)).$$

Since they form a pair of Galois connections, it is natural to ask whether they also form a conservative approximation. In the following we will consider only trace structures algebras whose domain consists of *all* the trace structures over the corresponding trace algebra. We call such trace structure algebras *complete*.

Corollary 4.20. Let $\mathcal{C} = (\mathcal{B}, \mathit{proj}, \mathit{rename})$ and $\mathcal{C}' = (\mathcal{B}', \mathit{proj}', \mathit{rename}')$ be trace algebras and let $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ be a binary relation between $\mathcal{B}$ and $\mathcal{B}'$ that preserves the alphabets. Let $\mathcal{A}$ and $\mathcal{A}'$ be the complete trace structure algebras over $\mathcal{C}$ and $\mathcal{C}'$. Then the following two statements are equivalent.

- $\langle \rho_\alpha, (\rho^{-1})_\gamma \rangle$ is a conservative approximation from $\mathcal{A}$ to $\mathcal{A}'$.
- For all trace structures p' in $\mathcal{A}'.D$, $\rho_\gamma(p') \preceq (\rho^{-1})_\alpha(p')$.

Proof: The result follows directly from corollary 2.101. □

The following two results give sufficient conditions to prove that the axialities of a relation on traces form a conservative approximation. The first shows that if ρ is total, the the corresponding Galois connections indeed form a conservative approximation. The second shows that if ρ is also a function, then the induced conservative approximation is also the tightest.

Lemma 4.21. Let $\mathcal{B}$ and $\mathcal{B}'$ be sets and let $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ be a binary relation between $\mathcal{B}$ and $\mathcal{B}'$. If ρ is total, then for all $Y \subseteq \mathcal{B}'$, $\rho_\gamma(Y) \subseteq (\rho^{-1})_\alpha(Y)$.

Proof: Let $Y \subseteq \mathcal{B}'$. Let $x \in \rho_\gamma(Y)$, and let $y' \in \mathcal{B}'$ be such that $(x, y') \in \rho$. We know that y' exists, since ρ is total. We must show that $x \in (\rho^{-1})_\alpha(Y)$. The proof consists of the following series of implications.

$x \in \rho_\gamma(Y)$

by definition 4.17,

$\Leftrightarrow \quad \forall y \in \mathcal{B}' \; [(x, y) \in \rho \Rightarrow y \in Y]$

since $(x, y') \in \rho$,

$\Rightarrow \quad y' \in Y$

since $(x, y') \in \rho$,

$\Rightarrow \quad \exists y \in \mathcal{B}' \; [y \in Y \wedge (x, y) \in \rho]$

$\Rightarrow \quad x \in \{\, x \in \mathcal{B} : \exists y \in \mathcal{B}' \; [y \in Y \wedge (x, y) \in \rho]\}$

$\Leftrightarrow \quad x \in \{\, x \in \mathcal{B} : \exists y \in \mathcal{B}' \; [y \in Y \wedge (y, x) \in \rho^{-1}]\}$

by definition 4.17,

$\Leftrightarrow \quad x \in (\rho^{-1})_\alpha(Y).$

□

Lemma 4.22. Let $\mathcal{B}$ and $\mathcal{B}'$ be sets and let $\rho : \mathcal{B} \mapsto \mathcal{B}'$ be a function from $\mathcal{B}$ and $\mathcal{B}'$. Then for all $Y \subseteq \mathcal{B}'$, $\rho_\gamma(Y) = (\rho^{-1})_\alpha(Y)$.

Proof: Let $Y \subseteq \mathcal{B}'$. Since ρ is total, by lemma 4.21, we only need to prove that $(\rho^{-1})_\alpha(Y) \subseteq \rho_\gamma(Y)$. Let $x \in (\rho^{-1})_\alpha(Y)$ and let $y' = \rho(x)$. Since ρ is a function, $(x, y) \in \rho$ if and only if $y = y'$.

$x \in (\rho^{-1})_\alpha(Y)$

by definition 4.17,

$\Leftrightarrow \quad x \in \{\, x \in \mathcal{B} : \exists y \in \mathcal{B}' \; [y \in Y \wedge (y, x) \in \rho^{-1}]\}$

$\Leftrightarrow \quad x \in \{\, x \in \mathcal{B} : \exists y \in \mathcal{B}' \; [y \in Y \wedge (x, y) \in \rho]\}$

since $(x, y') \in \rho$ and since $(x, y) \in \rho \Rightarrow y = y'$,

$\Rightarrow \quad y' \in Y$

since $(x, y') \in \rho$ and since $(x, y) \in \rho \Rightarrow y = y'$,

$\Rightarrow \quad \forall y \in \mathcal{B}' \; [(x, y) \in \rho \Rightarrow y \in Y]$

$\Rightarrow \quad x \in \{\, x \in \mathcal{B} : \forall y \in \mathcal{B}' \; [(x, y) \in \rho \Rightarrow y \in Y]\}$

by definition 4.17,

$\Leftrightarrow \quad x \in \rho_\gamma(Y).$

□

The above result shows that, for complete trace structure algebras, a function between the sets of traces induces the tightest conservative approximation (see corollary 2.105). Before we derive the close form expression for the conservative approximation induced by a function (and by a homomorphism of trace algebras in particular), we briefly review the notation proposed by

Loiseaux et al. [65] based on the forward and backward image of a relation. This notation is simpler to use to derive the closed form expression of the conservative approximation. Here we show that the forward and backward images are equivalent to the axialities of the relation.

Definition 4.23 (Forward and Backward Image Function). Let $\mathcal{B}$ and $\mathcal{B}'$ be sets and let $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ be a binary relation between $\mathcal{B}$ and $\mathcal{B}'$. The forward image $\mathit{post}[\rho] : 2^{\mathcal{B}} \mapsto 2^{\mathcal{B}'}$, and the backward image $\mathit{pre}[\rho] : 2^{\mathcal{B}'} \mapsto 2^{\mathcal{B}}$ of ρ are defined as

$$\begin{aligned} \mathit{post}[\rho](X) &= \{ y \in \mathcal{B}' : \exists x \in X \; [(x,y) \in \rho]\}, \\ \mathit{pre}[\rho](Y) &= \{ x \in \mathcal{B} : \exists y \in Y \; [(x,y) \in \rho]\}. \end{aligned}$$

Definition 4.24 (Dual of Function). Let $\mathcal{B}$ and $\mathcal{B}'$ be sets and let $f : 2^{\mathcal{B}} \mapsto 2^{\mathcal{B}'}$ be a function. The *dual* of f is the function $\widetilde{f} : 2^{\mathcal{B}} \mapsto 2^{\mathcal{B}'}$ such that

$$\widetilde{f}(X) = \overline{f(\overline{X})},$$

where $\overline{X}$ denotes the complement of the set X.

Theorem 4.25. Let $\mathcal{B}$ and $\mathcal{B}'$ be sets and let $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ be a binary relation between $\mathcal{B}$ and $\mathcal{B}'$. Then,

$$\begin{aligned} \rho_\alpha &= \mathit{post}[\rho], \\ \rho_\gamma &= \widetilde{\mathit{pre}[\rho]}. \end{aligned}$$

Proof: Clearly, by inspection of definition 4.17 and definition 4.23, $\rho_\alpha = \mathit{post}[\rho]$. The proof is completed by the following series of equalities, which begins with the definition of dual of a function (def. 4.24):

$$\begin{aligned} \widetilde{\mathit{pre}[\rho]}(Y) &= \overline{\mathit{pre}[\rho](\overline{Y})} \\ &\quad \text{by definition 4.23,} \\ &= \overline{\{ x \in \mathcal{B} : \exists y \in \overline{Y} \; [(x,y) \in \rho]\}} \\ &= \{ x \in \mathcal{B} : \neg\exists y \in \overline{Y} \; [(x,y) \in \rho]\} \\ &= \{ x \in \mathcal{B} : \forall y \in \overline{Y} \; [(x,y) \notin \rho]\} \\ &= \{ x \in \mathcal{B} : \forall y \in \mathcal{B}' \; [(x,y) \in \rho \Rightarrow y \in Y]\} \\ &\quad \text{by definition 4.17,} \\ &= \rho_\gamma(Y). \end{aligned}$$

□

Corollary 4.26. Let $\mathcal{B}$ and $\mathcal{B}'$ be sets and let $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ be a binary relation between $\mathcal{B}$ and $\mathcal{B}'$. Then,

$$\begin{aligned} (\rho^{-1})_\alpha &= \mathit{post}[\rho^{-1}], \\ (\rho^{-1})_\gamma &= \widetilde{\mathit{pre}[\rho^{-1}]}. \end{aligned}$$

The forward image of a relation is related to the backward image of the inverse relation as follow.

Lemma 4.27. Let $\mathcal{B}$ and $\mathcal{B}'$ be sets and let $\rho \subseteq \mathcal{B} \times \mathcal{B}'$ be a binary relation between $\mathcal{B}$ and $\mathcal{B}'$. Then,

$$\begin{aligned} \mathit{post}[\rho^{-1}] &= \mathit{pre}[\rho], \\ \widetilde{\mathit{pre}[\rho^{-1}]} &= \widetilde{\mathit{post}[\rho]}. \end{aligned}$$

Hence we can express all axialities in terms of images of ρ. In particular, it is easy to find a closed form expression for the conservative approximation induced by a homomorphism on traces, as defined in corollary 4.20.

Theorem 4.28. Let $\mathcal{C} = (\mathcal{B}, \mathit{proj}, \mathit{rename})$ and $\mathcal{C}' = (\mathcal{B}', \mathit{proj}', \mathit{rename}')$ be trace algebras and let h be a homomorphism from $\mathcal{C}$ to $\mathcal{C}'$. Let $\mathcal{A}$ and $\mathcal{A}'$ be the complete trace structure algebras over $\mathcal{C}$ and $\mathcal{C}'$, respectively. Then the pair of functions $\Psi = (\Psi_l, \Psi_u)$ defined by

$$\begin{aligned} \Psi_u(p) &= (A, h(P)), \\ \Psi_l(p) &= (A, \mathcal{B}'(A) - h(\mathcal{B}(A) - P)) \end{aligned}$$

is a conservative approximation from $\mathcal{A}$ to $\mathcal{A}'$.

Proof: Recall that a homomorphism on trace algebras preserves the alphabets. Let (A, P) be a trace structure in $\mathcal{A}$. Then,

$$\begin{aligned} \rho_\alpha(p) &= (A, \mathit{post}[h](P)) \\ &= (A, h(P)), \\ (\rho^{-1})_\gamma(p) &= (A, \widetilde{\mathit{post}[h]}(P)) \\ &= (A, \overline{\mathit{post}[h](\overline{P})}) \\ &= (A, \overline{h(\mathcal{B}(A) - P)}) \\ &= (A, \mathcal{B}'(A) - h(\mathcal{B}(A) - P)). \end{aligned}$$

The result then follows from corollary 4.20 and lemma 4.22. □

It is easy to show that because h is a homomorphism, then the conservative approximation induced by h is also compositional. Burch [12] actually derives a different formulation for the lower bound. Specifically, he has

$$\Psi'_l(p) = (A, h(P) - h(\mathcal{B}(A) - P)).$$

The two formulas are equivalent if h is surjective.

Lemma 4.29. Let $\mathcal{C}$ and $\mathcal{C}'$ be trace algebras and let h be a surjective homomorphism from $\mathcal{C}$ to $\mathcal{C}'$. Then for all P,

$$\mathcal{B}'(A) - h(\mathcal{B}(A) - P) = h(P) - h(\mathcal{B}(A) - P).$$

Proof: Let $y \in \mathcal{B}'(A)$ and let $x \in \mathcal{B}(A)$ be such that $y = h(x)$. We know that x exists since h is surjective. It is enough to show that if $y \in \mathcal{B}'(A) - h(\mathcal{B}(A) - P)$ then $y \in h(P)$.

$$\begin{aligned} & y \in \mathcal{B}'(A) - h(\mathcal{B}(A) - P) \\ & \quad \Rightarrow \quad y \notin h(\mathcal{B}(A) - P) \\ & \quad \Rightarrow \quad x \in P \\ & \quad \Rightarrow \quad y \in h(P). \end{aligned}$$

□

Even when h is not surjective, the two formulations give conservative approximations that have the same "distinguishing power" in terms of their ability to reflect verification results from the abstract to the concrete model.

Lemma 4.30. Let $\mathcal{C}$ and $\mathcal{C}'$ be trace algebras and let h be a homomorphism from $\mathcal{C}$ to $\mathcal{C}'$. Let $p_1 = (A, P_1)$ and $p = (A, P)$ be trace structures over $\mathcal{C}$ and $\mathcal{C}'$, respectively. Then

$$\Psi_u(p_1) \quad \preceq \quad \Psi_l(p)$$

if and only if

$$\Psi_u(p_1) \quad \preceq \quad \Psi'_l(p).$$

Proof: We will show that

$$h(P_1) \quad \subseteq \quad \mathcal{B}'(A) - h(\mathcal{B}(A) - P)$$

if and only if

$$h(P_1) \quad \subseteq \quad h(P) - h(\mathcal{B}(A) - P).$$

The backward implication is obvious. For the forward implication, assume that $h(P_1) \subseteq \mathcal{B}'(A) - h(\mathcal{B}(A) - P)$. We show that $P_1 \subseteq P$, which implies $h(P_1) \subseteq h(P)$. This, in turn, implies the desired result. The proof consists of the following series of implications.

$$\begin{aligned}
& x \in P_1 \\
& \quad \text{since } h(P_1) \subseteq \mathcal{B}'(A) - h(\mathcal{B}(A) - P), \\
& \quad \Rightarrow \quad h(x) \in \mathcal{B}'(A) - h(\mathcal{B}(A) - P) \\
& \quad \Rightarrow \quad h(x) \notin h(\mathcal{B}(A) - P) \\
& \quad \Rightarrow \quad x \notin \mathcal{B}(A) - P \\
& \quad \Rightarrow \quad x \in P.
\end{aligned}$$

Hence $P_1 \subseteq P$. □

Although the conservative approximations using the two different lower bounds are essentially equivalent, they are not equal. In particular, the inverses of the conservative approximation may be different. This is apparent if $\mathcal{C}'$ contains traces that are not in the image of the homomorphism. In this case, the inverse of $\Psi = (\Psi_l, \Psi_u)$ is never defined. To see why that is the case, assume $x' \in \mathcal{B}'(A)$ is not in the image of the homomorphism. Then, clearly, for all $P \subseteq \mathcal{B}(A)$, $x' \notin h(P)$, and $x' \in \mathcal{B}'(A) - h(\mathcal{B}(A) - P)$. Hence $\Psi_l(p)$ and $\Psi_u(p)$ are never equal. On the other hand, if $P = h^{-1}(h(P))$ (i.e., the set P includes all the traces that are mapped onto the set $h(P)$), then $h(\mathcal{B}(A) - P) \cap h(P) = \emptyset$. Then necessarily $h(P) = h(P) - h(\mathcal{B}(A) - P)$, which implies $\Psi'_l(p) = \Psi_u(p)$ and $\Psi_{inv}((A, h(P)) = (A, P)$. In other words, by ignoring the existence of additional traces, Ψ'_l is effectively determining that certain agents can be represented "exactly" at the abstract level.

In the case above, Ψ'_l is not even the concretization function of any Galois connection from $\mathcal{A}'$ to $\mathcal{A}$. Assume, in fact, that $p' = (A, P')$ is a trace structure in $\mathcal{A}'$ such that P' contains a trace x' that is not in the image of the homomorphism. By theorem 2.93, the abstraction map on p' is the least element of the set

$$\Delta_{\Psi'_l}(P') = \{\, p = (A, P) : P' \subseteq h(P) - h(\mathcal{B}(A) - P)\}.$$

But, clearly, there is no P such that $P' \subseteq h(P) - h(\mathcal{B}(A) - P)$, since $x' \in P'$ and for all P, $x' \notin h(P)$ since x' is not in the image of h. Hence, since $\Delta_{\Psi'_l}(P')$ is empty, it has no least element, and therefore, by theorem 2.97, Ψ'_l is not the concretization map of any Galois connection. Consequently, Ψ'_l is also not induced by any relation (or function) on the sets of traces in the form of an axiality.

An upper bound that includes the information carried by extra traces not in the image of the homomorphism also exists. Consider for example the following abstraction:

$$\Psi'_u(p) = (\mathcal{B}'(A) - h(\mathcal{B}(A))) \cup h(P).$$

The following two results again show that the conservative approximation (Ψ_l, Ψ'_u) has the same distinguishing power as (Ψ_l, Ψ_u), and that in case h is surjective, they are the same.

Lemma 4.31. Let $\mathcal{C}$ and $\mathcal{C}'$ be trace algebras and let h be a homomorphism from $\mathcal{C}$ to $\mathcal{C}'$. Let $p_1 = (A, P_1)$ and $p = (A, P)$ be trace structures over $\mathcal{C}$ and $\mathcal{C}'$, respectively. Then

$$\Psi_u(p_1) \quad \preceq \quad \Psi_l(p)$$

if and only if

$$\Psi'_u(p_1) \quad \preceq \quad \Psi_l(p).$$

Proof: The backward implication is again obvious. For the forward direction, assume

$$h(P_1) \subseteq \mathcal{B}'(A) - h(\mathcal{B}(A) - P).$$

We wish to show that

$$(\mathcal{B}'(A) - h(\mathcal{B}(A))) \cup h(P_1) \subseteq \mathcal{B}'(A) - h(\mathcal{B}(A) - P).$$

To do so, it is sufficient to show that $(\mathcal{B}'(A) - h(\mathcal{B}(A))) \subseteq \mathcal{B}'(A) - h(\mathcal{B}(A) - P)$, since we already know, by hypothesis, that $h(P_1) \subseteq \mathcal{B}'(A) - h(\mathcal{B}(A) - P)$. Let then $x \in \mathcal{B}'(A)$ be a trace in $\mathcal{C}'$ such that $x \in \mathcal{B}'(A) - h(\mathcal{B}(A))$. Then, necessarily, $x \notin h(\mathcal{B}(A))$. Therefore, since $h(\mathcal{B}(A) - P) \subseteq h(\mathcal{B}(A))$, $x \notin h(\mathcal{B}(A) - P)$. Hence, since $x \in \mathcal{B}'(A)$, $x \in \mathcal{B}'(A) - h(\mathcal{B}(A) - P)$. Consequently, $(\mathcal{B}'(A) - h(\mathcal{B}(A))) \subseteq \mathcal{B}'(A) - h(\mathcal{B}(A) - P)$. $\square$

Lemma 4.32. Let $\mathcal{C}$ and $\mathcal{C}'$ be trace algebras and let h be a surjective homomorphism from $\mathcal{C}$ to $\mathcal{C}'$. Then for all P,

$$(\mathcal{B}'(A) - h(\mathcal{B}(A))) \cup h(P) = h(P).$$

Proof: The result follows easily from the fact that if h is surjective, then $h(\mathcal{B}(A)) = \mathcal{B}'(A)$ and therefore $\mathcal{B}'(A) - h(\mathcal{B}(A)) = \emptyset$. □

In summary, given a homomorphism between trace structures, it is possible to derive conservative approximations between the corresponding complete trace structure algebras. The notion of axiality provides us with one formulation of the conservative approximation. We have also shown how to derive conservative approximations that have the same distinguishing power, but that differ in the existence of an inverse function. If the homomorphism is not surjective, then it is impossible to derive such approximations as axialities of a relation on traces, evidence of the fact that conservative approximations are more general than Galois connections.

In the rest of this chapter we will explore conservative approximations between some of the models that were presented in the previous section. We will do so by establishing homomorphisms on trace structures, and by choosing the formulation of conservative approximation given by (Ψ'_l, Ψ_u) above. We do so for several reason. First, our homomorphisms are, in general, surjective. Hence, for all purposes, and for refinement verification in particular, the choice of conservative approximation is irrelevant. In addition, (Ψ'_l, Ψ_u) clearly highlights that the inverse is defined for all trace structures $p = (A, P)$ such that $P = h^{-1}(h(P))$, that is for all the agents whose set of traces contains all the concretizations of their abstractions. This is convenient when considering the embedding of one agent model into another, as explained in section 2.8. Further the formulation also highlights that for all trace structures, $\Psi'_l(p) \preceq \Psi_u(p)$, a necessary and sufficient condition for the existence of a conservative approximation going in the opposite direction (since, because h is surjective, we *are* using the axiality).

If the trace structure algebras considered are not complete, as it is the case in some of the examples that we present, the upper and the lower bound computed according to our formulation may not correspond to any trace structure in the more abstract model. In this case, the conservative approximation must be altered. Specifically, a new conservative approximation (Ψ''_l, Ψ''_u) must be found such that for all trace structures p,

$$\begin{aligned} \Psi''_l(p) &\preceq \Psi'_l(p) \\ \Psi_u(p) &\preceq \Psi''_u(p) \end{aligned}$$

If that is the case, then, by theorem 2.61, (Ψ''_l, Ψ''_u) is also a conservative approximation. There might be a choice of the specific looser bound to be used. We do not consider this problem here, and reserve it for our future work.

4.5 Examples of Conservative Approximations

This section is devoted to extending the examples presented in section 4.3 by relating the different models with conservative approximations. We will address two cases in particular. The first is devoted to the study of an embedded software control application and to the construction of relationships between the continuous time, the non-metric time and the pre-post model. This is achieved by first constructing appropriate homomorphisms of trace algebra, and by applying the results of the previous section to obtain a conservative approximation.

The second set of examples includes some of the models used by our motivating example in section 1.6. Specifically, we will analyze how to construct conservative approximations from the continuous time model of computation to the discrete event model, and from the discrete event model to the process networks model. For this last example we will also consider the form of the inverse of the conservative approximation.

4.5.1 Cutoff Control

Our example is a small segment of code used for engine cutoff control [8]. This example is particularly interesting to us because the solution proposed in [8] includes the use of a hybrid model to describe the torque generation mechanism.

The behaviors of an automobile engine are divided into regions of operation, each characterized by appropriate control actions to achieve a desired result. The cutoff region is entered when the driver releases the accelerator pedal, thereby requesting that no torque be generated by the engine. In order to minimize power train oscillations that result from suddenly reducing torque, a closed loop control damps the oscillations using carefully timed injections of fuel. The control problem is therefore hybrid, consisting of a discrete (the fuel injection) and a continuous (the power train behavior) systems tightly linked. The approach taken in [8] is to first relax the problem to the continuous domain, solve the problem at this level, and finally abstract the solution to the discrete domain.

Figure 4.7 shows the top level routine of the control algorithm. Although we use a C-like syntax, the semantics are simplified, as described before for the pre-post model. The controller is activated by a request for an injection decision (this happens every full engine cycle). The algorithm first reads the current state of the system (as provided by the sensors on the power train), predicts the effect of injecting or not injecting on the future behavior of the system, and finally controls whether injection occurs. The prediction uses the value of the past three decisions to estimate the position of the future state. The control algorithm involves solving a differential equation, which is done in the call to `compute_sigmas` (see [8] for more details). A nearly optimal solution can be achieved without injecting intermediate amounts of fuel (i.e., either inject no fuel or inject the maximum amount). Thus, the only control inputs to the system are the actions `action_injection` (maximum injection) and `action_no_injection` (zero injection).

Even this small fragment of code highlights the different nature of several of the constructs. For example, the function call to `compute_sigmas`, and the corresponding implementation, need not be described in a model that uses a notion of time. Thus, the pre-post model is sufficient. On the other hand, the `await` statement depends upon the arrival of an event, and is therefore best represent in a timed model, whether it has a metric or not. Conversely, performance constraints, such as the maximum delay for an iteration of the loop, requires not only a notion of time, but also a metric. The homomorphisms below and the corresponding conservative approximations can then be used to translate one representation into the other, according to the type of analysis that must be performed on the design.

4.5.2 Homomorphisms

The trace algebras defined above cover a wide range of levels of abstraction. The first step in formalizing the relationships between those levels is to define homomorphisms between the trace algebras. As mentioned in section 4.4, trace algebra homomorphisms induce corresponding conservative approximations between trace structure algebras.

4.5.3 From Metric to Non-metric Time

A homomorphism from metric time trace algebra to non-metric time should abstract away detailed timing information. It is easy to define a homomorphism by simply interpreting the non-negative reals as the vertex set, and the assignments on the non-negative reals as the labeling function. The result is an lpo to which we apply the stutter removal procedure defined in subsection 4.3.2, and then take the isomorphism class.

Nevertheless, to better highlight what the homomorphism does, we here proceed in a more direct way. This requires characterizing events in metric time and mapping those events into a non-metric time domain. Since metric time trace algebra is, in part, value based, some additional definitions are required to characterize events at that level of abstraction.

```
void control_algorithm ( void ) {
    // State definition
    struct state {
        double $x_1$;
        double $x_2$;
        double $\omega_c$;
    } current_state;
    // Init the past three injections (assume injection before cutoff)
    double $u_1$, $u_2$, $u_3 = 1.0$;
    // Predictions
    double $\sigma_m$, $\sigma_0$;

    loop forever {
        await ( action_request );
        read_current_state ( current_state );
        compute_sigmas ( $\sigma_m$, $\sigma_0$, current_state, $u_1$, $u_2$, $u_3$ );
        // Update past injections
        $u_1 = u_2$;
        $u_2 = u_3$;
        // Compute next injection signal
        if ( $\sigma_m < \sigma_0$ ) {
            action_injection ( );
            $u_3 = 1.0$;
        } else {
            action_no_injection ( );
            $u_3 = 0.0$;
        }
    }
}
```

Figure 4.7: The control algorithm

Let x be a metric trace with signature γ and alphabet A such that

$$\begin{aligned}\gamma &= (V_{\mathbb{R}}, V_{\mathbb{Z}}, M_I, M_O)\\ A &= V_{\mathbb{R}} \cup V_{\mathbb{Z}} \cup M_I \cup M_O.\end{aligned}$$

We define the homomorphism h by defining a non-metric time trace $y = h(x)$. This requires building a vertex set V and a labeling function μ to construct an lpo. The trace y is the isomorphism class of this lpo. For the vertex set we take all reals such that an event occurs in the trace x, where the notion of event is formalized in the next several definitions.

Definition 4.33 (Stable function). Let f be a function over a real interval to $\mathbb{R}$ or $\mathbb{Z}$. The function is *stable at* t if and only if there exists an $\epsilon > 0$ such that f is constant on the interval $(t-\epsilon, t]$.

Definition 4.34 (Stable trace). A metric time trace x is *stable at* t if and only if for all $v \in V_{\mathbb{R}} \cup V_{\mathbb{Z}}$ the function $f(v)$ is stable at t; and for all $a \in M_I \cup M_O$, $f(a)(t) = 0$.

Definition 4.35 (Event). A metric time trace x has an *event at* $t > 0$ if it is not stable at t. Because a metric time trace doesn't have a left neighborhood at $t = 0$, we always assume the presence of an event at the beginning of the trace. If x has an event at t, the *action label* σ for that event is a function with domain A such that for all $v \in A$, $\sigma(a) = f(a)(t)$, where f is a component of x as described in the definition of metric time traces.

Now we construct the vertex set V and labeling function μ necessary to define y and, thereby, the homomorphism h. The vertex set V is the set of reals t such that x has an event at t. While it is convenient to make V a subset of the reals, remember that the tomset that results is an isomorphism class. Hence the metric defined on the set of reals is lost. The labeling function μ is such that for each element $t \in V$, $\mu(t)$ is the action label for the event at t in x.

Note that if we start from a partial trace in the metric trace we obtain a trace in the non-metric trace that has an initial and final event. It has an initial event by definition. It has a final event because the metric trace either has an event at δ (the function is not constant), or the function is constant at δ but then there must be an event that brought the function to that constant value (which, in case of identically constant functions, is the initial event itself).

To show that h does indeed abstract away information, consider the following situation. Let x_1 be a metric time trace. Let x_2 be same trace where time has been "stretched" by a factor of two (i.e., for all $v \in A_1$, $x_1(a)(t) = x_2(a)(2t)$). The vertex sets generated by the above process are isomorphic (the order of the events is preserved), therefore $h(x_1) = h(x_2)$.

4.5.4 From Non-metric to Pre-post Time

The homomorphism h from the non-metric time traces to pre-post traces requires that the signature of the trace structure be changed by removing M_I and M_O. Let $y = h(x)$. The initial state of y is formed by restricting $\mu(\min(V))$ (the initial state of x) to $V_{\mathbb{R}} \cup V_{\mathbb{Z}}$. If x is a complete trace, then the final state of y is $\perp$. If x is a complete trace, and there exists $a \in M_I \cup M_O$ and time t such that $f(a)(t) = 1$, the final state of y is also $\perp$. Otherwise, the final state of y is formed by restricting $\mu(\max(V))$.

4.5.4.1 Using Non-Metric Time Traces

Using an inverse conservative approximation, as described earlier, the pre-post trace semantics described in the previous subsection can be embedded into non-metric time trace structures. However, this is not adequate for two of the constructs used in figure 4.7: `await` and the non-terminating loop. These constructs must be describe directly at the lower level of abstraction provided by non-metric time traces.

As used used in figure 4.7, the `await(a)` simply delays until the external action `a` occurs. Thus, the possible partial traces of `await` are those where the values of the state variables remain unchanged and the action `a` occurs exactly once, at the endpoint of the trace. The possible complete traces are similar, except that the action `a` must never occur.

To give a more detailed semantics for non-terminating loops, we define the set of extensions of a non-terminating execution sequence Z to be the set

$$\mathit{ext}(Z) = \{\, x \in \mathcal{B}(\gamma) : \forall k\ [z_k \in \mathit{pref}(x)]\}.$$

For any non-terminating sequence Z, we require that $\mathit{ext}(Z)$ be non-empty, and have a unique maximal lower bound contained in $\mathit{ext}(Z)$, which we denote $\lim(Z)$. In the above definition of the possible traces of a loop, we modify the definition of the set of non-terminating behaviors $P_{N,\omega}$ to be the set of $\lim(Z)$ for all non-terminating execution sequences Z.

4.5.4.2 Using Metric Time Traces

Analogous to the embedding discussed in the previous subsection, non-metric time traces structures can be embedded into metric-time trace structures. Here continuous dynamics can be represented, as well as timing assumptions about programming language statements. Also, timing constraints that a system must satisfy can be represented, so that the system can be verified against those constraints.

4.5.5 From Continuous Time to Discrete Event

This abstraction is similar in nature to the one presented from continuous time to non-metric time. Similarly to that case, to construct an approximation we must first define the notion of an event at the level of the continuous time traces. Abstraction, in this case, can be done in several ways. One, for example, is to consider an event as the snapshot of the state at certain regular intervals. Another technique consists of abstracting the value domain, and identify an event whenever the signals cross certain discrete thresholds. As was done previously, we take yet another approach, and identify an event whenever any of the signals changes with respect to its previous value. To do that, we refer again to definition 4.35 to make this notion precise.

In the continuous time model signals may change value simultaneously. In the discrete event model, on the other hand, events are totally ordered, even when they have the same time stamp. Hence, after identifying an event, we must also decide how to order simultaneous events in the same time stamp. Because there is no obvious choice, we map each event in continuous time to the set of all possible orderings in discrete event. This choice implies that for each trace in the continuous time model there correspond several traces in the discrete event model. Consequently, the approach based on the homomorphism on traces outlined in the previous section will not work.

To construct a trace in the discrete event model we must create a sequence where each element corresponds to an event for some signal at some time in continuous time. To simplify the task, we introduce two additional, and somewhat more elaborate, trace algebras for the discrete event model.

In the first trace algebra, we construct a “sequence” by taking the set of reals as an index set, and by mapping the index set to sequences of events that represent the delta cycles for each particular time stamp. An empty sequence of delta cycles denotes the absence of events for the particular time stamp. Formally, we define the set of possible traces as:

$$\mathcal{B}(A) = \mathbb{R}^{/} \rightarrow (A \times V)^{\infty},$$

where A is, as usual, the set of signals, and V is the corresponding set of possible values. This formulation clearly includes systems that are not discrete: imagine, for instance, that the sequence corresponding to the delta cycles is non-empty for every $t \in \mathbb{R}^{/}$. Thus we must further restrict the set of possible traces to only those whose set of non-empty time stamps is discrete, as was discussed in subsection 4.3.5.

Projection and renaming are defined as expected. Their formal definition gives us the opportunity to introduce a construction theorems that allows one to build new trace algebras from existing ones. In this particular case, note how the set of traces is defined as a function whose range is the set of traces defined in subsection 4.3.3 for the CSP model. The following theorem shows that when projection and renaming are defined appropriately, the result is always another trace algebra. The proof is simple but tedious, so we omit the details.

Theorem 4.36. Let $\mathcal{C}' = (\mathcal{B}'(A), \mathit{proj}, \mathit{rename})$ be a trace algebra and let Z be a set. Then the trace algebra $\mathcal{C}$ such that:

$$\begin{aligned}
\mathcal{B}(A) &= Z \to \mathcal{B}'(A), \\
\mathit{proj}(B)(x) &= \lambda d \in Z\ [\mathit{proj}(B)(x(d))], \\
\mathit{rename}(r)(x) &= \lambda d \in Z\ [\mathit{rename}(r)(x(d))],
\end{aligned}$$

is a trace algebra.

In our particular case we let $\mathcal{B}'(A) = (A \times V)^{\infty}$, $Z = \mathbb{R}^{\not -}$ and projection and renaming as defined in subsection 4.3.3. Hence for a trace $x \in \mathbb{R}^{\not -} \to (A \times V)^{\infty}$ we have

$$\begin{aligned}
\mathit{proj}(B)(x) &= \lambda t \in \mathbb{R}^{\not -}\ [\mathit{proj}(B)(x(t))], \\
\mathit{rename}(r)(x) &= \lambda t \in \mathbb{R}^{\not -}\ [\mathit{rename}(r)(x(t))].
\end{aligned}$$

A trace structure has again signature $\gamma = (I, O)$ and is otherwise obtained as usual as a set of traces.

The second trace algebra that we introduce is similar to the one just presented, but without ordering information within a time stamp. Then we build a mapping from each of the new traces to a set of discrete event traces, that contain all possible interleavings of the events.

Recall (see above) that traces in the discrete event model of computation are of the form:

$$\mathcal{B}(A) = \mathbb{R}^{\not -} \to (A \times V)^{\infty}.$$

The ordering information in the sequence of delta cycles can be removed by considering the more abstract set of traces:

$$\mathcal{B}'(A) = \mathbb{R}^{\not -} \to 2^{A \times V}.$$

It is easy to construct a function h from $\mathcal{B}$ to $\mathcal{B}'$ that removes the ordering information. If $x \in \mathcal{B}(A)$ is of the form $x = x(t,n)$, we define $x' = h(x)$ as the trace $x' = x'(t)$ such that for all $t \in \mathbb{R}^{+}$

$$x'(t) = \{\, (a,v) \in A \times V : \exists n \in \mathbb{N}\, [x(t,n) = (a,v)\}.$$

It is easy to show that h is well defined, and that it is onto. However h is not one-to-one, so that its inverse h^{-1} maps a single trace $x' \in \mathcal{B}'(A)$ to a set of traces in $\mathcal{B}(A)$. This set of traces corresponds to all possible interleavings of the set of pairs of signals and values, with or without repetitions.

It is now easy to define a function g from traces in the continuous time to traces in the discrete event model without ordering. If $y = y(t,a)$ is a continuous time trace, then define $x' = g(y)$ as the trace $x' = x'(t)$ such that for all $t \in \mathbb{R}^{+}$

$$x'(t) = \{\, (a,v) \in A \times V : x \text{ has an event on signal } a \text{ at time } t \wedge x(t,a) = v\}.$$

We can now define an approximation between the continuous time and the discrete event model based on the functions g and h.

Let $p = (\gamma, P)$ be a trace structure in the continuous time model. To build an upper bound we naturally extend the functions g and h to sets of traces as follows:

$$\Psi_u(p) = (\gamma, h^{-1}(g(P))).$$

A lower bound could be constructed in several ways. Note, however, that without any further constraint the discrete event model can represent continuous functions exactly. In other words, since our mapping on trace structures is actually one-to-one, it does not constitute an abstraction. The obvious choice in this case is therefore to simply have

$$\Psi_l(p) = \Psi_u(p),$$

for all T.

The key to getting a real abstraction is that of defining exactly the conditions that make the discrete event model *discrete*. This can be done by replacing the set of reals in the definition of the trace algebra with a different set D. The result is a parametrized trace algebra

$$\mathcal{B}(A) = D \to (A \times V)^{\infty}.$$

Depending on the choice of D different kinds of abstractions are possible.

4.5.6 From Discrete Event to Process Networks

In this section we will explore the relationships between the discrete event model presented in subsection 4.3.5 and the process network model presented in subsection 4.3.4. We will use the simple version of DE that consists of a sequence of events.

During the presentation we will refer to figure 4.8 and figure 4.9. Figure 4.8 depicts the mappings that relate traces in the different domains. Figure 4.9 shows the corresponding mappings when applied to the domains of trace structures (sets of traces).

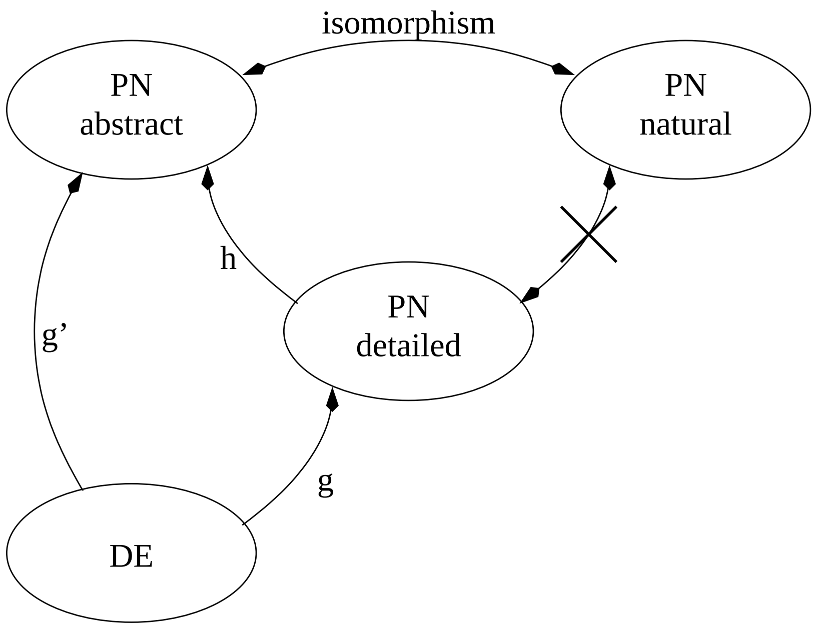

Figure 4.8: Relations between trace algebras

We have already pointed out that the natural domain is that of functions on streams. Our initial abstract formalization is a model of traces that is isomorphic to the set of streams. However, the corresponding formalization in terms of trace structures led to a problem with the composition operator: in the original model, composition is defined so that it includes only the least fixed-point of the functions that satisfy a certain equation; in our model, instead, composition includes all the fixed-points. Thus we are unable to find an isomorphism between the trace structures of our formalization and the agents in the natural domain, that is a one-to-one mapping that preserves composition.

We have then developed a more detailed domain, in which sequences are used to emphasize the order relationships between inputs and outputs that allows us to build the fixed-point

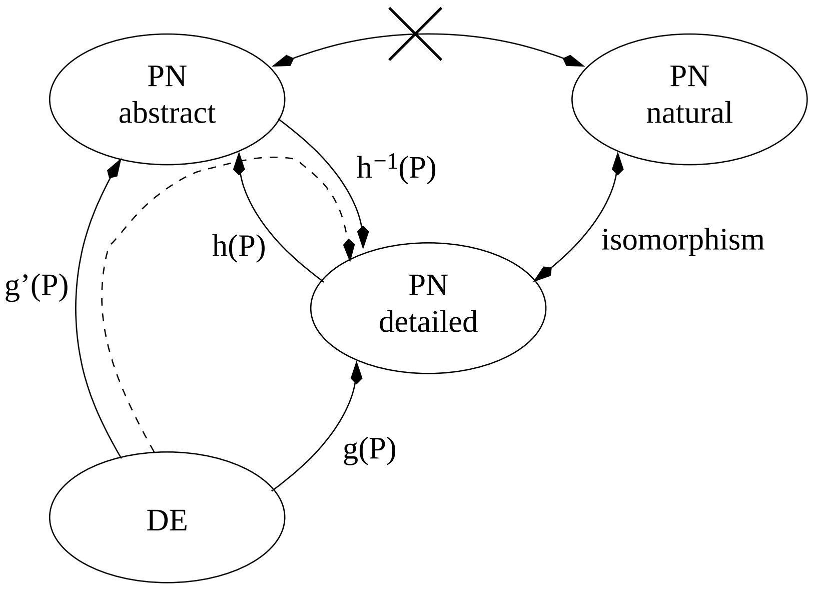

Figure 4.9: Relations between trace structure algebras

in the composition. By doing this we abandon the isomorphism of the traces with the domain of streams. To be classified as process network agents, trace structures in this formalization must satisfy constraints that ensure that a function on stream is in fact being constructed. The discussion then suggests that there is an isomorphism (which preserves the operation of composition) between the detailed model of trace structures and the agents in the natural semantic domain.

Recall that traces in the abstract process network algebra belong to the set:

$$\mathcal{B}(A) = A \to V^{\infty}.$$

Traces in the more detailed algebra belong to the other set:

$$\mathcal{B}(A) = (A \times V)^{\infty}.$$

As shown in subsection 4.3.4, traces in this more detailed model can be mapped into traces in the more abstract model by virtue of a homomorphism h that removes the order relationships across signals. When naturally extended to trace structures (i.e., to set of traces), h maps agents in the detailed domain into agents in the abstract domain. The homomorphism on individual traces is obviously not one-to-one. However, when considered as a mapping of trace structures from the (restricted set of agents in the) detailed trace structure algebra into the more abstract algebra, the function is a one-to-one mapping. In fact, if two trace structures p_1 and p_2 map into the same trace structure in the abstract algebra, then they must have the same fundamental mode representation R_0. The inductive construction of equation 4.1 then shows that $p_1 = p_2$. Because h is one-to-one when applied to agents, there is an inverse function h^{-1} from the abstract trace structure algebra into the detailed algebra.

The relationships between the discrete event and the process network model of computation can be described as a mapping to one of the two formulations. Recall that traces in the discrete event model are of the form:

$$\mathcal{B}(A) = (A \times V \times \mathbb{R}^{\not-})^{\infty}.$$

A straightforward mapping can be constructed from the discrete event traces to the detailed process network traces. The mapping is a function g that simply removes the time stamp from the sequence. In other words, if

$$x = \langle (a_0, v_0, t_0), (a_1, v_1, t_1), \ldots \rangle$$

is a discrete event trace, then

$$g(x) = \langle (a_0, v_0), (a_1, v_1), \ldots \rangle.$$

This mapping is a homomorphism on traces, in that it commutes with the operations of projection and renaming. In other words, if x is a discrete event trace, then

$$\begin{aligned} g(\mathit{proj}(B)(x)) &= \mathit{proj}(B)(g(x)), \\ g(\mathit{rename}(r)(x)) &= \mathit{rename}(r)(g(x)). \end{aligned}$$

The natural extension to sets of traces $g(P)$ of the homomorphism g is a function that maps discrete event agents into process network agents. This function is an upper bound Ψ_u of a conservative approximation:

$$\Psi_u(T) = (\gamma, g(P)).$$

For the lower bound Ψ_l we must map to a restricted set of traces. Namely, the inverse image of $\Psi_l(P)$ should map to traces that are *only* in P. This can be accomplished using the homomorphism g as follows:

$$\Psi_l(p) = (\gamma, g(P) - g(\mathcal{B}(A) - P)),$$

where $\mathcal{B}(A) - P$ is the complement of P with respect to the universe of traces. This lower bound can be made tighter by considering only the traces that occur in the agents that form the trace structure algebra.

It can be shown that the two mappings so defined form a conservative approximation. This formulation can be generalized. In fact, nothing in the derivation of Ψ_u and Ψ_l depends on the particular models of computation considered. Hence, whenever there is a homomorphism g between the sets of traces of two different models of computation, we can construct a conservative approximation using the same formulation. We refer the reader to [14] for more details on this technique.

What does this mapping look like? Consider for example the inverter shown in figure 4.10. It has an input a and an output b. If we assume the inverter has a constant positive delay δ, then a possible trace of the agent in the discrete event model might look like the following:

$$x = \langle (a, 0, 0), (b, 1, \delta), (a, 1, 3.5), (b, 0, 3.5 + \delta), \ldots \rangle,$$

assuming that $\delta < 3.5$. The corresponding trace in the process networks model is

$$x' = \langle (a, 0), (b, 1), (a, 1), (b, 0), \ldots \rangle.$$

This trace is included in the upper bound computed by Ψ_u. If the agent does not contain a trace for any possible delay δ, then this trace is not included in the lower bound Ψ_l. In fact, a trace y with a similar sequence of events, but different delay, would be in $\mathcal{B}(A)$ but not in P; because g discards the delays, $g(x) = g(y)$ and, by definition of Ψ_l above, x is removed from the mapping. In other words, the process network model does not distinguish between agents with different delays and we are indeed computing an approximation.

It is interesting to consider the inverse of this conservative approximation. The inverse mapping corresponds to trying to embed an agent of the process networks model into a discrete event context. Here we must find agents p such that $\Psi_u(p) = \Psi_l(p) = p'$. Because of the particular abstraction we have employed, this occurs whenever the agent p has non-deterministic delay. In this case, given a trace x, all other traces y with the same sequence of events but different delay are included in the set of possible traces of the agent, and therefore retained in the computation of the lower bound. Hence, for every agent p' in the process network model of computation, there exists an agent $p = \Psi_{inv}(p')$ in the discrete event model, where p has the same behaviors as p' and chooses non-deterministically the delay of the outputs. Any deterministic implementation of this embedding will therefore have to make an upfront choice regarding the timing of the agent.

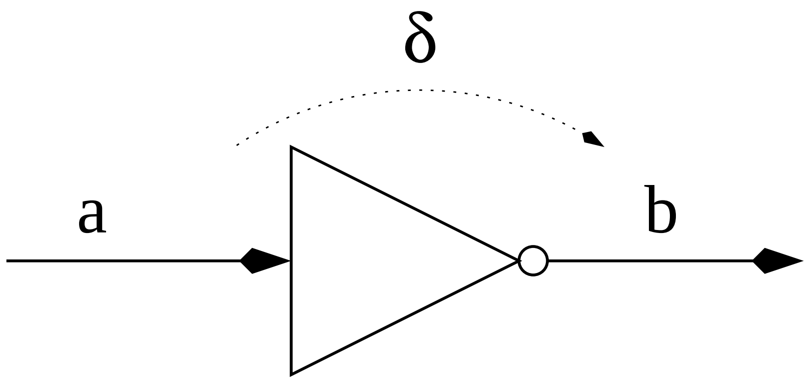

Figure 4.10: Inverter agent with delay δ

The functions Ψ_u and Ψ_l that we have just defined certainly constitute an abstraction. However, in this particular case the abstraction does not ensure that the corresponding trace structure in the process network algebra satisfies the constraints for that model defined in subsection 4.3.4 involving equation 4.1. In fact, for each trace in the discrete event model there should correspond several (possibly infinitely many) traces in the process network model that include all possible delayed outputs. It is possible to consider only a restricted version of the discrete event trace structures that maps correctly in the detailed process network algebra. To simplify this task, we will take an alternative route and use the abstract process network algebra as an intermediate step.

Notice that the abstract process network trace structure algebra requires that agents be monotonic and functional. This requirement must still be satisfied by the discrete event agent that we want to abstract. An equivalent constraint that can be imposed at the discrete event level is that of receptiveness. Intuitively, a trace structure is receptive if it can't constrain the value of its inputs. The technical definition of receptiveness (see [34]) requires the device of infinite games: an agent is receptive if it can always respond to an input with outputs that make the trace one of its possible traces.

We can show that if a discrete event agent p is both receptive and functional, then it is also monotonic (where the prefix order corresponds to the usual prefix on sequences). In fact, assume it is not monotonic. Then there are traces x and y in p such that $\mathit{proj}(I)(x) \sqsubseteq \mathit{proj}(I)(y)$, but $\mathit{proj}(O)(x) \not\sqsubseteq \mathit{proj}(O)(y)$. But if p is receptive, then x can be extended to a trace x' such that $\mathit{proj}(I)(x') = \mathit{proj}(I)(y)$ and $x' \in T$. By the functionality assumption, $x' = y$. But $x \sqsubseteq x'$, a contradiction. Hence p must be monotonic.

A homomorphism g' between the discrete event traces and the abstract process network traces is given by the composition of g and h. The natural extension to sets gives us a mapping $g'(P)$ on trace structures and a corresponding conservative approximation. An approximation from the discrete event trace structure algebra to the detailed process networks trace structure algebra can now be constructed by taking the composition of the mapping $g\!\!/$ and h^{-1} as shown in figure 4.9.

Chapter 5

Protocol Conversion

In the previous chapter we have presented several examples of models of computation formalized as trace-based agent algebras. In this chapter we derive a conformance order and a mirror function for trace structure algebras, and introduce a richer trace-based model, derived from Dill's trace structures [33], in which agents have two sets of traces, corresponding to *successful* and *faulty* behaviors, respectively. This representation is extremely useful in interface specifications, where the model must provide information about the conditions of correct operation. We demonstrate its use in the problem of protocol conversion.

5.1 Conformance and Mirrors for Trace Structure Algebras

In section 4.4 we have defined a simple ordering of trace structures based on trace containment (see definition 4.15). The order relationship is there restricted to agents that have the same alphabet. We now wish to generalize this definition, and to allow an order relationship to exist between agents that have different alphabet. We do so in two steps. We initially define an order which is convenient for deriving a mirror function in the algebra. This order, however, is such that the renaming operator is not $\top$-monotonic. Our venture into a non-$\top$-monotonic model is however only temporary, and required simply to make the mathematics more tractable. The order can then be modified to make all operators $\top$-monotonic by restricting the set of refinements to agents that have a smaller alphabet. In this case, the mirror function must also be modified, in a way similar to the one used in example 3.84. To keep the presentation simple, we will omit the details of this last step.

We begin by defining the agent order. Ideally, we would like to closely follow the definition of definition 4.15. For example, if p and p' are trace structures with alphabets A and A', respectively, we could require that the set of traces of p be a subset of *the projection onto* A of the set of traces of p'. Our objective is however that of deriving a mirror function, which, as we know, exists only if parallel composition is able to characterize the order using a single agent. This is in turn possible only if the inverse of the projection operator is surjective, in the sense that if $A \subseteq A'$, then for all traces $x \in \mathcal{B}(A)$, there exists a trace $y \in \mathcal{B}(A')$ such that $x = \mathit{proj}(A)(y)$. In that case, in fact, agents with larger alphabets have complete knowledge about agents with smaller alphabets. In addition, since we are not restricting the alphabets to be contained into one other, we look for a more general definition. The following definition is equivalent to our first attempt when inverse projection is surjective, and is convenient to use in our subsequent proofs. We say that p is less than or equal to p' whenever for all the traces in the combined alphabet $A \cup A'$, if the projection on A is contained in p, then the projection on A' is contained in p'.

Definition 5.1 (Agent Order). Let $p = (A, P)$ and $p' = (A', P')$ be two trace structures. Then

$$p \preceq p' \Leftrightarrow \forall x \in \mathcal{B}(A \cup A'), \mathit{proj}(A)(x) \in P \Rightarrow \mathit{proj}(A')(x) \in P'.$$

Note that if $A = A'$, then the above definition reduces to $P \subseteq P'$, consistently with definition 4.15. We must however show that the relation defined above is a preorder, i.e., that it is reflexive and transitive. This is the case only if, as described earlier, the inverse of the projection function is surjective.

Theorem 5.2. Let $\mathcal{C}$ be a trace algebra such that for all alphabets A and A', if $A \subseteq A'$ then for all traces $x \in \mathcal{B}(A)$ there exists a trace $y \in \mathcal{B}(A')$ such that $x = \mathit{proj}(A)(y)$. Then $\preceq$ is a preorder.

Proof: The relation $\preceq$ is clearly reflexive. We will use the following result.

Lemma 5.3. Let A, A' and A'' be alphabets such that $A \subseteq A'$ and $A \subseteq A''$. Then, for all $u \in \mathcal{B}(A')$ there exist $v \in \mathcal{B}(A'')$ and $w \in \mathcal{B}(A' \cup A'')$ such that $\mathit{proj}(A)(u) = \mathit{proj}(A)(v)$, $\mathit{proj}(A')(w) = u$ and $\mathit{proj}(A'')(w) = v$.

Proof: Let $u \in \mathcal{B}(A')$ be a trace. Let be $u' = \mathit{proj}(A' \cap A'')(u)$. Clearly $u' \in \mathcal{B}(A' \cap A'')$. Therefore, by hypothesis, there exists $v \in \mathcal{B}(A'')$ such that $\mathit{proj}(A' \cap A'')(v) = u'$. By T4, $\mathit{proj}(A)(u) = \mathit{proj}(A)(\mathit{proj}(A' \cap A'')(u)) = \mathit{proj}(A)(u') = \mathit{proj}(A)(\mathit{proj}(A' \cap A'')(v)) = \mathit{proj}(A)(v)$. In addition, by T8, there exists $w \in \mathcal{B}(A' \cup A'')$ such that $\mathit{proj}(A')(w) = u$ and $\mathit{proj}(A'')(w) = v$. □

To show that it is transitive, assume $p \preceq p'$ and that $p' \preceq p''$. We must show that $p \preceq p''$. To do so, let $z \in \mathcal{B}(A \cup A'')$ be such that $\mathit{proj}(A)(z) \in P$. We show that $\mathit{proj}(A'')(z) \in P''$.

In fact, by lemma 5.3, there exist $x \in \mathcal{B}(A \cup A')$ and $z' \in \mathcal{B}(A \cup A' \cup A'')$ such that $\mathit{proj}(A)(z) = \mathit{proj}(A)(x)$, $\mathit{proj}(A \cup A'')(z') = z$ and $\mathit{proj}(A \cup A')(z') = x$. Note also that $\mathit{proj}(A)(z') = \mathit{proj}(A)(\mathit{proj}(A \cup A')(z')) = \mathit{proj}(A)(x)$ and that $\mathit{proj}(A')(z') = \mathit{proj}(A')(\mathit{proj}(A \cup A')(z')) = \mathit{proj}(A')(x)$.

Likewise, there exist $y \in \mathcal{B}(A' \cup A'')$ and $z'' \in \mathcal{B}(A \cup A' \cup A'')$ such that $\mathit{proj}(A')(y) = \mathit{proj}(A')(z')$, $\mathit{proj}(A \cup A' \cup A'')(z'') = z'$ and $\mathit{proj}(A' \cup A'')(z'') = y$. Therefore, by T2, $z'' = z'$.

Since $\mathit{proj}(A)(z) \in P$ and $\mathit{proj}(A)(z) = \mathit{proj}(A)(x)$, and since $p \preceq p'$, by definition 5.1, $\mathit{proj}(A')(x) \in P'$. But $\mathit{proj}(A')(y) = \mathit{proj}(A')(z') = \mathit{proj}(A')(x)$, therefore $\mathit{proj}(A')(y) \in P'$. Since $p' \preceq p''$, $\mathit{proj}(A'')(y) \in P''$. Note however that $\mathit{proj}(A'')(z'') = \mathit{proj}(A'')(z') = \mathit{proj}(A'')(\mathit{proj}(A \cup A'')(z')) = \mathit{proj}(A'')(z)$. Similarly, $\mathit{proj}(A'')(z'') = \mathit{proj}(A'')(\mathit{proj}(A' \cup A'')(z'')) = \mathit{proj}(A'')(y)$. Hence, $\mathit{proj}(A'')(z) = \mathit{proj}(A'')(y)$. Therefore, $\mathit{proj}(A'')(z) \in P''$. Consequently, by definition 5.1, $p \preceq p''$. □

For the rest of this chapter we assume that the hypothesis of theorem 5.2 are satisfied by the trace algebras that we work with. The constraints essentially implies that the set of traces with a certain alphabet can be used to represent (via projection) all the traces that have smaller alphabet, or that, in other words, adding information to a trace does not destroy the existing information. Note that all of the examples presented in this work satisfy those assumptions.

As anticipated, the renaming operator is not $\top$-monotonic relative to this agent order. For the purpose of the mirror, however, we focus our attention on the parallel composition operator only, since it is the only operator responsible for the conformance order relative to composition required for a mirror function. The next result shows that, indeed, parallel composition is $\top$-monotonic relative to the agent order.

Theorem 5.4. Parallel composition is $\top$-monotonic relative to $\preceq$.

Proof: Let $p = (A, P)$ and $p' = (A', P')$ be trace structures such that $p \preceq p'$. We must show that for all $q = (A_q, P_q)$, $p \parallel q \preceq p' \parallel q$. We have

$$\begin{aligned}
p \parallel q &= (A \cup A_q, PQ = \{ x \in \mathcal{B}(A \cup A_q) : \mathit{proj}(A)(x) \in P \wedge \mathit{proj}(A_q)(x) \in P_q \}) \\
p' \parallel q &= (A' \cup A_q, P'Q = \{ x \in \mathcal{B}(A' \cup A_q) : \mathit{proj}(A')(x) \in P \wedge \mathit{proj}(A_q)(x) \in P_q \})
\end{aligned}$$

By hypothesis, since $p \preceq p'$, for all $x \in \mathcal{B}(A \cup A')$,

$$\textit{proj}\,(A)(x) \in P \Rightarrow \textit{proj}\,(A')(x) \in P'.$$

We must show that $p \parallel q \preceq p' \parallel q$, that is, for all $x \in \mathcal{B}(A \cup A' \cup A_q)$,

$$\textit{proj}\,(A \cup A_q)(x) \in PQ \Rightarrow \textit{proj}\,(A' \cup A_q)(x) \in P'Q.$$

Let $x \in \mathcal{B}(A \cup A' \cup A_q)$. The proof then consists of the following series of implications:

$$\begin{aligned}
&\textit{proj}\,(A \cup A_q)(x) \in PQ \\
&\quad\Rightarrow \quad \textit{proj}\,(A)(\textit{proj}\,(A \cup A_q)(x)) \in P \wedge \textit{proj}\,(A_q)(\textit{proj}\,(A \cup A_q)(x)) \in P_q \\
&\quad\;\; \text{since } A \subseteq A \cup A_q \text{ and } A_q \subseteq A \cup A_q\text{, by A20} \\
&\quad\Rightarrow \quad \textit{proj}\,(A)(x) \in P \wedge \textit{proj}\,(A_q)(x) \in P_q \\
&\quad\;\; \text{by hypothesis} \\
&\quad\Rightarrow \quad \textit{proj}\,(A')(x) \in P \wedge \textit{proj}\,(A_q)(x) \in P_q \\
&\quad\;\; \text{since } A' \subseteq A' \cup A_q \text{ and } A_q \subseteq A' \cup A_q\text{, by A20} \\
&\quad\Rightarrow \quad \textit{proj}\,(A')(\textit{proj}\,(A' \cup A_q)(x)) \in P \wedge \textit{proj}\,(A_q)(\textit{proj}\,(A' \cup A_q)(x)) \in P_q \\
&\quad\Rightarrow \quad \textit{proj}\,(A' \cup A_q)(x) \in P'Q.
\end{aligned}$$

□

The next step for deriving a mirror function is to choose a conformance set G such that the algebra has a G-conformance order relative to composition. This can be accomplished by having $G = \{\, p = (A, P) : P = \emptyset\}$. Clearly, G is downward closed relative to $\preceq$. The next theorem shows that G induces the required conformance order relative to composition. Note however that, in general, a conformance order depends on the particular set of trace structures that a trace structure algebra contains. Different sets of trace structures, in fact, induce different sets of contexts, and therefore a different notion of conformance. For the purpose of our work here we assume that the trace structure algebra is *complete*, i.e., it contains the set of *all* trace structures.

Theorem 5.5. Trace structure algebras have a G-conformance order relative to composition.

Proof: Let $p = (A, P)$ and $p' = (A', P')$ be trace structures. We must show that $p \preceq p'$ if and only if for all trace structures q, $p' \parallel q \in G$ implies $p \parallel q \in G$.

The forward direction follows from $\top$-monotonicity, since G is downward closed.

For the reverse direction, assume that for all trace structures q, $p' \parallel q \in G$ implies $p \parallel q \in G$. We must show that for all traces $x \in \mathcal{B}(A \cup A')$,

$$\mathit{proj}(A)(x) \in P \Rightarrow \mathit{proj}(A')(x) \in P'.$$

Let $q = (A', \mathcal{B}(A') - P')$. Then, clearly, $p' \parallel q \in G$. Therefore, also

$$p \parallel q = (A \cup A', \{\, x \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(x) \in P \wedge \mathit{proj}(A')(x) \in \mathcal{B}(A') - P' \}) \in G.$$

Hence, for all traces $x \in \mathcal{B}(A \cup A')$,

$$\mathit{proj}(A)(x) \in P \Rightarrow \mathit{proj}(A')(x) \notin \mathcal{B}(A') - P',$$

or, equivalently

$$\mathit{proj}(A)(x) \in P \Rightarrow \mathit{proj}(A')(x) \in P',$$

which proves the result. □

We now explore the structure of the compatibility set of a trace structure p' relative to the conformance set G. The form of the compatibility set is in fact related to the order according to the results of lemma 3.75 and lemma 3.76. The notation for the compatibility set makes use of the inverse of the projection function, defined as follows.

Definition 5.6 (Inverse Projection). Let $x \in \mathcal{B}(A)$ be a trace. Then

$$\mathit{proj}(A')^{-1}(x) = \{\, y \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(y) = x \}.$$

If a trace belongs to the trace sets of different alphabets (say, $x \in \mathcal{B}(A_1)$ and $x \in \mathcal{B}(A_2)$), then the notation that we use for inverse projection is ambiguous since it does not make clear which alphabet is assumed for the argument. This is not usually a problem, since the alphabet of the trace is clear from context. The alternative is to explicitly add the alphabet as a parameter to the operator. We do not use this solution to avoid cluttering our notation.

Like projection, inverse projection is naturally extended to sets of traces, provided that all the traces in the set have the same alphabet. The following result shows that projection and inverse projection distribute over set union.

Lemma 5.7. Let A and A' be alphabets and let $S, F \subseteq \mathcal{B}(A)$ be sets of traces. Then,

1. $\mathit{proj}(A')(S \cup F) = \mathit{proj}(A')(S) \cup \mathit{proj}(A')(F)$.

2. $\mathit{proj}(A')^{-1}(S \cup F) = \mathit{proj}(A')^{-1}(S) \cup \mathit{proj}(A')^{-1}(F)$.

Proof: The proof of item 1 is composed of the following series of equalities.

$$\begin{aligned}
\mathit{proj}(A')(S \cup F) &= \{\, y \in \mathcal{B}(A') : \exists x \in S \cup F\ [y = \mathit{proj}(A')(x)]\} \\
&= \{y \in \mathcal{B}(A') : \exists x \in S\ [y = \mathit{proj}(A')(x)] \vee \\
&\quad \vee \exists x \in F\ [y = \mathit{proj}(A')(x)]\} \\
&= \{\, y \in \mathcal{B}(A') : \exists x \in S\ [y = \mathit{proj}(A')(x)]\} \cup \\
&\quad \cup \{\, y \in \mathcal{B}(A') : \exists x \in F\ [y = \mathit{proj}(A')(x)]\} \\
&= \mathit{proj}(A')(S) \cup \mathit{proj}(A')(F).
\end{aligned}$$

Similarly, the proof of item 1 is composed of the following series of equalities.

$$\begin{aligned}
\mathit{proj}(A')^{-1}(S \cup F) &= \{\, y \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(y) \in S \cup F\} \\
&= \{\, y \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(y) \in S \vee \mathit{proj}(A)(y) \in F\} \\
&= \{\, y \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(y) \in S\} \cup \\
&\quad \cup \{\, y \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(y) \in F\} \\
&= \mathit{proj}(A')^{-1}(S) \cup \mathit{proj}(A')^{-1}(F).
\end{aligned}$$

□

Using this notation, the following result makes clear the connection between our notion of order and the traditional ordering based on trace containment.

Corollary 5.8. Let $p = (A, P)$ and $p' = (A', P')$ be trace structures. Then

$$p \preceq p' \Leftrightarrow \mathit{proj}(A')(\mathit{proj}(A')^{-1}(P)) \subseteq P'.$$

Proof: For the forward direction, assume $p \preceq p'$. Let $x' \in \mathit{proj}(A')(\mathit{proj}(A')^{-1}(P))$. Then there exists $x \in \mathcal{B}(A \cup A')$ such that $\mathit{proj}(A)(x) \in P$ and $\mathit{proj}(A')(x) = x'$. Since $p \preceq p'$, by definition 5.1, $\mathit{proj}(A')(x) \in P'$. Therefore $x' \in P'$.

For the reverse direction, assume $x \in \mathcal{B}(A \cup A')$ is a trace such that $\mathit{proj}(A)(x) \in P$. Then $x \in \mathit{proj}(A')^{-1}(P)$ and $\mathit{proj}(A')(x) \in \mathit{proj}(A')(\mathit{proj}(A')^{-1}(P))$. Hence, since $\mathit{proj}(A')(\mathit{proj}(A')^{-1}(P)) \subseteq P'$, $\mathit{proj}(A')(x) \in P'$. Therefore, by definition 5.1, $p \preceq p'$. □

If in addition $A \subseteq A'$, the above simply means that the inverse projection of p is contained in p', as shown below.

Lemma 5.9. Let $A \subseteq A'$. Then $\mathit{proj}(A')(\mathit{proj}(A')^{-1}(P)) = \mathit{proj}(A')^{-1}(P)$.

Proof: Clearly, by definition 5.6, $\mathit{proj}(A')^{-1}(P) \subseteq \mathcal{B}(A \cup A')$. But since $A \subseteq A'$, $A \cup A' = A'$. Therefore, by T2, $\mathit{proj}(A')(\mathit{proj}(A')^{-1}(P)) = \mathit{proj}(A')^{-1}(P)$. □

Recall that two agents p and p' are compatible if their parallel composition is defined and the result of the composition is an agent in the conformance set. The compatibility set of an agent p is the set of all agents compatible with p'. Using inverse projection, we can express the compatibility set of a trace structure p' explicitly, as shown by the next theorem.

Theorem 5.10 (Compatibility Set). Let $p = (A, P)$ and $p' = (A', P')$ be trace structures. Then $p \in \mathit{cmp}(p')$ if and only if

$$P \subseteq \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')).$$

Proof: Let $p' = (A', P')$ be a trace structure. The compatibility set of p' is the set

$$\mathit{cmp}(p') = \{p = (A, P) : p \parallel p' \in G\}.$$

Therefore

$$\begin{aligned} & p = (A, P) \in \mathit{cmp}(p') \\ \Leftrightarrow\quad & \{x \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(x) \in P \wedge \mathit{proj}(A')(x) \in P'\} = \emptyset \\ \Leftrightarrow\quad & P \cap \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')) = \emptyset \\ \Leftrightarrow\quad & P \subseteq \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')). \end{aligned}$$

□

Recall that since parallel composition is $\top$-monotonic and since G is downward closed and the algebra has a G-conformance order relative to composition, the set of maximal elements of the compatibility set are sufficient to characterize the entire set. Our next result shows that the compatibility set actually contains a greatest element. Recall, however, that since $\preceq$ is a preorder, in general there may be several greatest elements.

Theorem 5.11 (Greatest Elements). Let $p' = (A', P')$ be a trace structure. A trace structure $p = (A, P)$ such that $A' \subseteq A$ and

$$P = \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')).$$

is a greatest element of $\mathit{cmp}(p')$.

Proof: To prove the result, we start by showing that all the agents with the required property above are equivalent relative to the agent ordering. We then choose one representative of the equivalence class, and show that it is a greatest element.

Let $p_1 = (A_1, P_1)$ and $p_2 = (A_2, P_2)$ be two trace structures such that

$$\begin{aligned} P_1 &= \mathcal{B}(A_1) - \mathit{proj}(A_1)(\mathit{proj}(A_1)^{-1}(P')) \\ P_2 &= \mathcal{B}(A_2) - \mathit{proj}(A_2)(\mathit{proj}(A_2)^{-1}(P')) \end{aligned}$$

and such that $A' \subseteq A_1$ and $A' \subseteq A_2$. We show that $p_1 \approx p_2$. In fact, since $A' \subseteq A_1$ and $A' \subseteq A_2$, by lemma 5.9,

$$\begin{aligned} P_1 &= \mathcal{B}(A_1) - \mathit{proj}(A_1)^{-1}(P') \\ P_2 &= \mathcal{B}(A_2) - \mathit{proj}(A_2)^{-1}(P') \end{aligned}$$

To show that $p_1 \preceq p_2$, we use the contrapositive of the definition and show that if $x \in \mathcal{B}(A_1 \cup A_2)$ is a trace such that $\mathit{proj}(A_2)(x) \notin P_2$, then $\mathit{proj}(A_1)(x) \notin P_1$. The proof consists of the following series of implications:

$$\begin{aligned} &\mathit{proj}(A_2)(x) \notin P_2 \\ &\quad\Rightarrow \quad \mathit{proj}(A_2)(x) \in \mathit{proj}(A_2)^{-1}(P') \\ &\quad\Rightarrow \quad \mathit{proj}(A')(\mathit{proj}(A_2)(x)) \in P' \\ &\quad\ \text{since } A' \subseteq A_2 \\ &\quad\Rightarrow \quad \mathit{proj}(A')(x) \in P' \\ &\quad\ \text{since } A' \subseteq A_1 \\ &\quad\Rightarrow \quad \mathit{proj}(A')(\mathit{proj}(A_1)(x)) \in P' \\ &\quad\Rightarrow \quad \mathit{proj}(A_1)(x) \in \mathit{proj}(A_1)^{-1}(P') \\ &\quad\Rightarrow \quad \mathit{proj}(A_1)(x) \notin P_1. \end{aligned}$$

Therefore, $p_1 \preceq p_2$. Symmetrically, $p_2 \preceq p_1$. Hence, $p_1 \approx p_2$.

We now take $q = (A', \mathcal{B}(A') - P')$ as a representative of the class of compatible agents such that $A' \subseteq A$. It remains to show that if $p = (A, P)$ is a trace structure such that

$$P \subseteq \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P'))$$

(i.e., p is in the compatibility set of p'), then $p \preceq q$.

Recall that for agents with the same alphabet the order reduces to trace inclusion. Therefore it is sufficient to show that $p \preceq q$ only for the agents such that

$$P = \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')),$$

since the result holds for the remaining agents by transitivity. In addition, we have already showed that $p \preceq q$ in case $A' \subseteq A$, since, in that case, p and q are order equivalent.

Assume now that A does not include A' (i.e., $A' \not\subseteq A$). We again use the contrapositive and show that if $x \in \mathcal{B}(A \cup A')$ is a trace such that $\mathit{proj}(A')(x) \notin \mathcal{B}(A') - P'$, then $\mathit{proj}(A)(x) \notin P$. In fact,

$$\begin{aligned}
&\mathit{proj}(A')(x) \notin \mathcal{B}(A') - P' \\
&\quad \Rightarrow \quad \mathit{proj}(A')(x) \in P' \\
&\quad \Rightarrow \quad x \in \mathit{proj}(A)^{-1}(P') \\
&\quad \Rightarrow \quad proj(A)(x) \in \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')) \\
&\quad \Rightarrow \quad proj(A)(x) \notin \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')) \\
&\quad \Rightarrow \quad proj(A)(x) \notin P.
\end{aligned}$$

Hence $p \preceq q$, which proves the result. □

The compatibility set contains several greatest elements, which are equivalent to each other in terms of the agent ordering. Notice that as the alphabet gets larger more behaviors are added to a greatest element so that the new signals are used in ways (in fact, in all the ways) that are compatible with the behaviors of the original agents. This information is however irrelevant to the characterization of the order. Therefore it is natural to choose among the greatest element the one that has the smaller alphabet.

Corollary 5.12. $p = (A', \mathcal{B}(A') - P')$ is a greatest element of $\mathit{cmp}(p')$.

Given our choice of conformance set G, we have shown that trace structure algebra have a G-conformance order relative to composition, and that the compatibility set of every trace structure has a greatest element. The following result is now straightforward.

Corollary 5.13. Trace structure algebras have a mirror function relative to G, and for all trace structures $p = (A, P)$,

$$\mathit{mirror}(p) = (A, \mathcal{B}(A) - P).$$

Proof: The result follows directly from theorem 3.80. □

It is easy to show that trace structure algebras are also rectifiable, so that we can apply the local specification synthesis technique. It is instructive to compare our result with the solution proposed by Yevtushenko et al. [98] (see subsection 1.8.11 for their notation and a comparison), where the local specification is given by

$$S = \overline{A \cdot \overline{C}},$$

for different notions of the composition operator. To do so, we must restrict our attention to a solution with a specific alphabet, and therefore consider the formulation of theorem 3.120. Assuming that trace structures are built over the language (set of strings) of an alphabet, we have

$$p_1 \preceq \mathit{mirror}(\mathit{proj}(A_1)(p_2 \parallel \mathit{mirror}(p)),$$

where p is the global specification, p_2 is the context, and p_1 the unknown component. Our definition of mirror is equivalent to complementing the language, therefore

$$p_1 \preceq \overline{\mathit{proj}(A_1)(p_2 \parallel \overline{p})}.$$

Yevtushenko et al. prove their solution for both *parallel* and *synchronous* composition. In our case, the specialization involves only changing the definition of projection, since the definition of parallel composition is derived from that of projection. It is easy to show that the *parallel* composition corresponds to having projection retain the length of the sequence by inserting empty symbols in place of those that must be removed. Analogously, *parallel* composition corresponds to having projection alter the length of the sequence by removing the symbols that should not be retained. Note, however, that by employing our framework we need only prove that the operations satisfy the required properties for trace algebras in order to ensure the validity of the result.

5.2 Two-Set Trace Structures

The trace structure algebra model that we have been using so far is able to characterize the set of *possible* traces that each agent might exhibit in response to actions from its environment. There are applications however where we would like to express more than simply the possibility of a behavior occurring for an agent. For example, while a behavior may be possible for an agent, it may cause the agent to *fail*, or to enter a bad state. In other words, we are looking for a model that is able to express the circumstances under which the agent is operated on correctly by the environment.

One such model is the trace structure for asynchronous circuits based on *successes* and *failures* introduced by Dill [34]. In this model, a trace is a success if it is a possible behavior and is accepted as a correct use by the agent. A trace is a failure if it is a possible behavior of the agent, but is contrary to its intended use. In this section we generalize the model introduced by Dill to abstract executions, and derive a conformance order and a mirror function.

The agent model is based on the same notion of trace algebra as the traditional one-set trace structure. Trace structures are however augmented with an additional set of traces, as follows.

Definition 5.14 (Two-Set Trace Structure). Let $\mathcal{C} = (\mathcal{B}, \mathit{proj}, \mathit{rename})$ be a trace algebra over $\mathcal{A}$. The set of *two-set trace structures* over $\mathcal{C}$ is the set of ordered tuples (A, S, F), where

- A is an alphabet over $\mathcal{A}$,
- S is a subset of $\mathcal{B}(A)$, and
- F is a subset of $\mathcal{B}(A)$.

We call A the alphabet of the trace structure, S the set of *successful* traces and F the set of *failure* traces of the trace structure $p = (A, S, F)$.

Both successes and failures are legal behaviors of an agent. Therefore, for a two-set trace structure $p = (A, S, F)$ we define $P = S \cup F$ to be the set of *possible* traces of the trace structure. This notation is consistent with the one employed for the traditional one-set model. Note also that S and F need not be disjoint.

A trace structure algebra based on two-set trace structures is defined as usual, with the appropriate changes in the way the operators are computed.

Definition 5.15 (Two-Set Trace Structure Algebra). Let $\mathcal{C} = (\mathcal{B}, \mathit{proj}, \mathit{rename})$ be a trace algebra over $\mathcal{A}$ and let $\mathcal{T}$ be a subset of the two-set trace structures over $\mathcal{C}$. Then $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ is a *two-set trace structure algebra* if and only if the domain $\mathcal{T}$ is closed under the following operations on trace structures: parallel composition (definition 5.16), projection (definition 5.17) and renaming (definition 5.18).

Definition 5.16 (Parallel Composition). $p = p_1 \parallel p_2$ is always defined and

$$\begin{aligned}
A &= A_1 \cup A_2 \\
S &= \{ x \in \mathcal{B}(A) : \mathit{proj}(A_1)(x) \in S_1 \wedge \mathit{proj}(A_2)(x) \in S_2 \} \\
F &= \{ x \in \mathcal{B}(A) : \mathit{proj}(A_1)(x) \in F_1 \wedge \mathit{proj}(A_2)(x) \in P_2 \} \cup \\
&\quad\ \{ x \in \mathcal{B}(A) : \mathit{proj}(A_1)(x) \in P_1 \wedge \mathit{proj}(A_2)(x) \in F_2 \}.
\end{aligned}$$

Definition 5.17 (Projection). $p = \mathit{proj}(B)(p')$ is always defined and

$$\begin{aligned} A &= B \cap A' \\ S &= \mathit{proj}(B)(S'), \\ F &= \mathit{proj}(B)(F'), \end{aligned}$$

where *proj* is naturally extended to sets.

Definition 5.18 (Renaming). $p = \mathit{rename}(r)(p')$ is defined whenever $A' \subseteq \mathit{dom}(r)$. In that case

$$\begin{aligned} A &= r(A') \\ S &= \mathit{rename}(r)(S'), \\ F &= \mathit{rename}(r)(F'), \end{aligned}$$

where *rename* is naturally extended to sets.

The particular definition of parallel composition (def. 5.16) can be explained as follows. A trace is a success of the composite $p_1 \parallel p_2$ whenever the trace is a success of both p_1 and p_2. A trace is a failure of the composite if it is a possible traces of one component, and it is a failure of the other component. Note that if a trace is a failure of one component, but it is *not* a possible trace of the other component, the trace does not appear as a failure of the composite. This is because, in the interaction, the particular behavior that results in a failure will never be exercised, as it is ruled out by the other component.

The proofs of theorem 4.7 and theorem 4.8 can be adapted to show that two-set trace structure algebras are agent algebras, and normalizable agent algebras, respectively. We omit the details of these proof, and focus the rest of this section on defining an agent order and deriving the appropriate notions of conformance and mirror. As was already the case for one-set trace structures, the proposed ordering is such that the rename operator is not $\top$-monotonic. We remind the reader that we do so to simplify the notation and the presentation. The Locked Alphabet Algebra described in example 3.84 should be used as a guide to transform the model to a fully $\top$-monotonic agent algebra.

We have defined the order for one-set trace structures as the containment relationship between the trace sets. To put it another way, we are interpreting the trace set of a specification as the set of allowed behaviors. An agent is an implementation of the specification if it contains only allowed behaviors. The definition of order for two-set trace structures is similar. As for the one-set case, we require that a trace is a possible behavior of an implementation p only if it is also a possible behavior of the specification p', i.e., $P \subseteq P'$. At the same time, an implementation should operate correctly whenever its specification does. This implies that an implementation may fail only if its specification fails, which is equivalent to requiring that the failures of the implementation be contained in the set of failures of the specification, or in other words, that $F \subseteq F'$. The following definition generalizes these requirements to trace structures with arbitrary alphabet.

Definition 5.19 (Ordered Two-Set Trace Structure Algebra). Let $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ be a two-set trace structure algebra, and let $p = (A, S, F)$ and $p' = (A', S', F')$ be two trace structures. We say that p is less than or equal to p', written $p \preceq p'$, if and only if for all $x \in \mathcal{B}(A \cup A')$,

$$\begin{aligned} \textit{proj}(A)(x) \in P \quad &\Rightarrow \quad \textit{proj}(A')(x) \in P' \quad \text{and} \\ \textit{proj}(A)(x) \in F \quad &\Rightarrow \quad \textit{proj}(A')(x) \in F'. \end{aligned}$$

The proof that $\preceq$ is a preorder is similar to the proof of theorem 5.2.

Using inverse projection, we can express the same definition in the more traditional notation of trace inclusion.

Corollary 5.20. Let $p = (A, S, F)$ and $p' = (A', S', F')$ be trace structures. Then $p \preceq p'$ if and only if

$$\begin{aligned} \textit{proj}(A')(\textit{proj}(A')^{-1}(P)) \quad &\subseteq \quad P' \quad \text{and} \\ \textit{proj}(A')(\textit{proj}(A')^{-1}(F)) \quad &\subseteq \quad F'. \end{aligned}$$

As expected, when the alphabets of the agents are the same, the definition reduces to requiring that $P \subseteq P'$ and that $F \subseteq F'$. Note that the above definitions also imply that $S \subseteq S' \cup F'$. This simply indicates that an implementation may have a non-failing behavior where the specification had a failure.

It is easy to adapt the proof of theorem 5.4 to show that parallel composition is $\top$-monotonic relative to the defined agent ordering. We here state the result without proof.

Theorem 5.21. Parallel composition of two-set trace structures is $\top$-monotonic relative to $\preceq$.

5.2.1 Conformance and Mirrors

In order to derive a mirror function, we first need to characterize the order in terms of conformance relative to composition. For this purpose we must choose a conformance set G. Recall that the set of failures in a two-set trace structures represent the behaviors that are possible for an agent, but that denote an incorrect use of the agent on behalf of its environment. Recall also that a conformance set induces a notion of compatibility in the form of a compatibility set. In this case, it makes sense to consider two agents compatible whenever the two agents do not fail each other, or, in other words, when the set of failures of their parallel composition is empty. This notion is equivalent to the *failure-free* requirement of the asynchronous trace structures introduced by Dill [34]. The next result shows that two-set trace structures have a conformance order relative to composition with respect to this notion of compatibility. As for the one-set case, we assume that the trace structure algebra is complete.

Let $G = \{ p = (A, S, F) : F = \emptyset \}$. Clearly G is downward closed relative to $\preceq$.

Theorem 5.22. Two-Set trace structure algebras have a G-conformance order relative to composition.

Proof: Let $p = (A, S, F)$ and $p' = (A', S', F')$ be trace structures. We must show that $p \preceq p'$ if and only if for all trace structures q, $p' \parallel q \in G$ implies $p \parallel q \in G$. As usual, the forward direction follows from $\top$-monotonicity, since G is downward closed.

For the reverse direction, assume that for all trace structures q, $p' \parallel q \in G$ implies $p \parallel q \in G$. We must show that for all traces $x \in \mathcal{B}(A \cup A')$,

$$\begin{aligned} \textit{proj}(A)(x) \in P \quad &\Rightarrow \quad \textit{proj}(A')(x) \in P' \quad \text{and} \\ \textit{proj}(A)(x) \in F \quad &\Rightarrow \quad \textit{proj}(A')(x) \in F'. \end{aligned}$$

Let $q = (A', S' - F', \mathcal{B}(A') - P')$. The set of possible traces of q is $P'_q = (S' - F') \cup (\mathcal{B}(A') - (S' \cup F')) = \mathcal{B}(A') - F'$. Then

$$\begin{aligned} p' \parallel q &= (A', S' \cap (S' - F'), (F' \cap (\mathcal{B}(A') - F')) \cup ((\mathcal{B}(A') - P') \cap P')) \\ &= (A', S' - F', \emptyset), \end{aligned}$$

therefore $p' \parallel q \in G$. Hence, by assumption, also $p \parallel q \in G$. Therefore, since

$$\begin{aligned} p \parallel q = \; & (A \cup A', \{ x \in \mathcal{B}(A \cup A') : \textit{proj}(A)(x) \in S \wedge \textit{proj}(A')(x) \in (S' - F') \}, \\ & \{ x \in \mathcal{B}(A \cup A') : \textit{proj}(A)(x) \in F \wedge \textit{proj}(A')(x) \in \mathcal{B}(A') - F' \} \\ & \cup \{ x \in \mathcal{B}(A \cup A') : \textit{proj}(A)(x) \in P \wedge \textit{proj}(A')(x) \in \mathcal{B}(A') - P' \}) \end{aligned}$$

it must also be

$$\begin{aligned} \{ x \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(x) \in F \wedge \mathit{proj}(A')(x) \in \mathcal{B}(A') - F' \} &= \emptyset \quad \text{and} \\ \{ x \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(x) \in P \wedge \mathit{proj}(A')(x) \in \mathcal{B}(A') - P' \}) &= \emptyset \end{aligned}$$

or, equivalently, for all traces $x \in \mathcal{B}(A \cup A')$,

$$\begin{aligned} \mathit{proj}(A)(x) \in F &\Rightarrow \mathit{proj}(A')(x) \in F' \quad \text{and} \\ \mathit{proj}(A)(x) \in P &\Rightarrow \mathit{proj}(A')(x) \in P', \end{aligned}$$

which proves the result. □

Analogously to one-set trace structures, we now explore the form of the compatibility set and search for a greatest element. The greatest element is then used to construct a mirror function. As mentioned above, and given the particular conformance set G, two agents are compatible if they don't fail each other. Therefore, the failures of one should not be possible traces of the other. The following result expresses formally this intuitive notion.

Theorem 5.23 (Compatibility Set). . Let $p = (A, S, F)$ and $p' = (A', S', F')$ be trace structures. Then $p \in \mathit{cmp}(p')$ if and only if

$$F \subseteq \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')) \tag{5.1}$$

$$P \subseteq \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \tag{5.2}$$

Proof: Recall that the set of failures of the parallel composition $p \parallel p'$ is given by (definition 5.16)

$$\begin{aligned} F'' = {} & \{ x \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(x) \in F \wedge \mathit{proj}(A')(x) \in P' \} \cup \\ & \{ x \in \mathcal{B}(A \cup A') : \mathit{proj}(A)(x) \in P \wedge \mathit{proj}(A')(x) \in F' \}. \end{aligned}$$

Since G is composed of all and only the agents with an empty set of failures, p and p' are compatible if and only if $F'' = \emptyset$. Consequently, both terms in the union above must be empty. The proof can therefore be completed by a series of double implications similar to the one used in the proof of theorem 5.10. □

When the alphabets are the same, p is compatible with p' if and only if $F \subseteq \overline{P'}$ and $P \subseteq \overline{F'}$, where complementation includes only traces with the given alphabet. The greatest elements of the compatibility set are again to be found as those agents whose alphabet includes the alphabet of p', and whose sets of traces satisfy equation 5.1 and equation 5.2 when the containment relation is replaced by equality. The following result states this property in terms of the successes and the failure sets, rather than the possible and the failure traces.

Theorem 5.24 (Greatest Element). Let $p' = (A', S', F')$ be a trace structure. A trace structure $p = (A, S, F)$ such that $A' \subseteq A$ and

$$\begin{aligned} F &= \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')) \\ S &\supseteq \mathit{proj}(A)(\mathit{proj}(A)^{-1}(S')) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \\ S &\subseteq \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \end{aligned}$$

is a greatest element of $\mathit{cmp}(p')$.

Proof: We refer to the proof of theorem 5.11 to show that p is a greatest element if

$$\begin{aligned} F &= \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P')) \\ P &= \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \end{aligned}$$

To express the result in terms of S, recall that, by definition, $P = S \cup F$. Therefore S must be a subset of P and must contain at least all the elements of P that are not in F. Thus,

$$P - F \subseteq S \subseteq P.$$

Hence,

$$S \subseteq P = \mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')),$$

and

$$\begin{aligned}
S &\supseteq P - F \\
&\supseteq (\mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F'))) - (\mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P'))) \\
&\supseteq \mathcal{B}(A) - (\mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(P'))) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \\
&\quad \text{since } P' = S' \cup F' \\
&\supseteq \mathcal{B}(A) - (\mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(S' \cup F'))) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \\
&\quad \text{since, by lemma 5.7, projection and inverse projection distribute over set union} \\
&\supseteq \mathcal{B}(A) - (\mathcal{B}(A) - (\mathit{proj}(A)(\mathit{proj}(A)^{-1}(S')) \cup \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')))) - \\
&\qquad - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \\
&\supseteq \mathcal{B}(A) - (\mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(S')) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F'))) - \\
&\qquad - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \\
&\quad \text{since } \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \text{ is removed anyway} \\
&\supseteq \mathcal{B}(A) - (\mathcal{B}(A) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(S'))) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')) \\
&\supseteq \mathit{proj}(A)(\mathit{proj}(A)^{-1}(S'))) - \mathit{proj}(A)(\mathit{proj}(A)^{-1}(F')).
\end{aligned}$$

□

As for the single-set case, we choose among the greatest element the one that has the smaller alphabet. In addition, we select the greatest element with the smallest success set.

Corollary 5.25. $p = (A', S' - F', \mathcal{B}(A') - P')$ is a greatest element of $\mathit{cmp}(p')$.

Since two-set trace structure algebra have a G-conformance order relative to composition, and since the compatibility set of every trace structure has a greatest element, the algebra has a mirror function.

Corollary 5.26. Two-set trace structure algebras have a mirror function relative to G, and for all two-set trace structures $p = (A, S, F)$,

$$\mathit{mirror}(p) = (A, S - F, \mathcal{B}(A) - P).$$

Proof: The result follows directly from theorem 3.80. □

It is instructive to compare our results with the one obtained by Dill [34]. In his work, Dill defines two models based on trace structures for asynchronous circuits. There, a trace structure distinguishes between input and output signals. In addition, trace structures are defined so that they are *receptive* relative to their input signals. This implies that the trace structure must contain a possible trace for all possible sequences of input signals (the trace may indifferently be a success or a failure, as long as it is a possible trace). This ensures that a component, viewed as a trace structure, always exhibits *some* behavior in response to any possible input from the environment, whether or not the behavior is consistent with the intended use of the component. However, this also restricts the kind of trace structures that can be part of the trace structure algebra. In particular, only trace structures that are receptive are considered in the set of trace structures of the algebra.

In contrast, we have assumed so far that trace structure algebras are complete. With that assumption, we have shown that the agent order defined in definition 5.19 is a G-conformance order relative to composition, when G is taken to be the set of trace structures that are *failure-free* (theorem 5.22). As trace structures are removed from the trace structure algebra, and the notion of compatibility is left constant, the conformance order changes, as already demonstrated in theorem 3.114. More specifically, the conformance order in the subalgebra is *stronger* than the conformance order in the superalgebra, in the sense that if $p \preceq p'$ in the complete algebra, then $p \preceq p'$ in the incomplete algebra.

Dill defines the order for asynchronous trace structures to correspond exactly to a G-conformance order, where G is the set of failure-free agents. By our previous argument, the order in Dill's trace structure algebra is therefore stronger than the order defined in definition 5.19. Consequently, Dill's trace structure algebra is not closed under the mirror function derived in corollary 5.26 (if it were, then the orders would be the same, as shown in theorem 3.113). This is not surprising: if $p = (A, S, F)$ is a *receptive* trace structure, then $\mathit{mirror}(p) = (A, S - F, \mathcal{B}(A) - P)$ is not necessarily receptive, and therefore may not be included in the trace structure algebra.

Nonetheless, Dill's trace structures do have a mirror function. Dill proves this result by showing that each trace structure p in his model is order equivalent to a trace structure $\hat{p}$ in *canonical form*. A trace structure $\hat{p}$ in canonical form is essentially the most general representative of an equivalence class under the agent ordering, and is obtained by applying two operations, called *autofailure manifestation* and *failure exclusion*, that extend the sets of traces of a trace structure by adding those traces that do not affect the conformation order. The mirror of a trace structure in canonical form is the same as the mirror had the algebra been complete, and can therefore be computed as described in corollary 5.26. The mirror for a trace structure p that is *not* in canonical form is obtained by first transforming p into the order equivalent trace structure $\hat{p}$ in canonical form, and by then taking the mirror. Since p is order equivalent to $\hat{p}$, then $\mathit{mirror}(p) = \mathit{mirror}(\hat{p})$.

The construction employed in Dill's asynchronous trace structures is in fact a special case of a more general fact. As the order in the subalgebra is strengthened by the removal of agents, trace structures get organized into equivalence classes. The mirror for an equivalence class corresponds to the mirror of an upper bound of the equivalence class in the superalgebra. If the upper bound is also the greatest element of the equivalence class, then it is a canonical form. The next result makes these conditions precise. In the following we will consider an algebra $\mathcal{Q}'$ and a subalgebra $\mathcal{Q}$. Hence, the primed version of the operators and relations will refer to the operators and relations of $\mathcal{Q}'$, while the un-primed version to those of $\mathcal{Q}$.

Lemma 5.27. Let $\mathcal{Q}'$ be an agent algebra with a mirror function mirror' relative to a conformance set G'. Let $\mathcal{Q}$ be a subalgebra (def. 2.18) of $\mathcal{Q}'$, with a mirror function mirror relative to $G = G' \cap \mathcal{Q}.D$. Then for all agents $p \in \mathcal{Q}.D$,

1. $\mathit{cmp}(p) \subseteq \mathit{cmp}'(p)$,
2. if $\mathit{mirror}(p)$ is defined, then $\mathit{mirror}'(p)$ is defined,
3. if $\mathit{mirror}(p)$ is defined, then $\mathit{mirror}(p) \preceq' \mathit{mirror}'(p)$, and
4. if $\mathit{mirror}'(p) \approx' q$ and $q \in \mathcal{Q}.D$, then $\mathit{mirror}(p) \approx q$.

Proof: To show item 1, we show that if $q \in \mathit{cmp}(p)$, then $q \in \mathit{cmp}'(p)$. The proof consists of the following series of implications.

$q \in \mathit{cmp}(p)$

by definition 3.72,

$\Leftrightarrow \quad q \parallel p \in G$

since $\mathcal{Q}$ is a subalgebra of $\mathcal{Q}'$, $q \parallel p = q \parallel' p$, therefore

$\Rightarrow \quad q \parallel' p \in G$

since $G \subseteq G'$,

$\Rightarrow \quad q \parallel' p \in G'$

by definition 3.72,

$\Leftrightarrow \quad q \in \mathit{cmp}'(p).$

Therefore, $\mathit{cmp}(p) \subseteq \mathit{cmp}'(p)$.

Item 2 follows directly from item 1. In fact, if $\mathit{mirror}(p)$ is defined, then by definition 3.61, $\mathit{cmp}(p) \neq \emptyset$. Therefore, since $\mathit{cmp}(p) \subseteq \mathit{cmp}'(p)$, also $\mathit{cmp}'(p) \neq \emptyset$. Hence, again by definition 3.61, $\mathit{mirror}'(p)$ is defined.

We now show item 3. Assume that $\mathit{mirror}(p)$ is defined. Then, by item 2, $\mathit{mirror}'(p)$ is also defined. By theorem 3.80, $\mathit{mirror}(p) = \max(\mathit{cmp}(p))$, hence $\mathit{mirror}(p) \in \mathit{cmp}(p)$. By the argument above, then, $\mathit{mirror}(p) \in \mathit{cmp}'(p)$. Therefore, since also $\mathit{mirror}'(p) = \max'(\mathit{cmp}'(p))$, $\mathit{mirror}(p) \preceq' \mathit{mirror}'(p)$.

To show item 4, assume $q \in \mathcal{Q}.D$ and that $\mathit{mirror}'(p) \approx' q$. Consider the following series of implications that start from lemma 3.62.

$p \parallel' \mathit{mirror}'(p) \in G'$

since $\mathit{mirror}'(p) \approx' q$,

$\Rightarrow \quad p \parallel' q \in G'$

since both p and q are in $\mathcal{Q}$, and since $\mathcal{Q}$ is a subalgebra of $\mathcal{Q}'$,

$\Rightarrow \quad p \parallel q \in G'$

since $p \parallel q \in \mathcal{Q}.D$ and $G = G' \cap \mathcal{Q}.D$,

$\Rightarrow \quad p \parallel q \in G$

by definition 3.72,

$\Rightarrow \quad q \in \mathit{cmp}(p)$

since $\mathit{mirror}(p) = \max(\mathit{cmp}(p))$,

$\Rightarrow \quad q \preceq \mathit{mirror}(p)$.

On the other hand, by item 3, $\mathit{mirror}(p) \preceq' \mathit{mirror}'(p)$. Therefore, since $q \approx' \mathit{mirror}'(p)$, also $\mathit{mirror}(p) \preceq' q$. Hence, by theorem 3.114, $\mathit{mirror}(p) \preceq q$. Hence, since both $q \preceq \mathit{mirror}(p)$ and $\mathit{mirror}(p) \preceq q$, $\mathit{mirror}(p) \approx q$. □

Theorem 5.28 (Canonical Form). Let $\mathcal{Q}'$ be an agent algebra with a mirror function mirror' relative to a conformance set G'. Let $\mathcal{Q}$ be a subalgebra (def. 2.18) of $\mathcal{Q}'$, with a mirror function mirror relative to $G = G' \cap \mathcal{Q}.D$. Let $p \in \mathcal{Q}.D$ be an agent such that $\mathit{mirror}(p)$ is defined, and let $[p]$ denote the set of agents that are order equivalent to p in $\mathcal{Q}$. Then,

1. $\mathit{mirror}'(\mathit{mirror}(p))$ is an upper bound of $[p]$ in $\mathcal{Q}'$, and

2. $\mathit{mirror}'(p) \approx' \mathit{mirror}(p)$ if and only if $p = \max'([p])$ and $\mathit{mirror}'(\mathit{mirror}(p)) \approx' q$ and $q \in \mathcal{Q}.D$.

Proof: We first show that the relevant mirrors are defined. In fact, since $\mathit{mirror}(p)$ is defined, then, by item 2 of lemma 5.27, $\mathit{mirror}'(p)$ is also defined. In addition, by theorem 3.63, $\mathit{mirror}(\mathit{mirror}(p))$ is defined. Therefore, by item 2 of lemma 5.27, also $\mathit{mirror}'(\mathit{mirror}(p))$ is defined.

To prove item 1, we show that if $q \in [p]$, then $q \preceq' \mathit{mirror}'(\mathit{mirror}(p))$. The proof is composed of the following series of implications.

$q \in [p]$

by definition of $[p]$,

$\Rightarrow \quad q \preceq p$

since $\mathcal{Q}$ has a mirror function relative to G,

$\Leftrightarrow \quad q \parallel \mathit{mirror}(p) \in G$

since $\mathcal{Q}$ is a subalgebra of $\mathcal{Q}'$

$\Rightarrow \quad q \parallel' \mathit{mirror}(p) \in G$

since $G \subseteq G'$,

$\Rightarrow \quad q \parallel' \mathit{mirror}(p) \in G'$

since $\mathit{mirror}'(\mathit{mirror}(p))$ is defined, by theorem 3.63, theorem 3.66 and corollary 3.33,

$\Rightarrow \quad q \parallel' \mathit{mirror}'(\mathit{mirror}'(\mathit{mirror}(p))) \in G'$

since mirror' is a mirror function relative to G' for $\mathcal{Q}'$,

$\Leftrightarrow \quad q \preceq' \mathit{mirror}'(\mathit{mirror}(p))$.

To prove the forward direction of item 2, assume that $\mathit{mirror}'(p) \approx' \mathit{mirror}(p)$. To prove that $p = \max'([p])$, observe that, by item 1, $\mathit{mirror}'(\mathit{mirror}(p))$ is an upper bound of $[p]$. In addition, by theorem 3.66, $\mathit{mirror}'(\mathit{mirror}'(p)) \approx' p$. Also, since by hypothesis $\mathit{mirror}'(p) \approx' \mathit{mirror}(p)$, by corollary 3.70, $\mathit{mirror}'(\mathit{mirror}(p)) \approx' \mathit{mirror}'(\mathit{mirror}'(p))$. Therefore, since $\approx'$ is transitive, $\mathit{mirror}'(\mathit{mirror}(p)) \approx' p$. Hence, p is also an upper bound of $[p]$. But since by definition $p \in [p]$, $p = \max'([p])$. In addition, since $p \in \mathcal{Q}.D$, $\mathit{mirror}'(\mathit{mirror}(p)) \approx q = p$ and $q \in \mathcal{Q}.D$.

We now prove the backward direction of item 2. The proof is composed of the following series of implications.

$\mathit{mirror}'(\mathit{mirror}(p)) \approx' q \wedge q \in \mathcal{Q}.D$

by item 4 of lemma 5.27,

$\Rightarrow \quad \mathit{mirror}(\mathit{mirror}(p)) \approx q$

since, by theorem 3.66, $\mathit{mirror}(\mathit{mirror}(p)) \approx p$, by transitivity,

$\Rightarrow \quad q \approx p$

by definition of order equivalence,

$\Leftrightarrow \quad q \in [p]$

since, by hypothesis, $p = \max'([p])$,

$\Rightarrow \quad q \preceq' p$

since $\mathit{mirror}'(p)$ is defined, by definition 3.61,

$\Rightarrow \quad q \parallel' \mathit{mirror}'(p) \in G'$

since $q \approx' \mathit{mirror}'(\mathit{mirror}(p))$, by corollary 3.33,

$\Rightarrow \quad \mathit{mirror}'(\mathit{mirror}(p)) \parallel' \mathit{mirror}'(p) \in G'$

by definition 3.61,

$\Rightarrow \quad \mathit{mirror}'(p) \preceq' \mathit{mirror}(p)$

by item 3 of lemma 5.27, $\mathit{mirror}(p) \preceq' \mathit{mirror}'(p)$, therefore,

$\Rightarrow \quad \mathit{mirror}'(p) \approx' \mathit{mirror}(p)$.

$\square$

Theorem 5.28 shows that if the mirror of an agent p in the subalgebra is equal to (more precisely, order equivalent to in $\mathcal{Q}'$) its mirror in the superalgebra, then p is the greatest element in terms of the order in $\mathcal{Q}'$ of the equivalence class of the order equivalent agents in the subalgebra. In this case, p is the canonical form of $[p]$, and the mirror function of the superalgebra can be applied to any agent in $[p]$ after canonicalization. A sufficient condition that a subalgebra must satisfy to apply this technique is the following.

Theorem 5.29. Let $\mathcal{Q}'$ be an agent algebra with a mirror function *mirror*$'$ relative to a conformance set G'. Let $\mathcal{Q}$ be a subalgebra (def. 2.18) of $\mathcal{Q}'$, with a mirror function *mirror* relative to $G = G' \cap \mathcal{Q}.D$. Assume also that for all agents $p \in \mathcal{Q}.D$, $[p]$ has a greatest element in $\mathcal{Q}'$, and let $C = \{\, q \in \mathcal{Q}.D : \exists p \in \mathcal{Q}.D\ [q = \max'([p])]\}$ be the set of the greatest elements, If for all $q \in C$, $\mathit{mirror}'(q) \in \mathcal{Q}.D$, then $\mathit{mirror}_1(p) = \mathit{mirror}'(\max'([p]))$ is a mirror function relative to G for $\mathcal{Q}$.

Proof: By item 4 of lemma 5.27, for all $q \in C$, $\mathit{mirror}(q) \approx \mathit{mirror}'(q)$. In addition, by corollary 3.70, for all agents $p \in \mathcal{Q}.D$, $\mathit{mirror}(p) \approx \mathit{mirror}(\max'([p]))$, and therefore $\mathit{mirror}(p) \approx \mathit{mirror}_1(p)$. Hence, by corollary 3.81, $\mathit{mirror}_1(p)$ is a mirror function for $\mathcal{Q}$ relative to G. □

Analogously to one-set trace structure algebras, also two-set trace structure algebras are rectifiable. The next section illustrates the use of two-set trace structures and the local specification synthesis technique to solve the problem of protocol conversion.

5.3 Local Specifications and the Problem of Converter Synthesis

Two-Set trace structures are particularly well suited to modeling behavioral interfaces and protocols. The set of failure traces, in fact, states the conditions of correct operation of an agent. They can therefore be interpreted as assumptions that agents make relative to their environment. Two agents are compatible whenever they respect those assumptions, i.e., they do not engage in behaviors that make the other agent fail. Interface protocols can often be described in this way. The transactions that do not comply with the protocol specification are considered illegal, and therefore result in an incorrect operation of the agent that implements the protocol.

In this section we present an example of use of such interface specifications, together with an application of the local specification synthesis technique to deriving a converter between two incompatible protocols. We first set up and solve the conversion problem for send-receive protocols, where the *sender* and the *receiver* are specified as automata. A third automaton, the *requirement*, specifies constraints on the converter, such as buffer size and the possibility of message loss. We then repeat and extend the example using a two-set synchronous trace structure model and the local specification synthesis technique.

5.3.1 Automata-based Solution

We illustrate our approach to protocol conversion by way of an example, which is an extension (and in some sense, also a simplification) of the one found in [74]. A producer and a consumer component wish to communicate some complex data across a communication channel. They both partition the data into two parts. The interface of the producer is defined so that it can wait an unbounded amount of time between the two parts. Because the sender has only outputs, this is equivalent to saying that the interface does not guarantee to its environment that the second part will follow the first within a fixed finite time. On the other hand, the interface of the consumer is defined so that it requires that once the first part has been received, the second is also received during the state transition that immediately follows the first. Because the receiver has only inputs, this specification corresponds to an assumption that the receiver makes on the set of possible environments that it can work with. Clearly, the two protocols are incompatible. Below, we illustrate how to synthesize a converter that enables them to communicate correctly. In particular, the guarantees of the sender are not sufficient to prove that the assumptions of the receiver are always satisfied. Thus a direct composition would result in a possible violation of the protocols. Because no external environment can prevent this violation (the system has no inputs after the composition), an intermediate converter must be inserted to make the communication possible.

The two protocols can be represented by the automata shown in figure 5.1. There, the

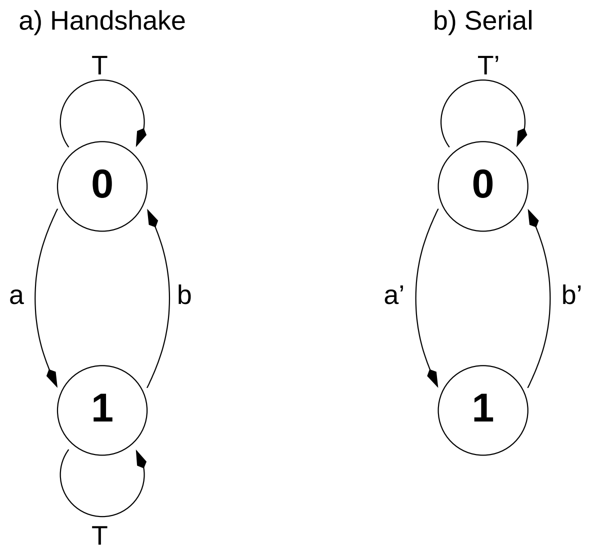

Figure 5.1: Handshake and serial protocols

symbols a and b (and their primed counterparts) are used to denote the first and the second part of the data, respectively. The symbol $\top$ denotes instead the absence or irrelevance of the data. In other words, it acts as a don't care.

Figure 5.1.a shows the producer protocol. The self loop in state 1 indicates that the transmission of a can be followed by any number of cycles before b is also transmitted. We call this protocol *handshake* because it could negotiate when to send the second part of the data. After b is transmitted, the protocol returns to its initial state, and is ready for a new transaction. The ability to handle multiple transactions is also an extension of our previous work.

Figure 5.1.b shows the receiver protocol. Here state 1 does not have a self loop. Hence, once a has been received, the protocol assumes that b is transmitted in the cycle that immediately follows. This protocol is called *serial* because it requires a and b to be transferred back-to-back. Similarly to the sender protocol, once b is received the automaton returns to its initial state, ready for a new transaction.

We have used non-primed and primed versions of the symbols in the alphabet of the automata to emphasize that the two sets of signals are different and should be connected through a converter. It is the specification (below) that defines the exact relationships that must hold between the elements of the two alphabets. Note that in the definition of the two protocols nothing relates the quantities of one (a and b) to those of the other (a' and b'). The symbol a could represent the toggling of a signal, or could symbolically represent the value of, for instance, an 8-bit variable. It is only in the interpretation of the designer that a and a' actually hold the same value. The specification that we are about to describe does not enforce this interpretation, but merely defines the (partial) order in which the symbols can be presented to and produced by the converter. It is possible to explicitly represent the values passed; this is necessary when the behavior of the protocols depends on the data, or when the data values provided by one protocol must be modified (translated) before being forwarded to the other protocol. The synthesis of a protocol converter would then yield a converter capable of both translating data values, and of modifying their timing and order. However, the price to pay for the ability to synthesize data translators is the state explosion in the automata to describe the interfaces and the specification.

Observe also that if a and b are symbolic representation of data, some other means must be available in the implementation to distinguish when the actual data corresponds to a or to b. At this level of the description we don’t need to be specific; examples of methods include toggling bits, or using data fields to specify message types.

What constitutes a correct transaction? Or in other words, what properties do we want the communication to have? In the context of this particular example the answer seems straightforward. Nonetheless, different criteria could be enforced depending on the application. Each criterion is embodied by a different specification.

One example of a specification is shown in figure 5.2. The alphabet of the automaton is derived from the Cartesian product of the alphabets of the two protocols for which we want to build a converter. This specification states that no symbols should be discarded or duplicated by the converter, and symbols must be delivered in the same order in which they were received;

moreover, the converter can store at most one undelivered symbol at any time. The three states in

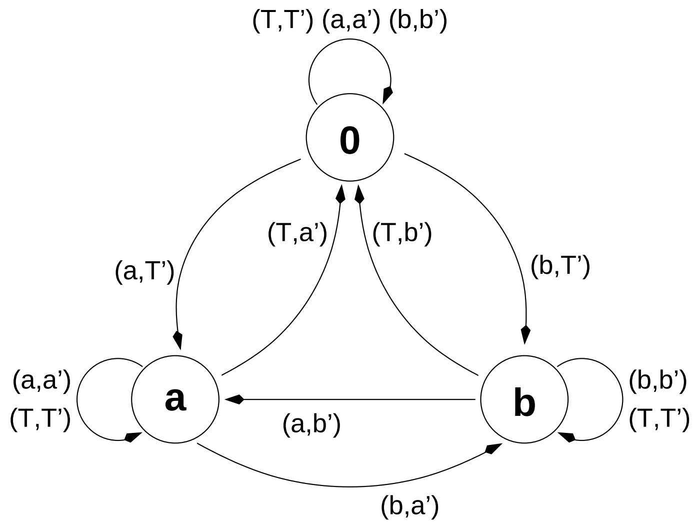

Figure 5.2: Specification automaton

the specification correspond to three distinct cases.

- State **0** denotes the case in which all received symbols have been delivered (or that no symbol has been received, yet).
- State **a** denotes the case in which symbol a has been received, but it hasn't been output yet.
- Similarly, state **b** denotes the case in which symbol b has been received, but not yet output.

Note that this specification is not concerned with the particular form of the protocols being considered (or else it would itself function as the converter); for example, it does not require that the symbols a or b be received in any particular order (other than the one in which they are sent). On the other hand, the specification makes precise what the converter can, and cannot do, ruling out for instance converters that simply discard all input symbols from one protocol, never producing any output for the destination protocol. In fact, the specification admits the case in which a and b are transferred in the reversed order. It also does not enforce that a and b always occur in pairs, and admits a sequence of a's without intervening b's (or vice versa). The specification merely asserts that a' should occur no earlier than a (an ordering relation), and that d must occur whenever a new a or b occurs. In fact, we can view the specification as an observer that specifies what *can* happen (a transition on some symbol is available) and what *should not* happen (a transition on some symbol is not available). As such, it is possible to decompose the specification into several automata, each one of which specifies a particular property that the synthesized converter should exhibit. This is similar to the monitor-based property specification proposed by Shimizu et al. [87] for the verification of communication protocols. In our work, however, we use the monitors to drive the synthesis so that the converter is guaranteed to exhibit the desired properties (correct-by-construction).

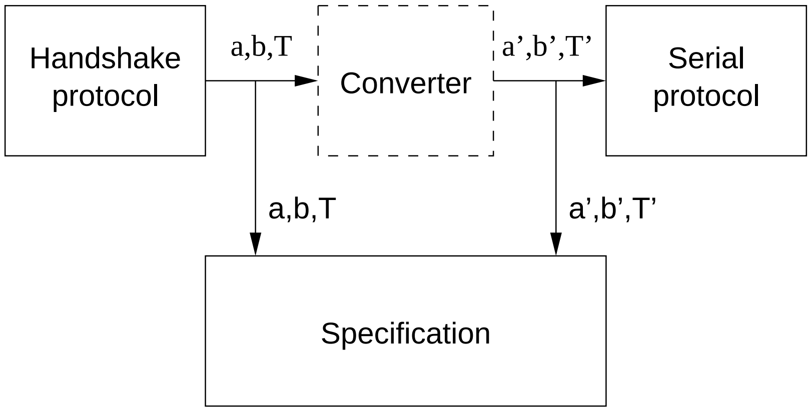

Figure 5.3: Inputs and outputs of protocols, specification, and converter.

A high-level view of the relationship between the protocols and the specification is presented in figure 5.3. The protocol *handshake* produces outputs a and b, the protocol *serial* accepts inputs a' and b'. The specification accepts inputs a, b, a', b', and acts as a global observer that states what properties the converter should have. Once we compose the two protocols and the specification, we obtain a system with outputs a, b, and inputs a', b' (figure 5.3). The converter will have inputs and outputs exchanged: a and b are the converter inputs, and a', b' its outputs.

The synthesis of the converter begins with the composition (product machine) of the two protocols, shown in figure 5.4. Here the direction of the signals is reversed: the inputs to the protocols become the outputs of the converter, and vice versa. This composition is also a specification for the converter, since on both sides the converter must comply with the protocols that are being interfaced. However *this* specification does not have the notion of synchronization (partial order, or causality constraint) that the specification discussed above dictates.

We can ensure that the converter satisfies both specifications by taking the converter to be composition of the product machine with the specification, and by removing transitions that violate either protocol or the correctness specification. Figure 5.5 through figure 5.7 explicitly show the steps that we go through to compute this product. The position of the state reflects the position of the corresponding state in the protocol composition, while the label inside the state represents the

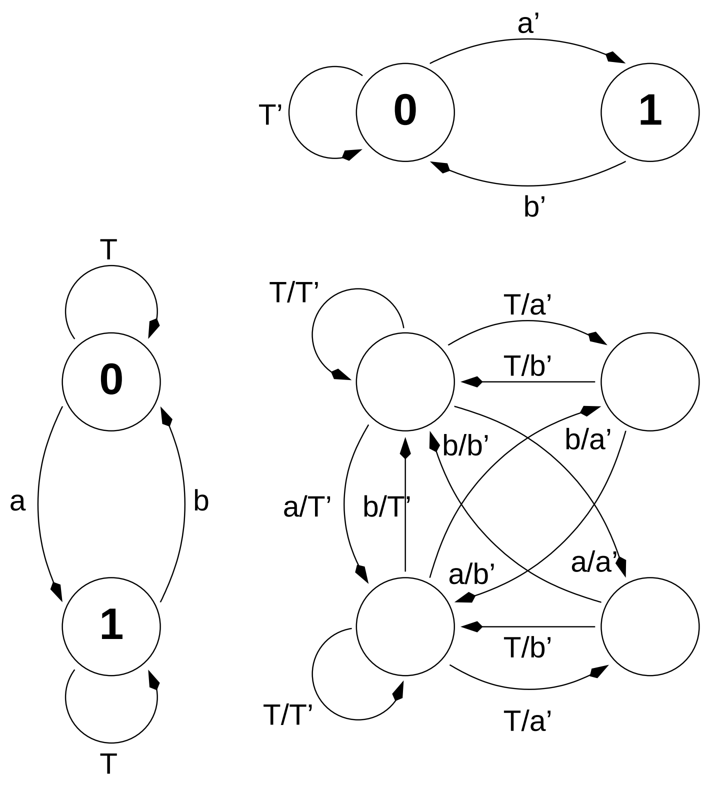

Figure 5.4: Composition between handshake and serial

corresponding state in the specification. Observe that the bottom-right state is reached when the specification goes back to state **0**. This procedure corresponds to the synthesis algorithm proposed in our previous work [74]. The approach here is however fundamentally different: the illegal states are defined by the specification, and not by the particular algorithm employed.

The initial step is shown in figure 5.5. The composition with the specification makes the transitions depicted in dotted line illegal (if taken, the specification would be violated). However, transitions can be removed from the composition only if doing so does not result in an assumption on the behavior of the sender. In figure 5.5, the transition labeled $\top/d$ leaving state **0** can be removed because the machine can still respond to a $\top$ input by taking the self loop, which is legal. The same applies to the transition labeled $b/\top'$ leaving state **a** which is replaced by the transition labeled b/a'. However, removing the transition labeled $\top/b'$ leaving the bottom-right state would make the machine unreceptive to input $\top$. Equivalently, the converter is imposing an assumption on the producer that $\top$ will not occur in that state. Because this assumption is not verified, and because we can’t change the producer, we can only avoid the problem by making the bottom-right

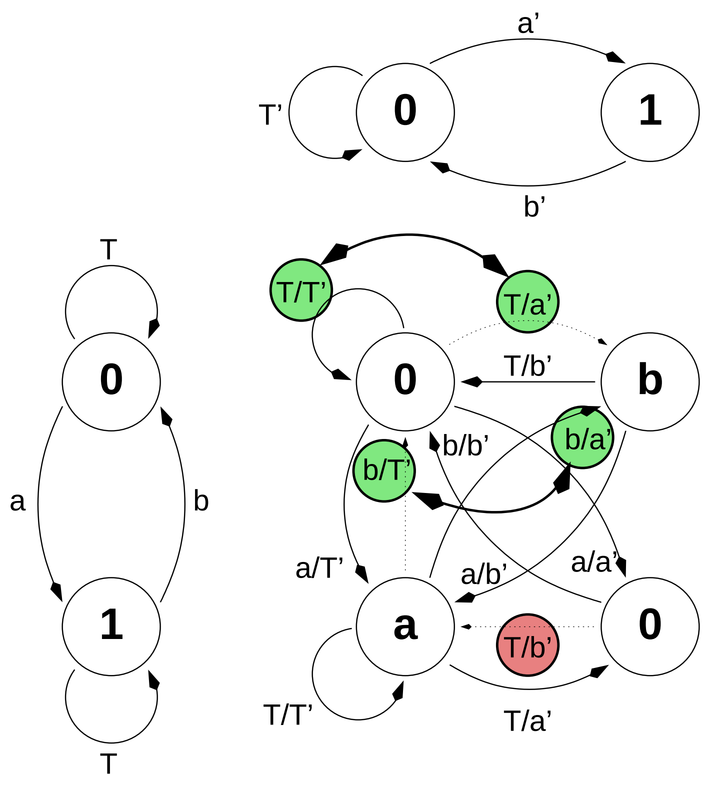

Figure 5.5: Converter computation, phase 1

state unreachable, and remove it from the composition.

The result is shown in figure 5.6. The transitions that are left dangling because of the removal of the state should also be removed, and are now shown in dotted lines. The same reasoning as before applies, and we can only remove transitions that can be replaced by others with the same input symbol. In this case, all illegal transitions can be safely removed.

The resulting machine shown in figure 5.7 has now no illegal transitions. This machine complies both with the specification and with the two protocols, and thus represents the correct conversion (correct relative to the specification). Notice how the machine at first stores the symbol a without sending it (transition $a/\top'$). Then, when b is received, the machine sends a', immediately followed in the next cycle by b', as required by the serial protocol.

5.3.2 Trace-Based Solution

The solution to the protocol conversion problem as described in the previous section requires that we develop a trace-based model of a synchronous system. The model that we have in

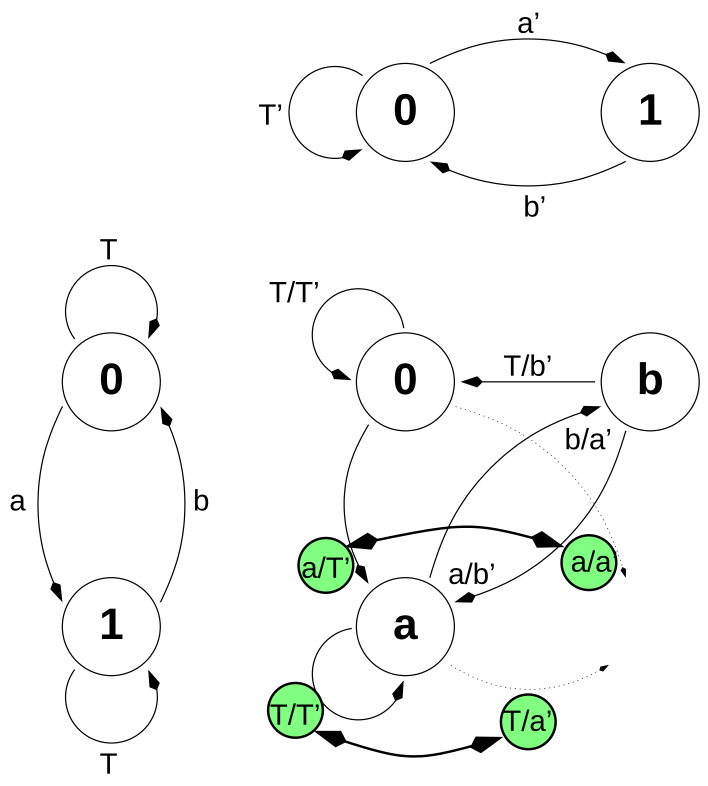

Figure 5.6: Converter computation, phase 2

mind is essentially identical to the synchronous models proposed by Burch [12] and Wolf [96]. For our simple case, an individual execution of an agent (a trace) is a sequence of actions from the alphabet $\mathcal{A} = \{\top, a, b, a', b'\}$ of signals, where $\top$ denotes the absence of an action. This is similar to the form of the traces of the CSP model described in subsection 4.3.3. The renaming operator is also defined similarly. Projection however is different. In the CSP case, projection shrinks the sequence as it removes elements. This would be appropriate for an asynchronous model. Here, instead, we need to retain the information on the cycle count. Therefore, projection simply replaces the action to be hidden by the special value $\top$. For instance,

$$\mathit{proj}(\{a\})(\langle a, b, a, \top, b, a, b, b, a, \ldots\rangle) = \langle a, \top, a, \top, \top, a, \top, \top, a, \ldots\rangle.$$

Parallel composition is defined as usual (def. 5.16). Because the length of the sequence is retained, parallel composition results in a lock step execution of the agents.

It is easy to represent the two protocols and the correctness specification as two-set trace structures constructed from the trace algebra just described. We can represent the sets of traces using the automata of figure 5.1 and figure 5.2. Note that now the specification consists of only

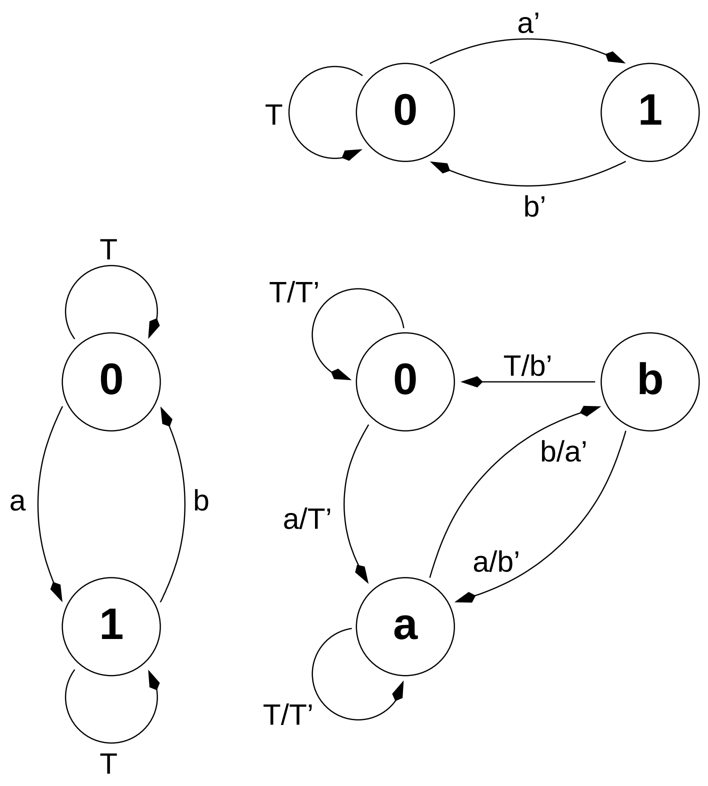

Figure 5.7: Converter computation, phase 3

outputs, since even the inputs to the receiver protocols are converted into outputs once the final system that includes the converter is constructed. Therefore, we do not need to add any failure to the specification, nor to the sender protocol, which also consists of just outputs. The receiver, on the other hand, must be augmented with a state representing the failure traces. A transition to this additional state is taken from each state on all the inputs for which an action is not already present.

We are interested in a solution with a specific alphabet. Therefore we adopt the simple agent ordering that requires the alphabets of the agents being compared to be the same. In this case, if S is the sender protocol, R the receiver, C the converter and P the specification, we may compute the converter by setting up the following local specification synthesis problem:

$$S \parallel R \parallel C \preceq P.$$

The solution is given by theorem 3.120, as

$$C \preceq \mathit{mirror}(S \parallel R \parallel \mathit{mirror}(P)).$$

Note that projections are not needed in this case, since after the composition of S and R the alphabet is always $\mathcal{A} = \{\top, a, b, \top', a', b'\}$, which is also the alphabet of C. Note also that the mirror switches inputs with outputs, so that the parallel composition in the solution is well defined.

We require that our agents be receptive, that is that they have a possible transition on every input. We are therefore working in a subalgebra of the complete trace structure algebra. The mirror of an agent is thus computed by first determining the greatest element of its equivalence class (the canonical form) according to the results of theorem 5.28. This can be achieved by applying the operations of autofailure manifestation and failure exclusion, similarly to the the synchronous trace structure algebra of Wolf [96]. A state is an autofailure if all its outgoing transitions are failures. Failure exclusion, instead, results in the removal of successful transitions whenever they are matched by a corresponding failing transition on the same input. After these operation, the mirror can be computed by applying the formula of corollary 5.25. Because language complementation is involved, this is most easily done by first making the automaton deterministic. For a deterministic and receptive agent the mirror can be computed by replacing for each state the existing outgoing failing transitions with transitions whose input symbol is not already handled by some other outgoing transition.

When doing so in the example above, we obtain exactly the result depicted in figure 5.7, with additional failing transitions that stand to represent the flexibility in the implementation. In particular, the state labeled **0** in figure 5.7 has failing transitions on input b, the state labeled **1** on input a and the state labeled **2** on input b.

5.3.3 End to End Specification

A potentially better approach to protocol conversion consists of changing the topology of the local specification problem, by providing a global specification that extends end to end from the sender to the receiver, as shown in figure 5.8. The global specification in this case may be limited to talking about the behavior of the communication channel as a whole, and would be independent of the particular signals employed internally by each protocol. In addition, in a scenario where the sender and the receiver function as layers of two communicating protocol stacks, the end to end behavior is likely to be more abstract, and therefore simpler to specify, than the inner information exchange.

We illustrate this case by modifying the previous example. In order to change the topology, the sender and receiver protocols must be modified to include inputs from (for the sender) and outputs to (for the receiver) the environment. This is necessary to let the protocols receive and

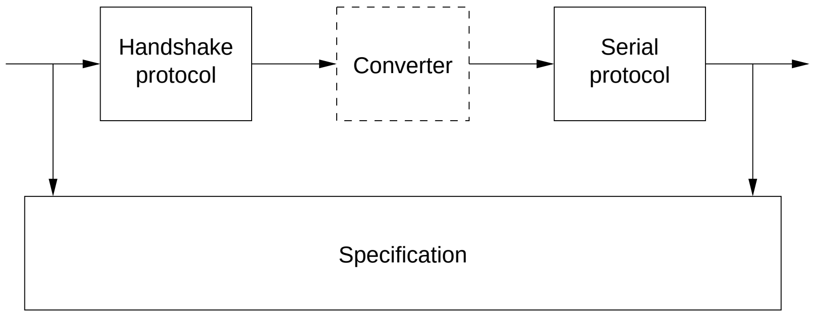

Figure 5.8: End to end specification

deliver the data transmitted over the communication channel, and to make it possible to specify a global behavior. In addition to adding connections to the environment, in this example we also explicitly model the data. Thus, unlike the previous example where the specification only required that a certain ordering relationship on the data be satisfied, we can here express true correctness by specifying that if a value is input to the system, the *same* value is output by the system at the end of the transaction. Since the size of the state space of the automata increases exponentially with the size of the data, we will limit the example to the communication of a two-bit integer value. To make the example more interesting, we modify the protocols so that the sender serializes the least significant bit first, while the receiver expects the most significant bit first. In this case, the converter will also need to reorder the sequence of the bits received from the sender.

All signals in the system are binary valued. The protocols are simple variations of the ones depicted in figure 5.1. The inputs to the sender protocol include a signal $\mathbf{ft}$ that is set to 1 when data is available, and two additional signals that encode the two-bit integer to be transmitted. The outputs also include a signal $\mathbf{st}$ that clocks the serial delivery of the data, and one signal $\mathbf{sd}$ for the data itself. The sender protocol is depicted in figure 5.9. We adopt the convention that a signal is true in the label of a transition when it appears with its original name, and it is false when its name is preceded by an $\mathbf{n}$. Hence, for example, $\mathbf{ft}$ implies that $\mathbf{ft} = 1$, and $\mathbf{nft}$ that $\mathbf{ft} = 0$. The shaded state labeled **F** in the automaton accepts the failure traces, while the rest of the states accept the successful traces. Note that the protocol assumes that the environment refrains from sending new data while in the middle of a transfer. In addition, the protocol may wish to delay the transmission of the second bit of the data for as many cycles as desired.

Similarly, the receiver protocol has inputs $\mathbf{rt}$ and $\mathbf{rd}$, where $\mathbf{rt}$ is used to synchronize the start of the serial transfer with the other protocol; the output $\mathbf{tt}$ finally informs the environment

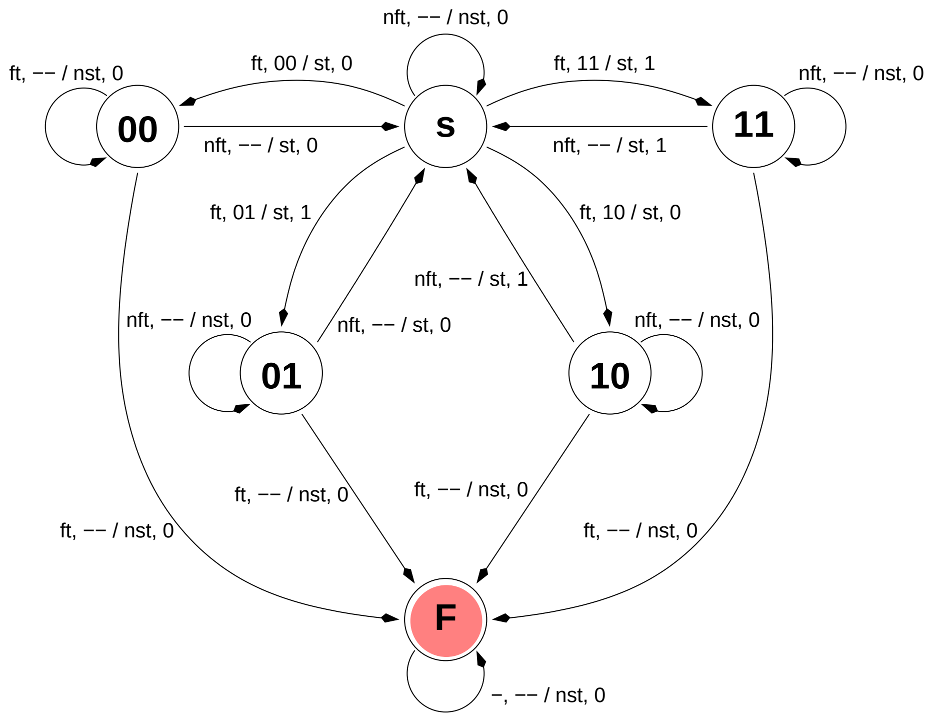

Figure 5.9: The sender protocol

when new data is available. The receiver protocol is depicted in figure 5.10. The receiver fails if the second bit of the data is not received within the clock cycle that follows the delivery of the first bit.

The automaton for the global specification is shown in figure 5.11. The global specification has the same inputs as the sender protocol, and the same outputs as the receiver protocol. A trace is successful if a certain value is received on the sender side, and the same value is emitted immediately or after an arbitrary delay on the receiver side. Analogously to the sender protocol, the specification fails if a new data value is received while the old value has not been delivered yet.

Following the same notation as the previous example, the solution to the conversion problem can be stated as

$$C \preceq \mathit{mirror}(\mathit{proj}(\{\mathbf{st}, \mathbf{sd}, \mathbf{rt}, \mathbf{rd}\})(S \parallel R \parallel \mathit{mirror}(P))).$$

The projection is now essential to scope down the solution to only the signals that concern the conversion algorithm. The agents of the algebra are again receptive, therefore similar considerations as those expressed before for the computation of the mirror in a subalgebra apply. In particular, autofailure manifestation and failure exclusion is applied before computing the mirror in order to reach the greatest element of the equivalence class of order equivalent agents. The agent is also

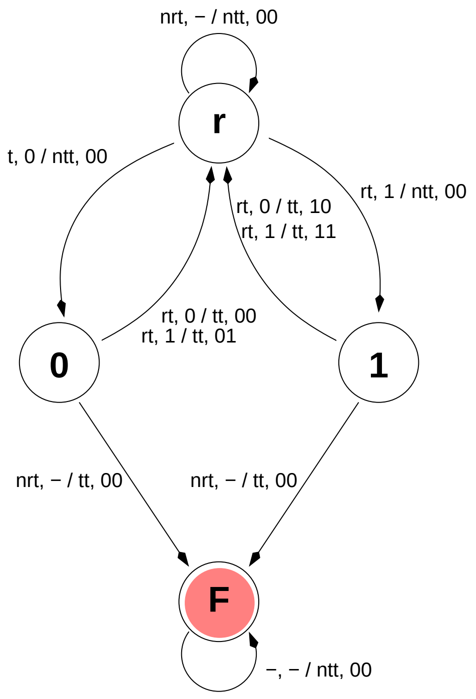

Figure 5.10: The receiver protocol

made deterministic if necessary. The result of the computation is shown in figure 5.12, where, for readability, the transitions that lead to the failure states have been displayed in dotted lines. The form of the result is essentially identical to that of figure 5.7. Note how the converter switches the position of the most and the least significant bit of the data during the transfer. In this way the converter makes sure that the correct data is transferred from one end to the other. Note, however, that the new global specification (figure 5.11) had no knowledge whatsoever of how the protocols were supposed to exchange data. Failure traces again express the flexibility in the implementation, and at the same time represent assumptions on the environment. These assumption are guaranteed to be satisfied (modulo a failure in the global specification), since the environment is composed of the sender and the receiver protocol, which are known variables in the system.

The solution excludes certain states that lead to a deadlock situation. This is in fact an important side effect of our specific choice of synchronous model, and has to do with the possibility of combinational loops that may arise as a result of a parallel composition. When this is the case, the mirror of an otherwise receptive agent may not be receptive. This is because it is perfectly admissible in the model to avoid a failure by witholding an input, i.e., by constraining the environment not to generate an input. But since the environment is not constrained, this can only be achieved by

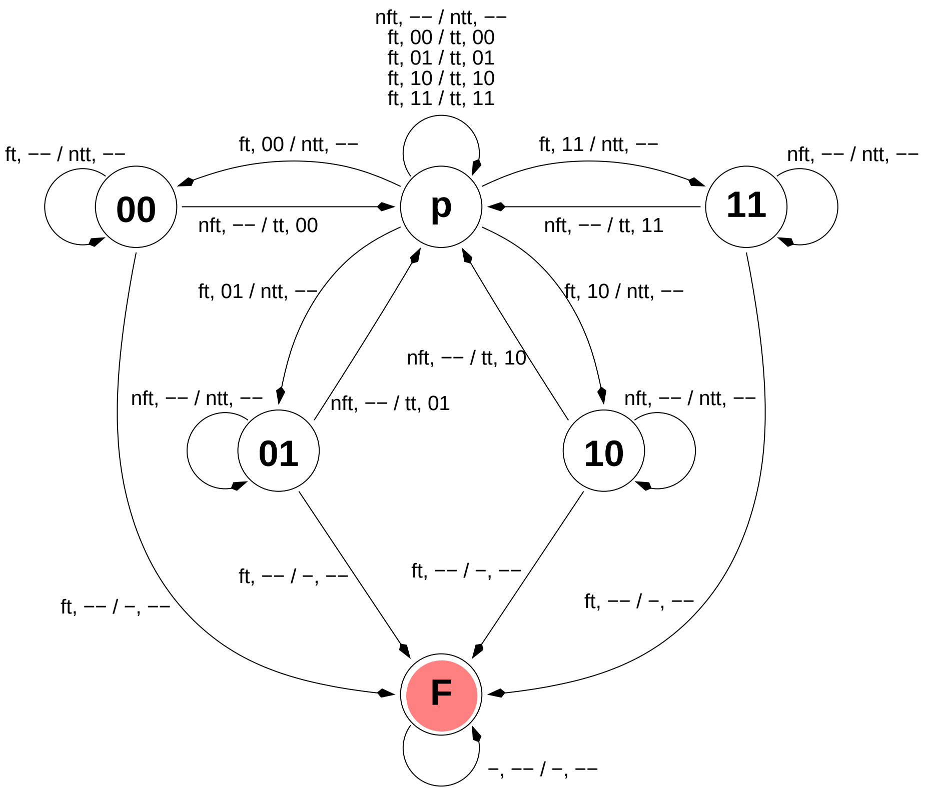

Figure 5.11: The global specification

“stopping time” before reaching the deadlock state. Since this would be infeasible in any reasonable physical model, we consider deadlock states tantomount to an autofailure, and remove them from the final result. This problem can be solved by employing a synchronous model that deals with combinational loops directly. This is an aspect of the implementation that has been extensively studied by Wolf [96], who proposes to use a three-valued model that includes the usual binary values 0 and 1, and one additional value to represent the oscillating, or unknown, behavior that results from the combinational loops. Exploring the use of this model in the context of protocol specification and converter synthesis is part of our future work.

A similar condition may occur when an agent tries to “guess” the future, by speculating the sequence of inputs that will be received in the following steps. If the sequence is not received, the agent will find itself in a deadlock situation, unable to roll back to a consistent state. This is again admissible in our model, but would be ruled out if the right notion of receptiveness were adopted. These states and transitions are also pruned as autofailures.

We have implemented this trace structure algebra in a prototype application in approxi-

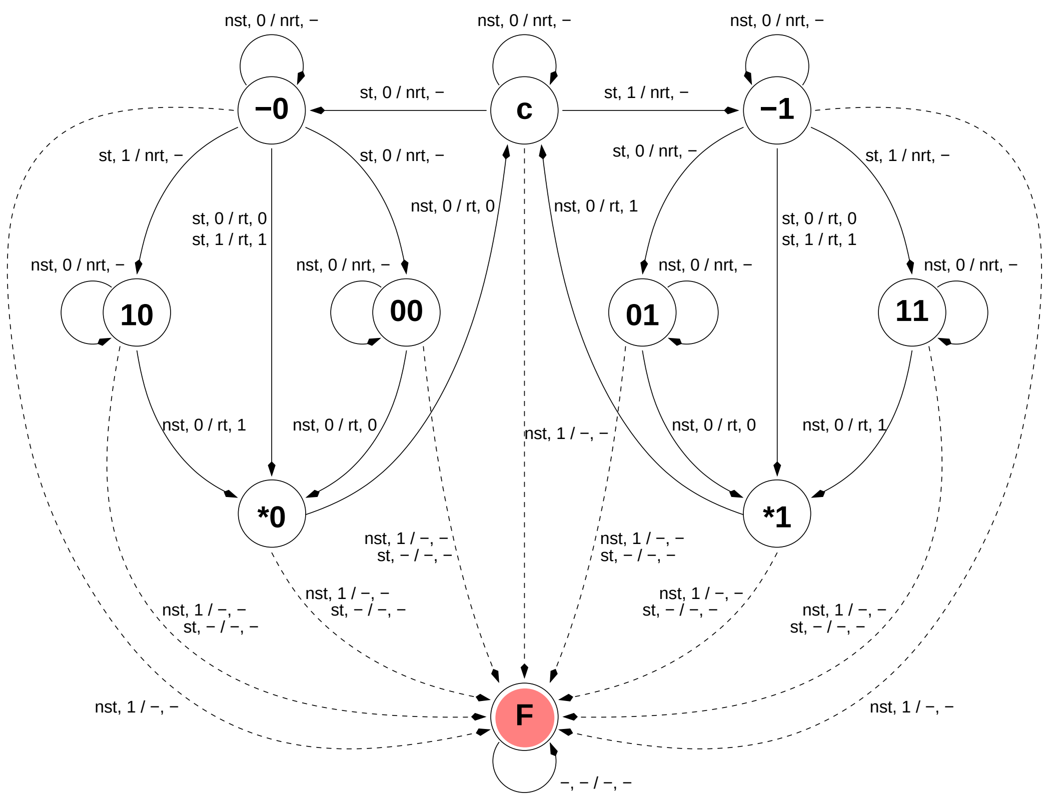

Figure 5.12: The local converter specification

mately 2400 lines of C++ code. In the code, we explicitly represent the states and their transitions, while the formulas in the transitions are represented implicitly using BDDs (obtained from a separate package). This representation obviously suffer from the problem of state explosion. This is particularly true when the value of the data is explicitly handled by the protocols and the specification, as already discussed. A better solution can be achieved if the state space and the transition relation are also represented implicitly using BDDs. Note, in fact, that most of the time the data is simply stored and passed on by a protocol specification and is therefore not involved in deciding its *control flow*. The symmetries that result can therefore likely be exploited to simplify the problem and make the computation of the solution more efficient.

Note that the converter that we obtain is non-deterministic and could take paths that are "slower" than one could expect them to be. This is evident in particular for the states labeled **-0** and **-1** which can react to the arrival of the second piece of data by doing nothing, or by transitioning directly to the states ***0** and ***1**, respectively, while delivering the first part of the data. This is because our procedure derives the full flexbility of the implementation, and the specification depicted in figure 5.11 does not mandate that the data be transferred as soon as possible. A “faster” implementation can be obtained by selecting the appropriate paths whenever a choice is available, as shown in figure 5.13. In this case, the converter starts the transfer in the same clock cycle in

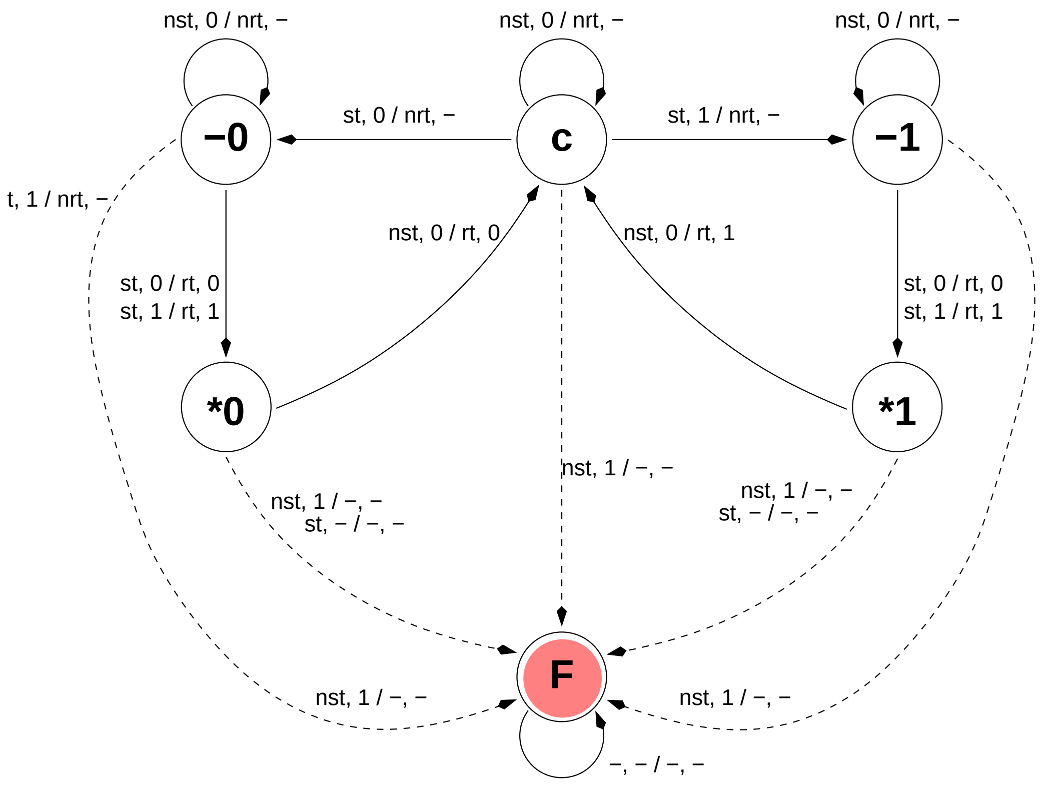

Figure 5.13: The optimized converter

which the last bit from the sender protocol is received. Other choices as also possible. In general, a fully deterministic converter can be obtained by optimizing certain parameters, such as the number of states or the latency of the computation. More sophisticated techniques might also try to enforce properties that were not included already in the global specification.

Chapter 6

Conclusions

6.1 Summary

In this thesis, we have described general techniques for constructing semantic domains for different models of computation at different levels of abstraction. Each semantic domain is constructed as an algebra, called an agent algebra, that includes a set of agents and three operators on agents. The operators, called renaming, projection and parallel composition, correspond to the operation of instantiation, scoping and composition of the model of computation.

The approach that we have taken in building our framework is axiomatic. The semantic domains that fit in our framework can take any form. Our only requirement is that the operators on agents satisfy certain properties (axioms) that formalize their intuitive interpretation. The results that we derive in the framework depend only on the axioms, and they therefore apply to all the semantic domains that satisfy the requirements.

In particular we have considered semantic domains that are ordered by a relation of refinement. We have introduced the notion of a $\top$-monotonic function as an extension of monotonicity to partial functions, that is justified by interpreting the refinement relation as a substitutability ordering. We have then studied relationships between different semantic domains in the form of conservative approximations, i.e., pairs of functions that preserve refinement verification results across different semantic domains. We have also shown that conservative approximations are more general than the traditional notion of abstract interpretation.

We have then characterized the order of a semantic domain in terms of a conformance relationship, which is based on substitutability of agents under every context. We have also shown under what conditions a single context, called the mirror, can be used to characterize the conformance relationship. We have used the mirror to algebraically solve the problem of the synthesis of the local specification, a useful technique that can be applied in many different circumstances.

6.2 Future Work

In this work we have laid the semantic foundations, and derived general properties and results, for studying the relationships and the verification and synthesis techniques of different semantic domains. Here we outline possible directions for further research and for applications of our framework.

6.2.1 Extensions to the Theory

In this dissertation we have presented the basic theory of agent algebras and of the subclass of trace-based agent algebras. While we have provided exact or sufficient characterizations of many of the concepts involved in the design of semantic domains, several extensions to the theory are still possible.

We have introduced a preorder on the agents to model a relationship of substitutability. One aspect that is of particular interest is to study the properties of the semantic domain when the order gives rise to a lattice structure. In this case, certain upper bounds and greatest elements that are required for constructions that involve mirror functions and/or conservative approximations are guaranteed to exist. In particular, we have shown in section 3.5 that the mirror, when it exists, can be used as a complementation operator to construct a conservative approximation from a Galois connection. The ramifications of this technique are yet to be explored.

We have also shown in theorem 2.59 that the inverse of a conservative approximation always corresponds to the greatest and the least element of the equivalence classes induced by the upper and lower bound of a conservative approximation. If the inverse is not defined, these elements do not exist. It would be interesting to explore how a semantic domain should be “completed” to add the necessary elements that make the inverse of a conservative approximation always defined.

In the context of trace-based agent algebras we have introduced the concept of complete and partial traces, which is inspired by the work of Dill [34] and Burch [12]. The axioms introduced by Burch should be reconsidered in light of our new results on trace-based agent algebras. In particular, we are interested in deriving sufficient conditions for the set of partial traces to exactly characterize the set of complete traces of an algebra. This is interesting when the verification problem is limited to considering only safety properties. Our current work includes exploring these conditions. In particular we have characterized complete traces as the limit of directed sets of partial traces and defined the concept of a basis set of traces that is similar to the one introduced by Stoy [89]. The main result of this characterization shows that the set of partial traces completely characterizes the algebra if and only if the algebra is closed under limits and the limit of the set of prefixes of a complete trace is equal to the complete trace. Considerations of space have precluded us from presenting these results in this work. We are currently exploring sufficient conditions that are simpler to check, and that guarantee the desired characterization.

6.2.2 Finitely Representable Models

The work in this dissertation is based on a denotational representation of the semantic domains that is convenient for manual, formal reasoning. In doing so, we have de-emphasized finitely representable, or executable models. The applications of the techniques described in this thesis, however, will likely have to be supported by a finite representation of the models for which the operators of the algebra are computable. This is, for example, essential to carry out simulations, or to solve problems of refinement verification. In our work we have concentrated on problems that relate to the correctness of the technique. A finite representation also raises the question of the efficiency of a certain computation. We believe that the upfront theoretical work is useful in two ways. First, once a model of computation is shown to fit in our framework, the results are guaranteed to be correct no matter what the implementation looks like. The model designer can therefore concentrate on improving the efficiency of the implementation. Second, the conditions that are sufficient and (often) necessary to apply a certain result may guide the model implementer and suggest ways of increasing the efficiency of the implementation.

Among the models that we are most interested in are synchronous and asynchronous models at different levels of abstraction. The relationship between these models could shed light on the properties of GALS systems and on their correct deployment. In particular we intend to adapt the work of Benveniste et al. [11] by deriving conservative approximations between synchronous and asynchronous models. These approximations are interesting, because it can be shown that their inverses are not embeddings. In addition, we plan to use the more expressive synchronous model proposed by Wolf [96] that correctly handles combinatorial loops, and, therefore hierarchical designs.

In the context of trace-based agent algebras, finitely representable models often rely on an implicit representation in the form of a machine that distinguishes the traces that are part of an agent from those that are not. Automata are one example of such recognizers for the special case of languages over a set of symbols. Partial behaviors, and the operation of concatenation, could be used to extend the automata model to recognize arbitrary traces. To do so, it is sufficient to partition the set of partial traces of an agent into equivalence classes. Each equivalence class consists of the set of traces that share the same suffixes within a particular agent. Thus, an equivalence class encodes all the information necessary to determine the future behavior of the agent. The equivalence classes can therefore be taken as the set of states of the generalized automaton. An agent transitions from one equivalence class to another by executing a partial trace. Partial traces thus form the labels of the transitions of the generalized automaton.

A simple construction can be used in order to represent a generalized automaton as a trace-based agent algebra. We first define the set of atomic partial traces as the set of triples (s, x, s'), where s and s' are generalized states, and x is a partial trace that labels a transition between s and s'. The set of partial and complete traces can then be obtained simply as the closure of the set of atomic traces under a special operation of concatenation that is defined only if the final state of the first trace matches the initial state of the second trace (and if the concatenation of the partial traces on the transition is also defined). We are exploring the use of this construction in our implementation of the local specification synthesis technique.

6.2.3 Applications

Several applications of this framework can be considered. Our main interest is in the study of heterogeneous systems. In particular, our future research includes applying our results to the formal definition of the interaction of models of computation in the Metropolis framework. The concept of a conservative approximation plays here a central role.

The formalization of the process of platform-based design presented in subsection 2.8.5 is also part of our current research. The concept and the formalization of the common semantic platform employed in the mapping process can be used to study new ways of combining the functional and the architectural representations of a system. More efficient estimation techniques can thus be devised if the correct abstractions are employed.

Many are the applications of the local specification synthesis technique. As explained in the introduction, several engineering problems can be stated in those terms, and therefore algebraically solved using our solution. Our future research includes applying our results to optimize the efficiency of the computation for specific finitely representable models. One such application is the automatic generation of modules that translate an abstract, transaction-level, protocol specification to one that is more detailed and suitable for design at the RTL level. By employing a combination of transaction-level and RTL-level models, verification and simulation can be made more efficient.

6.2.4 Generalized Conservative Approximations

Conservative approximations have been introduced as a broad class of relationships between models of computation that preserve refinement verification results. We have explored examples of conservative approximations for trace-based agent algebras, and we have shown how these can be obtained from homomorphisms on traces. The homomorphism however is defined to preserve the alphabet of traces, so that the conservative approximation, too, must preserve the alphabet. More interesting conservative approximations can be constructed by letting the homomorphism change the "signature" of a trace. For example, we might adopt the following definition.

Definition 6.1 (Homomorphism). Let $\mathcal{C}$ and $\mathcal{C}'$ be trace algebras. Let $h : \mathcal{B} \mapsto \mathcal{B}'$ be a function such that for all alphabets A, there exists an alphabet A' such that $h(\mathcal{B}(A)) \subseteq \mathcal{B}'(A')$. Then h is a homomorphism from $\mathcal{C}$ to $\mathcal{C}'$ if and only if

$$\begin{aligned} h(\mathit{rename}(r)(x)) &= \mathit{rename}(r)(h(x)), \\ h(\mathit{proj}(B)(x)) &= \mathit{proj}(B)(h(x)), \end{aligned}$$

where the right hand side of the equation is defined if the left hand side is defined.

While such a homomorphism can change the alphabet of a trace, it can't change it arbitrarily. In fact, in order for the right hand side above to be defined in the case of *rename*, the alphabet of $h(x)$ must be a subset of the alphabet of x. This is sufficient if we are abstracting certain signals, like clocks and activation signals, that have no meaning in a more abstract model. This is also appropriate for converting a detailed protocol specification into a more abstract, transaction-based, specification.

Arbitrary changes of the signature are also possible. In that case, however, we must consider functions between trace algebras that are not only applied to traces, but also to the operators of the algebra. In that case, the function h must satisfy the following conditions,

$$\begin{aligned} h(\mathit{rename}(r)(x)) &= \mathit{rename}(h(r))(h(x)), \\ h(\mathit{proj}(B)(x)) &= \mathit{proj}(h(B))(h(x)), \end{aligned}$$

where $h(r)$ and $h(B)$ are the appropriate renaming function and projection sets to be applied in the abstract model. By doing so we can modify the alphabet of a trace in arbitrary ways. Note that in this case the homomorphism becomes similar to a functor between categories, where a category has traces as objects and the operators of the algebra as morphisms.

We have considered conservative approximations induced by a homomorphism for complete trace structure algebras. A promising avenue of future research consists in studying conditions to obtain the most faithful abstraction in case the trace structure algebra is *not* complete (see also the discussion in subsection 1.8.5).

6.2.5 Cosimulation

We have mostly worked with denotational models. Yet, agent algebras can be constructed where agents are described operationally, for example using transition systems. Partial traces, in this case, would be essential to describe the state that a system reaches after a finite number of transitions, or simulation steps. Agent algebras could then be used to study the behavior of a simulator for a model of computation, by relating its executions to a denotational semantic domain through the use of a conservative approximation.

Our current interest however is directed towards understanding the relationships between the operational semantics of simulators for different models of computation. In particular, we are considering the co-composition of agents that belong to pairs of models of computation related by a common refinement, as described in section 2.8. The common refinement also induces a relationship between the partial executions of the simulators of the individual models, which must be kept under synchronization to approximate the behavior obtained in the common refinement. We are researching ways to derive the correct synchronization between the simulators, so that the cosimulation is consistent with the common refinement. This technique can therefore be applied to formally describe the interaction of different simulation engines, and to the process of architecture exploration in particular. In this context, we are also exploring the use of closure operators as described in subsection 2.8.4.

Bibliography

[1] M. Josie Ammer, Michael Sheets, Tufan C. Karalar, Mika Kuulusa, and Jan Rabaey. A low-energy chip-set for wireless intercom. In *Proceedings of the Design Automation Conference (DAC)*, Anaheim, CA, June 2-6 2003.

[2] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 1995.

[3] J. Augusto De Oliveira and Hans Van Antwerpen. The Philips Nexperia digial video platforms. In Grant Martin and Henry Chang, editors, *Winning the SoC Revolution. Experiences in Real Design*, pages 67–96. Kluwer Academic Publishers, 2003.

[4] Adnan Aziz, Felice Balarin, Robert K. Brayton, Maria D. Di Benedetto, Alex Saldanha, and Alberto L. Sangiovanni-Vincentelli. Supervisory control of finite state machines. In Pierre Wolper, editor, *Proceedings of Computer Aided Verification: 7th International Conference, CAV'95, Liege, Belgium, July 1995*. Springer, 1995. LNCS vol. 939.

[5] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, June 1997.

[6] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Yosinori Watanabe, and Guang Yang. Concurrent execution semantics and sequential simulation algorithms for the metropolis meta-model. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, Estes Park, CO, May 2002.

[7] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto L. Sangiovanni Vincentelli, Marco Sgroi, and Yosinori Watanabe. Modeling and designing heterogeneous systems. In J. Cortadella and A. Yakovlev, editors, *Advances in Concurrency and System Design*. Springer-Verlag, 2002.

[8] A. Balluchi, M. Di Benedetto, C. Pinello, C. Rossi, and A. Sangiovanni-Vincentelli. Cut-off in engine control: a hybrid system approach. In *IEEE Conf. on Decision and Control*, 1997.

[9] M. Barr and C Wells. *Category Theory for Computer Science*. Prentice Hall, 1990.

[10] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[11] Albert Benveniste, Luca P. Carloni, Paul Caspi, and Alberto L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In Rajeev Alur and Insup Lee, editors, *Third International Conference on Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 35–50, Philadelphia, PA, October 2003. Springer-Verlag.

[12] Jerry R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1992.

[13] Jerry R. Burch, David L. Dill, Elizabeth S. Wolf, and Giovanni De Micheli. Modeling hierarchical combinational circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93)*, pages 612–617, November 1993.

[14] Jerry R. Burch, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In M. Koutny and A. Yakovlev, editors, *Application of Concurrency to System Design*, 2001.

[15] Jerry R. Burch, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Using multiple levels of abstractions in embedded software design. In Henzinger and Kirsch [48], pages 324–343.

[16] Luca P. Carloni, Alberto L. Sangiovanni-Vincentelli Fernando De Bernardinis, and Marco Sgroi. The art and science of integrated systems design. In *Proceedings of the* 28^{th} *European Solid-State Circuits Conference, ESSCIRC 2002*, Firenze, Italy, September 2002.

[17] S. Chaki, S.K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Proc. 29th ACM Symp. Princ. of Prog. Lang.*, 2002.

[18] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, , and Freddy Y. C. Mang. Synchronous and bidirectional component interfaces. In *Proceedings of the 14th International*

Conference on Computer-Aided Verification (CAV), volume 2404 of *Lecture Notes in Computer Science*, pages 414–427. Springer-Verlag, 2002.

[19] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Marielle Stoelinga. Resource interfaces. In *Proceedings of the Third International Conference on Embedded Software (EMSOFT)*, volume 2855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[20] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew J. McNelly, and Lee Todd. *Surviving the SOC Revolution. A Guide to Platform-Based Design*. Kluwer Academic Publishers, Norwell, MA, 1999.

[21] Elaine Cheong, Judy Liebman, Jie Liu, and F. Zhao. TinyGALS: a programming model for event-driven embedded systems. In *Proceedings of the* 18^{th} *Annual ACM Symposium on Applied Computing*, pages 698–704, March 2003.

[22] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[23] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[24] Peter Cumming. The TI OMAP^{TM} platform approach to SoC. In Grant Martin and Henry Chang, editors, *Winning the SoC Revolution. Experiences in Real Design*, pages 97–118. Kluwer Academic Publishers, 2003.

[25] Julio L. da Silva Jr., Marco Sgroi, Fernando De Bernardinis, Suetfei Li, Alberto L. Sangiovanni-Vincentelli, and Jan Rabaey. Wireless protocols design: Challenges and opportunities. In *Proceedings of teh* 8^{th} *IEEE International Workshop on Hardware/Software Codesign, CODES 2000*, pages 147–151, San Diego, CA, May 2000.

[26] Julio L. da Silva Jr., Jason Shamberger, M. Josie Ammer, Chunlong Guo, Suetfei Li, Rahul Shah, Tim Tuan, Mike Sheets, Jan M. Rabaey, Bora Nikolic, Alberto L. Sangiovanni-

Vincentelli, and Paul Wright. Design methodology for picoradio networks. In *Proceedings of the Design Automation and Test in Europe*, Munich, Germany, March 2001.

[27] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, Jie Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the ptolemy project. ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, July 1999.

[28] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xia Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Heterogeneous concurrent modeling and design in java. Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley, March 2001.

[29] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[30] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In Henzinger and Kirsch [48], pages 148–165.

[31] Luca de Alfaro, Thomas A. Henzinger, and Marielle Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer-Verlag, 2002.

[32] E. A. de Kock de, G. Essink, W. J. M. Smits, P. van der Wolf, J. Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: application modeling for signal processing systems. *Proceedings of the* 37^{th} *Design Automation Conference*, 2000.

[33] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1988. Also appeared as [34].

[34] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[35] Santanu Dutta, Rune Jensen, and Alf Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital tv systems. *IEEE Design & Test of Computers*, September-October 2001.

[36] H. D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, second edition, 1994.

[37] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[38] Stephen Edwards. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, 2000.

[39] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[40] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2. Module Specifications and Constraints*. EATCS Monographs on Theoretical Computer Science. Springer, 1989.

[41] Hartmut Ehrig, Paul Boehm, and Werner Fey. Algebraic concepts for formal specification and transformation of modular software systems. In *Proceedings of the* 23^{rd} *Annual Hawaii International Conference on System Sciences*, volume 2, pages 153–164, January 2-5 1990.

[42] M. Erné, J. Koslowski, A. Melton, and G. E. Strecker. A primer on galois connections. In *Papers on General Topology and Applications*, volume 704 of *Ann. New Yosk Acad. Sci.*, pages 103–125. Madison, WI, 1993.

[43] Alberto Ferrari and Alberto L. Sangiovanni-Vincentelli. System design: Traditional concepts and new paradigms. In *Proceedings of the International Conference on Computer Design, ICCD 1999*, pages 2–12, October 1999.

[44] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, 2002.

[45] Yuri Gurevich. Evolving algebra 1993: Lipari guide. In Egon Boerger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1994.

[46] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[47] Thomas A. Henzinger. Masaccio: a formal model for embedded components. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *TCS 00: Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 549–563. Springer-Verlag, 2000.

[48] Thomas A. Henzinger and Christoph M. Kirsch, editors. *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[49] Thomas A. Henzinger, M. Minea, and V. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In M. di Benedetto and A. Sangiovanni-Vincentelli, editors, *HSCC 00: Hybrid Systems—Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 275–290. Springer-Verlag, 2001.

[50] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[51] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1986.

[52] Ben Horowitz, Judy Liebman, C. Ma, T. John Koo, Thomas A. Henzinger, Alberto L. Sangiovanni-Vincentelli, and Shankar Sastry. Embedded software design and system integration for rotocraft uav using platforms. In *Proceedings of the* 15^{th} *IFAC World Congress on Automatic Control*. Elsevier, December 2002.

[53] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proceedings of the IFIP Congress 74*, Information Processing 74, pages 471–475, Amsterdam, The Netherlands, 1974. North Holland.

[54] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Proceedings of IFIP Congress*, Information Processing 77, pages 993–998, Toronto, Canada, August 1977.

[55] Kurt Keutzer, Richard Newton, Jan Rabaey, and Alberto L. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523–1543, December 2000.

[56] Cindy Kong and Perry Alexander. Heterogeneous computer-based system specification. In *Formal Specification of Computer-Based Systems Workshop*, Washington, DC, April 20 2001.

[57] Cindy Kong and Perry Alexander. Modeling model of computation ontologies in rosetta. In *Formal Specification of Computer-Based Systems Workshop*, Lund, Sweden, April 10-11 2002.

[58] Cindy Kong and Perry Alexander. Multi-faceted requirements modeling and analysis. In *IEEE Joint International Requirements Engineering Conference*, Essen, Germany, September 9-13 2002.

[59] Cindy Kong and Perry Alexander. The rosetta meta-model framework. In *Proceedings of the IEEE Engineering of Computer-Based Systems Symposium and Workshop*, Huntsville, AL, April 7-11 2003.

[60] Kim G. Larsen and Liu Xinxin. Equation solving using modal transition systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 90)*, pages 108–117, June 4-7 1990.

[61] Luciano Lavagno, Alberto L. Sangiovanni-Vincentelli, and Ellen Sentovich. Models of computation for embedded system design. In *Proceedings of the NATO ASI on System Level Synthesis for Electronic Design*, Il Ciocco, Italy, August 1998. Kluwer Academic Publishers.

[62] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 17(12):1217–1229, December 1998.

[63] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75:1235–1245, September 1987.

[64] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. In Henzinger and Kirsch [48].

[65] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.

[66] S. MacLane. *Categories for the Working Mathematician*. Graduate Text in Mathematics. Springer Verlag, New York, 1971.

[67] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[68] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[69] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[70] T. J. Mowbray, W. A. Ruh, and R. M Soley. *Inside CORBA: Distributed Object Standards and Applications*. Addison-Wesley, 1997.

[71] Radu Negulescu. *Process Spaces and the Formal Verification of Asynchronous Circuits*. PhD thesis, University of Waterloo, Canada, 1998.

[72] Radu Negulescu. Process spaces. In C. Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[73] Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'02)*, November 2002.

[74] Roberto Passerone, James A. Rowson, and Alberto L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *DAC*, San Francisco, CA, June 1998.

[75] B. C. Peirce. *Category Theory for Computer Scientists*. MIT Press, 1991.

[76] Vaughan R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

[77] Jan Rabaey, M. Josie Ammer, Julio L. da Silva jr., Danny Patel, and S. Roundy. Picoradio supports ad hoc ultra-low power wireless networking. *IEEE Computer Magazine*, pages 42–48, July 2000.

[78] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[79] James A. Rowson and Alberto L. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the 34^{th} Design Automation Conference, DAC 1997*, pages 178–183, June 1997.

[80] Martijn J. Rutten, Jos T. J. van Eijndhoven, Egbert G. T. Jaspers, Pieter van der Wolf, Om Prakash Gangwal, Adwin Timmer, and Evert-Jan D. Pol. A heterogeneous multiprocessor architecture for flexible media processing. *Design & Test of Computers*, pages 39–50, July-August 2002.

[81] Alberto Sangiovanni-Vincentelli, Marco Sgroi, and Luciano Lavagno. Formal models for communication-based design. In *Proceedings of the Eleventh International Conference on Concurrency Theory*, August 2000.

[82] Alberto L. Sangiovanni-Vincentelli. Defining platform-based design. *EEdesign*, February 2002.

[83] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. A classification of models for concurrency. In *Proceedings of the 4^{th} International Conference on Concurrency Theory, CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1993.

[84] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170:297–348, 1996.

[85] Marco Sgroi, Michael Sheets, Andrew Mihal, Kurt Keutzer, Sharad Malik, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. Addressing system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38^{th} Design Automation Conference, DAC 2001*, pages 667–672, Las Vegas, NV, June 2001.

[86] C. J. Richard Shi. Entity overloading for mixed-signal abstraction in VHDL. In *Proceedings of EURO-DAC '96, European Design Automation Conference*, pages 562–567, September 23-27 1996.

[87] Kanna Shimizu, David L. Dill, , and Alan J. Hu. Monitor-based formal specification of PCI. In *FMCAD*, Austin, Texas, 2000.

[88] Pete Sterrantino, David Hacker, and Ralph Fravel. An embedded computer-based intraoperative spinal nerve integrity monitor. In *Proceedings of the 16^{th} IEEE Symposium on Computer-Based Medical Systems (CBMS 2003)*, pages 367–370, June 2003.

[89] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1997.

[90] Functional specification for SystemC 2.0, January 2001. Available from the Open SystemC Initiative (OSCI) at http://www.systemc.org.

[91] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[92] Paul Taylor. *Practical Foundations of Mathematics*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1999. Also available on the web at the address http://www.dcs.qmul.ac.uk/∼pt/Practical-Foundations/index.html.

[93] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[94] Brett Warneke, Matt Last, Brian Leibowitz, and Kris Pister. Smart Dust: Communicating with a cubic-millimeter computer. *IEEE Computer*, 34:44–51, January 2001.

[95] Glynn Winskel and Mogens Nielsen. Models for concurrency. In S. Abramsky et al., editor, *Handbook of Logic in Computer Science*, volume Vol. 4, Semantic Modelling. Oxford University Press, Oxford, 1995.

[96] Elizabeth S. Wolf. *Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution*. PhD thesis, Department of Computer Science, Stanford University, October 1995.

[97] B. Woodward, R. S. H. Istepanian, and C. I. Richards. Design of a telemedicine system using a mobile telephone. *IEEE Transactions on Information Technology*, 5(1):13–15, March 2001.

[98] Nina Yevtushenko, Tiziano Villa, Robert K. Brayton, Alex Petrenko, and Alberto L. Sangiovanni-Vincentelli. Sequential synthesis by language equation solving. Memorandum No. UCB/ERL M03/9, Electronic Research Laboratory, University of California at Berkeley, 2003.