Overcoming Heterophobia: Modeling Concurrency in Heterogeneous Systems

Jerry Burch
Cadence Berkeley Laboratories
2001 Addison St third floor
Berkeley, CA 94709, USA
jrb@cadence.com

Roberto Passerone
Cadence Design Systems, Inc.
2670 Seely Road
San Jose, CA 95134
robp@cadence.com

Alberto L. Sangiovanni-Vincentelli
Department of EECS
University of California at Berkeley
Berkeley, CA 94720
alberto@eecs.berkeley.edu

Abstract

*System level design is complex. One source of this complexity is that systems are often heterogeneous: different models of computation (*e.g., *dataflow, FSMs) are used to describe different components of a system. Existing formal methods for concurrent systems are typically based on one particular model of computation, so it is difficult to formalize the interaction between heterogeneous components. In this paper, we develop a framework for formalizing the relationships between different models of computation.*

1 Introduction

Microscopic devices, powered by ambient energy in their environment, will be able to sense numerous fields, position, velocity, and acceleration, and communicate with appropriate and sometimes substantial bandwidth in the near area. Larger, more powerful systems within the infrastructure will be driven by the continued improvements in storage density, memory density, processing capability, and system-area interconnects as single board systems are eclipsed by complete systems on a chip. Data movement and transformation is of central importance in such applications. Future devices will be network-connected, channeling streams of data into the infrastructure, with moderate processing on the fly. Others will have narrow, application-specific UIs. Applications will not be centered within a single device, but stretched over several, forming a path through the infrastructure. In such applications, the ability of the system designer to specify, manage, and verify the functionality and performance of *concurrent behaviors* is essential.

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 set of given constraints and specifications without resorting to extensive simulation or tests on prototypes. In the face of growing complexity and tightening of time-to-market, cost and safety constraints, this approach will have to yield to more rigorous methods. We believe that it is most likely that the preferred approaches to the implementation of complex embedded systems will include the following aspects:

1. Design time and cost are likely to dominate the decision-making process for system designers. Therefore, design reuse in all its shapes and forms, as well as just-in-time, low-cost design debug techniques will be of paramount importance.

2. Designs must be captured at the highest level of abstraction to be able to exploit all the degrees of freedom that are available. Such a level of abstraction should not make any distinction between hardware and software, since such a distinction is the consequence of a design decision.

3. The implementation of efficient, reliable, and robust approaches to the design, implementation, and programming of concurrent systems is essential. In essence, whether the silicon is implemented as a single, large chip or as a collection of smaller chips interacting across a distance, the problems associated with concurrent processing and concurrent communication must be dealt

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE

with in a uniform and scalable manner. In any large-scale embedded systems program, concurrency must be considered as a first class citizen at all levels of abstraction and in both hardware and software.

4. Concurrency implies communication among components of the design. Communication is too often intertwined with the behavior of the components of the design so that it is very difficult to separate out the two issues. Separating communication and behavior is essential to overcome system design complexity. If in a design component behaviors and communications are intertwined, it is very difficult to re-use components since their behavior is tightly dependent on the communication with other components of the original design.

We have advocated the introduction of rigorous methodologies for system-level design for years (e.g., [1, 12]) but we feel that there is still much to do. Recently we have directed our efforts to a new endeavor that tries to capture the requirements of present day embedded system design: the Metropolis project.

The *Metropolis* project, supported by the Gigascale Silicon Research Center, started two years ago and involves a large number of people in different research institutions. It is based on the following principles:

- **Orthogonalization of concerns**: In Metropolis, behavior is clearly separated from implementation. Communication and computation are orthogonalized. Communication is recognized today as the main difficulty in assembling systems from basic components. Errors in software systems can often be traced to communication problems. Metropolis was created to deal with communication problems as the essence of the new design methodology. Communication-based design will allow the composition of either software or hardware blocks at any layer of abstraction in a controlled way. If the blocks are correct, the methodology ensures that they communicate correctly.

- **Solid theoretical foundations that provide the necessary infrastructure for a new generation of tools**: We believe that without a rigorous approach, the goal of correct, efficient, reliable and robust designs cannot be achieved. The tools used in Metropolis will be interoperable and will work at different levels of abstraction, they will verify, simulate, and map designs from one level of abstraction to the next, help choose implementations that meet constraints and optimize the criteria listed above. The theoretical framework is necessary to make our claims of correctness and efficiency true. Metropolis will deal with both embedded software and hardware designs since it will intercept the design specification at a higher level of abstraction. The design specifications will have precise semantics. The semantics is essential to be able to: (i) reason about designs, (ii) identify and correct functional errors, (iv) initiate synthesis processes.

- **Reduction of design time and cost by using platforms**: Platforms have been a common approach to reuse. We have formalized and elaborated the concept of platform to yield an approach that combines hardware and software platforms to build a system platform. An essential part of a platform is its communication architecture. Communication-based design principles have been used to define standard communication schemes, but we advocate a more abstract use of communication-based design to allow more flexible and better-specified communication architectures.

The Metropolis methodology, by leveraging these three basic principles, builds an environment where the design of complex systems will be a matter of days versus the many months needed today Complex, heterogeneous designs will be mapped into flexible system platforms by highly optimized "design agents" and verified by "verification agents" in a formal logic framework.

An essential aspect of the Metropolis methodology is the adoption of formal definition of the semantics of communication so that implementation choices will be correct by construction.

Several formal models have been proposed over the years (see e.g. [7]) to capture one or more aspects of computation as needed in embedded system design. We have been able to compare the most important models of computations using a unifying theoretical framework introduced recently by Lee and Sangiovanni-Vincentelli [9].

However, this denotational framework has only helped us to identify the sources of difficulties in combining different models of computation that are certainly needed when complex systems are being designed. In this case, the partition of the functionality of the design into different models of computation is somewhat arbitrary as well as arbitrary are the communication mechanisms used to connect the "ports" of the different models. We believe that it is possible to optimize across model-of-computation boundaries to improve performance and reduce errors in the design at an early stage in the process.

There are many different views on how to accomplish this. There are two essential approaches: one is to develop encapsulation techniques for each pair of models that allow different models of computation to interact in a meaningful way, i.e., data produced by one object are presented to the other in a consistent way so that the object "understands" [3, 4]. The other is to develop an encompassing framework where all the models of importance "reside" so that their combination, re-partition and communication happens in the same generic framework and as such may be better understood and optimized. While we realize that today heterogeneous models of computation are a necessity, we believe that the second approach will be possible and will provide a designer a powerful mechanism to actually select the appropriate models of computation, (e.g., FSMs, Data-flow, Discrete-Event, that are positioned in the theoretical framework in a precise order relationship so that their interconnection can be correctly interpreted and refined) for the essential parts of his/her design.

In this paper, we focus on this very aspect of the approach: the formal definition of a framework where formal models can be rigorously defined and compared, and their interconnections can be unambiguously specified. We use a kind of abstract algebra to provide the underlying mathematical machinery. We believe that this framework will be essential to provide the foundations of an intermediate format that will provide the Metropolis infrastructure with a formal mechanism for interoperability among tools and specification methods.

The paper is organized as follows. Section 2 describes our view of the requirements for a formal model of heterogeneous systems. Sections 3 through 6 describe our framework, which is based on trace algebra [2]. Section 7 summarizes and describes future work.

2 Requirements for a Formal Model

This section describes our view of what a model of computation is, our approach to constructing formal models, and what assumptions that approach depends on.

2.1 What is a Model of Computation?

In our terminology, a model of computation is a distinctive paradigm for computation, communication, etc. For example, the Mealy machine model of computation is a paradigm where data is communicated via signals and all agents operate in lockstep (we use "agent" as a generic term that includes both hardware circuits and software processes). The Kahn Process Network model 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.

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 [9] and the operational semantics underlying the Ptolemy II simulator [4]. 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 a computation domains) are distinguished by differences in firing rules and communication protocols.

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.

2.2 Strategy for Constructing a Formal Model

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.

There is an important tradeoff 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 tradeoffs in this regard, rather than be required to conform to a particular choice made by us.

2.3 Assumptions About Models of Computation

We wish to have a very general framework that can handle a variety of models of computation. However, we make some assumptions about the semantic domains that will be used. We have proved many generic theorems that hold for any semantic domain that satisfies these assumptions. To analyze a newly constructed semantic domain within this framework, one starts by proving that the domain satisfies the assumptions. Then, the above generic theorems can be used without having to reprove them.

Our most restrictive assumption is that all models of computation will be linear time, rather than branching time. Generally, there are two reasons why branching time models are used. First, branching time temporal logics such as CTL can express useful non-linear properties that are relatively inexpensive to verify using automatic model checkers. Although the semantics of CTL cannot be represented in our framework, CTL can still be used as a specification language to verify linear time models, which do fit in our framework.

The second reason to use branching time models has to do with certain semantic inadequacies of linear time models. Our approach to handling these inadequacies follows Dill [6]. To describe this approach, we use the standard CCS "Vending Machine" example, which is illustrated in figure 1 (see page 54 of Dill for a similar discussion of this example). The first vending machine inputs money (action a) and lets the customer select one of two items by taking inputs b and c. To simplify the example, the vending machine halts after just one transaction. The other vending machine takes money, then makes an internal decision about which product it will allow the customer to choose (without telling the customer). The customer selects item b or c, and either gets it or not, depending on the decision made by the machine.

These two vending machines should be distinguished. However, the sets of traces of the two machines appears to be the same: $\{ab, ac\}$. CCS and other process algebras use a branching time semantics to distinguish the two machines. However there are other approaches. Notice that the actions a, b and c are rendezvous in this model of computation. One difference between the vending machines is that if the customer makes choice b, then the first vending machine is guaranteed to complete the b rendezvous while the second vending machine may not. This difference is not reflected in the trace set because there is no indication of a rendezvous that is initiated by the customer but not completed by the machine. If the alphabet of the trace language is enriched with symbols that indicate a partially complete rendezvous (for example), then the two vending machines can be distinguished by their trace sets, without requiring branching time semantics.

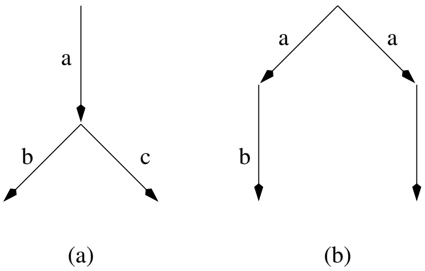

Figure 1. CCS-like trees for two "vending machines"

The above argument that branching time is not needed in many cases is, of course, merely suggestive, not definitive. A long term goal of this research is to extend the framework to include branching time models so we can more precisely characterize when such models are needed. However, for the main goal of this research, improving formal semantic methods for heterogeneous systems, we believe that branching time semantics are not necessary.

Our current models of agents do not allow ports to be of different types. Also, we do not explicitly model bidirectional communication: all ports are designated as inputs or outputs. However, the effects of these limitations are more cosmetic than fundamental. Different port types can be modeled using a value domain that is the union of all types of interest. Bidirectional communication is modeled by allowing multiple output ports to share control over signals. The fact that such output ports are "output" in name only is why we consider this restriction to be merely cosmetic. We plan to extend our framework to remove these cosmetic restrictions.

There are many other traits that distinguish models of computation, including: levels of abstraction, partial orders models (*e.g.*, POMSETS [11] and Mazurkiewicz Traces [10]) vs total orders, action-based vs value-based, different styles of computation, communication and coordination. For all these traits, our framework is unbiased. We have constructed different models with several different combinations of these traits, all of which satisfy the assumptions we place on semantic domains.

2.4 Overview

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 sections.

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 T is a specification and T' is an implementation, then T' is said to satisfy T if the language of T' is a subset the language of T. The idea is that each sequence, sometimes called a *trace,* represents a behavior; an implementation satisfies a specification iff 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 implementaiton 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 proceeed from a definition of behavior and structural properties of models towards the notion of "approximations" as a way of capturing the behavior containment idea. Once we have established this key relationship, it is possible to compare and combine models trying to find common grounds where behavior representations are consistent. Intuitively, we may find different ways of approximating models and consequently compositions and comparisons are dependent of 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 of the way the two models communicate. These assumptions more often than not are implicit and may be imposed by the tools designers use. We do not want to be prisoners of this way of designing that may yield sub-optimal and even incorrect implementations. This is the main motivation of the heavy machinery we introduce. Bearing with us on this tortuous path will hopefully yield the fruit of a rigorous way of defining, composing and proving properties of heterogeneous models of computation.

In our framework, traces can be any mathematical object that has certain properties. In section 3, these properties are formalized in the axioms of *trace algebra,* which is a kind of abstract algebra that has a set of traces as its domain. Section 4 describes *trace structure algebra,* which has as its domain a set of trace structures, each containing a subset of the traces from a given trace algebra. The notion of one trace structure satisfying another is based on trace set containment.

Here is a simple example of a trace algebra and a trace structure algebra. Let the set of traces over an alphabet A be A^∞, which is the set of finite and infinite sequences over A. A tuple $((I, O), P)$ is a trace structure if $P \subseteq A^\infty$, where I and O are sets of input and output signals, respectively, and $A = I \cup O$ is the alphabet of the trace structure.

We define the operations of parallel composition, projection (for abstracting away internal signals) and renaming (for instantiating models with new port names) on trace structures by first defining projection and renaming on individual traces. If $x \in A^\infty$ and $B \subseteq A$, then $\mathit{proj}(B)(x)$ is the string formed from x by removing all symbols not in B. If r is a bijection with domain A, then $\mathit{rename}(r)(x)$ is the string formed from x by replacing every symbol a with $r(a)$.

Projection and renaming on trace structures are just the natural extensions of the corresponding operations on traces. In particular, if $T = ((I, O), P)$ is a trace structure, $I \subseteq B \subseteq A$ and r is a bijection with domain A, then

$$\begin{aligned} \mathit{proj}(B)(T) &= ((I, O \cap B), \mathit{proj}(B)(P)), \\ \mathit{rename}(r)(T) &= ((r(I), r(O)), \mathit{rename}(r)(P)), \end{aligned}$$

where the operations of projection and renaming on traces are naturally extended to sets of traces. If $T = ((I, O), P)$ is equal to the parallel composition of T' and T'', then P is the set of $x \in A^\infty$ such that

$$\mathit{proj}(A')(x) \in P' \wedge \mathit{proj}(A'')(x) \in P''.$$

Given our definition of projection on strings, this is a natural definition of parallel composition.

Looking at the above definitions more closely, we can see how these concepts can be generalized to include many different kinds of models. Rather than always using strings in a formal language as the domain of traces, we can use any domain that has projection and renaming operations defined on it and that satisfies certain requirements. These requirements are formalized in the axioms of trace algebra. In each case, the operations on trace structures are defined exactly as above, in terms of the operations on individual traces. The resulting trace structure algebra enjoys certain useful properties because the underlying traces satisfy the axioms of trace algebra.

Trace structures algebras are best used in combination, and indeed they must be related if we are to address the problems of heterogeneous design. Hence, after providing some very general methods for constructing different models of concurrent systems, we study how to prove relationships between these models. One of these relationships is the concept of a *conservative approximation* [2]. Informally, a conservative approximation is a mapping between the trace structures of two trace structure algebras such that the following condition is satisfied: if an "implementation satisfies" a specification in the first trace structure algebra, then the corresponding "implementation also satisfies" the corresponding specification in the second algebra. 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 and reasoning. In the paper we show how to derive a conservative approximation from a relation between the trace algebras underlying the trace structure algebras.

While a conservative approximation is used to abstract away details of a model to simplify the representation and the verification task, inverses of conservative approximations can be used to refine a specification into a more detailed one. In the paper we characterize the conditions under which an inverse is defined, and we use it to solve the problem of composing different models by mapping them into a common refinement. The restrictions on the behaviors computed at the lower level are finally abstracted again in the two original models. This approach can be applied transitively to a set of models that are related by such mappings.

3 Trace Algebra

We 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 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. The form of trace algebra we define here has only complete traces; it is intended to represent only complete behaviors. We use the symbol '$\mathcal{C}$' to denote trace algebras. Since we only consider here trace algebras with complete traces and without partial traces, we use a subscript 'C' (*e.g.,* '$\mathcal{C}_C$') to denote the trace algebras used in this paper.

We begin with a few preliminary definitions.

Definition 1. We use W to denote a set of *signals.* The set W is usually infinite, but this is not required.

Definition 2. An *alphabet A over W* is any subset of W.

The *rename* operation uses a *renaming function,* which is a bijection from one alphabet to another.

Definition 3. A function r with domain A and codomain B, where A and B are alphabets over W, is a *renaming function over W* if r is a bijection.

Definition 4. The function $r\,|_{B' \to B''}$ is constructed from the function r by restricting to domain B' and codomain B''.

Now we are ready to define trace algebra.

Definition 5. A *trace algebra $\mathcal{C}_C$ over W* is a triple

$$(\mathcal{B}_C, \mathit{proj}, \mathit{rename}).$$

For every alphabet A over W, $\mathcal{B}_C(A)$ is a non-empty set, called the set of traces over A. Slightly abusing notation, we also write $\mathcal{B}_C$ as an abbreviation for

$$\bigcup \{\mathcal{B}_C(A) : A \subseteq W\}.$$

For every alphabet B over W and every renaming function r over W, $\mathit{proj}(B)$ and $\mathit{rename}(r)$ are partial functions from $\mathcal{B}_C$ to $\mathcal{B}_C$. The following axioms T1 through T8 must also be satisfied. For all axioms that are equations, we assume that the left side of the equation is defined.

T1. $\mathit{proj}(B)(x)$ is defined iff there exists an alphabet A such that $x \in \mathcal{B}_C(A)$ and $B \subseteq A$. When defined, $\mathit{proj}(B)(x)$ is an element of $\mathcal{B}_C(B)$.

T2. $\mathit{proj}(B)(\mathit{proj}(B')(x)) = \mathit{proj}(B)(x)$.

T3. If $x \in \mathcal{B}_C(A)$, then $\mathit{proj}(A)(x) = x$.

T4. Let $x \in \mathcal{B}_C(A)$ and $x' \in \mathcal{B}_C(A')$ be such that $\mathit{proj}(A \cap A')(x) = \mathit{proj}(A \cap A')(x')$. For all A'' where $A \cup A' \subseteq A''$, there exists $x'' \in \mathcal{B}_C(A'')$ such that $x = \mathit{proj}(A)(x'')$ and $x' = \mathit{proj}(A')(x'')$.

T5. $\mathit{rename}(r)(x)$ is defined iff x is an element of the set $\mathcal{B}_C(\mathit{dom}(r))$. When defined, $\mathit{rename}(r)(x)$ is an element of $\mathcal{B}_C(\mathit{codom}(r))$.

T6. $\mathit{rename}(r)(\mathit{rename}(r')(x)) = \mathit{rename}(r \circ r')(x)$.

T7. If $x \in \mathcal{B}_C(A)$, then $\mathit{rename}(\mathit{id}_A)(x) = x$, where id_A is the identity function with domain A.

T8. $\mathit{proj}(r(B))(\mathit{rename}(r)(x)) =$
$\mathit{rename}(r\mid_{B \to r(B)})(\mathit{proj}(B)(x))$.

T1 and T5 state when the operations on traces are defined. T2, T3, T6, T7 and T8 are clearly consistent with the intuitive meaning of the projection and renaming operations. The remaining axiom, T4 is a kind of "diamond property", as illustrated in figure 2. As an example of applying T4, consider the case where traces are sequences. Let $A = \{a, b\}$, $A' = \{b, c\}$, $x = abab$ and $x' = bcb$. Clearly $\mathit{proj}(A \cap A')(x)$ and $\mathit{proj}(A \cap A')(x')$ are both equal to bb. Choosing $x'' = abacb$ demonstrates the T4 holds for this pair of sequences. Intuitively, T4 requires that if two traces x and x' are compatible on their shared signals (*i.e.*, $A \cap A'$), then there exists a trace x'' that corresponds to the synchronous composition of x and x'.

Note 6. We naturally extend the renaming and projection operations on traces to operations on sets of traces. For example, if $\mathit{rename}(r)(x)$ is defined for every x in X, then $\mathit{rename}(r)(X)$ is defined such that

$$\mathit{rename}(r)(X) = \{\mathit{rename}(r)(x) : x \in X\}.$$

3.1 Examples

As an example trace algebra, we formalize the trace algebra briefly described at the beginning of section 3, which we call $\mathcal{C}^I_C$. We always use the symbol '$\mathcal{C}$' to denote trace algebras, and the superscript 'I' is a mnemonic for an (untimed) interleaving model; the subscript 'C' indicates that there are only complete traces in the trace algebra (*i.e.,* a trace algebra without partial traces).

Definition 7. For a given set of signals W, the trace algebra $\mathcal{C}^I_C = (\mathcal{B}^I_C, \mathit{proj}^I, \mathit{rename}^I)$ over W is defined as follows:

- For every alphabet A over W, the set $\mathcal{B}^I_C(A)$ of traces over A is A^∞, which is the set of finite and infinite sequences over A.
- If $x \in \mathcal{B}^I_C(A)$ and $B \subseteq A$, then $\mathit{proj}^I(B)(x)$ is the sequence formed from x by removing every symbol a not in B. More formally, if $x' = \mathit{proj}^I(B)(x)$, then the length of x' (written $\mathit{len}(x')$) is

  $$|\{j \in \mathcal{N} : 0 \leq j < \mathit{len}(x) \wedge x(j) \in B\}|$$

  and $x'(k) = x(n)$ for all $k < \mathit{len}(x')$, where n is the unique integer such that $x(n) \in B$ and

  $$k = |\{j \in \mathcal{N} : 0 \leq j < n \wedge x(j) \in B\}|.$$
- If $x \in \mathcal{B}^I_C(A)$ and r is a renaming function over W with domain A, then

  $$\mathit{rename}(r)(x) = \lambda n \in \mathcal{N}^{\not-}[r(x(n))].$$

Trace algebra can be used to construct a large variety of behavior models. The trace algebra $\mathcal{C}^I_C$, for which $\mathcal{B}_C(A) = A^\infty$, is just one example. To provide more intuition about the range of possible trace algebras, we informally describe several more examples.

The simplest possible trace algebra has exactly one trace; call it x_0. For any alphabet A, the set of traces over A is $\mathcal{B}_C(A) = \{x_0\}$. If B is an alphabet and r is a renaming function, then $\mathit{proj}(B)(x_0)$ and $\mathit{rename}(r)(x_0)$ are defined and are equal x_0. This trace algebra does not distinguish between any behaviors; all behaviors are represented by the same trace. For this reason it is not a useful trace algebra, but it does satisfy the necessary axioms.

A slightly more complicated trace algebra has $\mathcal{B}_C(A) = 2^A$. For any trace x, $\mathit{proj}(B)(x)$ is defined and is equal to $x \cap B$. On the other hand, $\mathit{rename}(r)(x)$ is defined iff $x \subseteq \mathit{dom}(r)$; when defined, it is equal to $r(x)$, where r is naturally extended to sets. It is easy to show that this trace algebra satisfies T1 through T8; in particular, if x and x' satisfy the hypothesis of T4, then $x'' = x \cup x'$ is sufficient to show that T4 is satisfied. Traces in this trace algebra do not provide any information about actions occurring in sequence, only information about what actions occurred a non-zero number of times during a behavior. Alternatively, if $a \in x$, then this could be interpreted to mean that a occurred an odd number of times during the behavior represented by x.

Traces in the last two examples provide less information about a behavior than do traces in $\mathcal{C}^I_C$. As an example of a trace algebra that provides more information than $\mathcal{C}^I_C$, let $\mathcal{B}_C(A) = (2^A)^\omega$. For any trace x, $\mathit{proj}(B)(x)$ is defined and is formed from x by intersecting each element of the sequence with B. The function $\mathit{rename}(r)$ is the natural extension of r to sequences of sets. Unlike traces in $\mathcal{C}^I_C$, these traces can be interpreted as providing information about the time at which events occur. If x is such a

$$\begin{tikzcd}
& x'' \in \mathcal{B}(A'') \arrow[dl, "\mathit{proj}(A)"'] \arrow[dr, "\mathit{proj}(A')"] & \\
x \in \mathcal{B}(A) \arrow[dr, "\mathit{proj}(A \cap A')"'] & & x' \in \mathcal{B}(A') \arrow[dl, "\mathit{proj}(A \cap A')"] \\
& \mathit{proj}(A \cap A')(x) = \mathit{proj}(A \cap A')(x') &
\end{tikzcd}$$

Figure 2. According to T4, if there exists an x and an x' that satisfy the lower half of the diamond, then there exists an x'' that satisfies the upper half, for any alphabet A'' such that $A \cup A' \subseteq A''$.

trace, then $x(n)$ is the set of events that occurred at time n. The set $x(n)$ must be defined for all integers n; therefore, each trace x must be an infinite sequence.

A trace algebra that provides an intermediate amount of information between the last example and $\mathcal{C}_C^I$ can be constructed by letting $\mathcal{B}_C(A) = (2^A - \{\emptyset\})^\infty$. The renaming operation is the same as the last example, except that it is also extended to finite sequences. Projection is similar to the last example, except that after doing the intersection, any instances of the empty set that result must be removed from the sequence. Like $\mathcal{C}_C^I$, this trace algebra is untimed; however, it represent simultaneity explicitly, unlike interleaving semantics.

In the continuous time trace algebra $\mathcal{C}_C^{CTU}$, each trace over an alphabet A is an element of $2^{A \times \Re^{\not -}}$, where $\Re^{\not -}$ is the set of non-negative real numbers. Each trace is a set of events; each event is an ordered pair of an action and a time stamp. An isomorphic trace algebra can be constructed by taking advantage of the natural bijection between $2^{A \times \Re^{\not -}}$ and $\Re^{\not -} \mapsto 2^A$. If x is a trace in $\Re^{\not -} \mapsto 2^A$, then $x(t)$ is the set of actions that occurred at time t.

All of the trace algebras we have described are action based, but trace algebra can also be used for state-based models. For an agent with alphabet A, we interpret each $a \in A$ as a state variable. Let V be the set of values that can be taken by state variables. Then, each state is an element of $A \mapsto V$. A trace algebra based on sequences of states would have $\mathcal{B}_C(A)$ equal to $(A \mapsto V)^\omega$, which can also be written as $\mathcal{N}^{\not -} \mapsto (A \mapsto V)$.

For a continuous time, state-based model, let $\mathcal{B}_C(A) = \Re^{\not -} \mapsto (A \mapsto V)$. If x is such a trace, then $x(t)$ is the state at time t. If V is the set of real numbers, then this trace algebra could be used as a circuit model that represents both continuous time and continuous voltage.

The following three examples will be used throughout the rest of the paper to explore how we can characterize different models of computation in terms of their underlying traces. We will consider Kahn Process Networks, synchronous or finite-state systems, and discrete-time systems. In all three examples we only explicitly define the projection operation, while the renaming operation is implicitly defined in the obvious way.

3.1.1 Kahn Process Networks

Kahn Process Networks, agents are characterized by a function that takes a stream of values from each of the inputs, and produces streams of values on each of the outputs. In this section, we describe the trace algebra $\mathcal{C}_C^K$ used in modeling Kahn Process Networks.

Assume, without loss of generality, that all inputs and all outputs take values from a set V. Then we can represent the set of streams at a particular input or output as a sequence of values over V. The set of the finite and infinite sequences from V is denoted as V^∞. A behavior of a Kahn Process is composed of a stream for each of the inputs and each of the outputs. If the alphabet A is the set of inputs and outputs for a process, then a trace of a Kahn Process is a function $f : A \mapsto V^\infty$.

We define the projection operation for Kahn Processes as follows: if B is a set of inputs and outputs, and $B \subseteq A$ then the projection $proj(B)(f)$ of a trace $f : A \mapsto V^\infty$ is defined as the restriction $f|_B$ of the function f:

$$proj(B)(f) = f|_B$$

To show that this is a trace algebra we must prove that it satisfies the necessary axioms. T4 is the only axiom that does not follow easily from the definitions:

Theorem 8. If $f : A \mapsto V^\infty$ and $f' : A' \mapsto V^\infty$ are traces such that $proj(A \cup A')(f) = proj(A \cup A')(f')$, then for all A'', where $A \cup A' \subseteq A''$, there exists f'' such that $f = proj(A)(f'')$ and $f' = proj(A')(f'')$.

Let $f'' : A'' \mapsto V^\infty$ be such that f'' agrees with f on A and agrees with f' on A'. We know that f and f' agree on the common elements of A and A', so f'' is well defined and satisfies the theorem.

3.1.2 Synchronous Networks

A synchronous agent processes all of its inputs and outputs in (a sequence of) atomic steps. This is the case, for example, for Mealy machines. The definition of the traces in this case differs slightly from the previous case because we want a structure that can represent the simultaneity of the events at the interface of a process. We denote this trace algebra with $\mathcal{C}_C^S$.

A synchronous process operates on sequences of values that are synchronized. Consider the set of values V and the set of inputs and outputs A as before. Then the set of value that can be observed at any synchronization point (e.g. a clock edge) is given by a function $g : A \mapsto V$. We are interested in a sequence of these values. Hence a trace in this case is an element of the set of sequences $(A \mapsto V)^\infty$. The operation of projection is defined in a similar way and consists of restricting the domain of each of the functions in the sequence to a smaller set of elements. Hence, if we denote with $\langle g_i \rangle$ a sequence of functions, and for $B \subseteq A$, we define the operation of projection as

$$proj(B)(\langle g_i \rangle) = \langle g_i|_B \rangle.$$

As with Kahn Processes, the operation of projection satisfies T4.

In order to avoid having to change the alphabet when using functions later in the paper, we always add an element to A to represent a global clock signal. This is not necessary at this level of abstraction and is done only for convenience.

3.1.3 Discrete Time Networks

The last example trace algebra is a discrete-time model, in which a process samples the sets of inputs and produces a set of outputs at discrete instants in time. Moreover, the instants in time are characterized by their distance, i.e. they form a metric space. Without loss of generality, we will consider discrete time models where the model of time is represented by the positive integers $\mathcal{N}^{\neq}$. In practice, the integers represent time in a particular unit, for example picoseconds. Note also that we are considering a regular sample time, while in general a discrete time model may involve sampling at irregular times (for example the sampling time may be adjusted according to the rate of variations of values in the signals). This trace algebra is denoted by $\mathcal{C}_C^D$.

In the discrete-time model each input and output signal assumes a value at each time stamp. Hence, traces are element of $\mathcal{N}^{\neq} \mapsto (A \mapsto V)$.

As in the other two cases we can define the operation of projection on traces by just restricting the function to a subset of the domain A:

$$proj(B)(\mathcal{N}^{\neq} \mapsto g) = \mathcal{N}^{\neq} \mapsto g|_B$$

As before, the validity of the axioms of trace algebra are easy to verify.

4 Trace Structure Algebra

Agents communicate through signals. We associate with each agent an *agent signature* (or just *signature*), which describes sets of input signals and output signals.

Definition 9. The set of *agent signatures* Γ *over* W is the set of ordered pairs (I, O) such that I and O are disjoint subsets of W. We use γ to denote agent signatures (often called just *signatures*).

In a signature (I, O) over W, the set W is usually infinite and the sets I and O are usually finite, but this is not required. In future work, we plan to extend signatures to allow bidirectional signals and to associate type information with each signal.

Definition 10. If $\gamma = (I, O)$ is a signature over W, then $A = I \cup O$ is the *alphabet of* γ.

0-7695-1071-X/01 $10.00 © 2001 IEEE

Note 11. When we mention a signature γ, we also implicitly define I and O so that $\gamma = (I, O)$. We also implicitly define A to be the alphabet of γ. If the name of the signature is decorated with primes and/or subscripts, those decorations carry over to the implicitly defined quantities. For example, mentioning a signature γ_1' implicitly defines I_1', O_1' and A_1'.

The parallel composition of two agents T and T' (written $T \parallel T'$) corresponds to, for example, joining two circuits or running two processes concurrently. In the resulting composition, T and T' communicate through shared signals. We require that no signal be an output of both T and T'. The agent $\mathit{rename}(r)(T)$ is formed from T by renaming the signals of T according to r. If B is a subset of the alphabet of T, then $\mathit{proj}(B)(T)$ has B as its alphabet; the remaining signals of T are not externally visible. We allow only outputs of T to be hidden, so B must contain all of the inputs of T.

We are now ready to define the concept of a trace structure algebra. Trace structures are constructed from the traces of a trace algebra, and are used to represent agents. Here we consider trace structures that contain one set of traces, which represents the set of *possible* behaviors of an agent.

Definition 12. Let $\mathcal{C}_C = (\mathcal{B}_C, \mathit{proj}, \mathit{rename})$ be a trace algebra over W. The set of *trace structures* over $\mathcal{C}_C$ is the set of ordered pairs (γ, P), where

- γ is a signature over W,
- A is the alphabet of γ, and
- P is a subset of $\mathcal{B}_C(A)$.

We call γ the *signature* and P the set of *possible traces* of a trace structure $T = (\gamma, P)$.

A trace structure (γ, P) represent an agent with signature γ; each trace in P represents a possible complete behavior of the agent.

Note 13. When we mention a trace structure T, we implicitly define γ to be its signature and P to be its set of possible traces. If the name of the trace structure is decorated with primes and/or subscripts, those decorations carry over to the implicitly defined quantities. For example, mentioning a trace structure T_1' implicitly defines a signature γ_1' and P_1'. This, as described in note 11, also implicitly defines I_1', O_1' and A_1'.

Definition 14. If $\mathcal{C}_C = (\mathcal{B}_C, \mathit{proj}, \mathit{rename})$ is a trace algebra over W and $\mathcal{T}$ is a subset of the trace structures over $\mathcal{C}_C$, then $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ is a *trace structure algebra* iff the domain $\mathcal{T}$ is closed under the following operations on trace structures: parallel composition (def. 15), projection (def. 16) and renaming (def. 17).

We use the subscript C in $\mathcal{A}_C$ to denote a trace structure algebra that is built from a trace algebra $\mathcal{C}_C$ that has only complete traces (no partial traces).

To complete the definition of trace structure algebra, we need to define the operations on trace structures mentioned in definition 14.

Definition 15. If $O \cap O' = \emptyset$, then $T'' = T \parallel T'$ is defined and

$$\begin{aligned} \gamma'' &= ((I \cup I') - (O \cup O'),\ O \cup O') \\ P'' &= \{x \in \mathcal{B}_C(A'') : \mathit{proj}(A)(x) \in P \wedge \\ &\qquad \mathit{proj}(A')(x) \in P'\}. \end{aligned}$$

Definition 16. If $I \subseteq B \subseteq A$, then

$$\mathit{proj}(B)(T) = ((I, O \cap B), \mathit{proj}(B)(P)).$$

Definition 17. If r is a renaming function with domain A, then

$$\mathit{rename}(r)(T) = ((r(I), r(O)), \mathit{rename}(r)(P)).$$

It can be shown, using the axioms of trace algebra, that the operations of parallel composition, projection and renaming on trace structures satisfy the following identities. In all of the identities, there is an implicit assumption that the left hand side of the equation is defined; in each case, if the left hand side is defined, then so is the right hand side.

$$(T \parallel T') \parallel T'' = T \parallel (T' \parallel T'').$$

$$T \parallel T' = T' \parallel T.$$

$$\mathit{rename}(r)(\mathit{rename}(r')(T)) = \mathit{rename}(r \circ r')(T).$$

$$\begin{aligned} &\mathit{rename}(r)(T \parallel T') = \\ &\quad \mathit{rename}(r\mid_{A \to r(A)})(T) \parallel \\ &\quad \mathit{rename}(r\mid_{A' \to r(A')})(T'). \end{aligned}$$

$\mathit{rename}(\mathit{id}_A)(T) = T$,
where id_A is the identity function with domain A.

$$\mathit{proj}(B)(\mathit{proj}(B')(T)) = \mathit{proj}(B)(T).$$

$$\mathit{proj}(A)(T) = T.$$

$$\begin{aligned} &\mathit{proj}(B)(T \parallel T') = \\ &\quad \mathit{proj}(B \cap A)(T) \parallel \mathit{proj}(B \cap A')(T'), \\ &\quad \text{if } (A \cap A') \subseteq B. \end{aligned}$$

$$\begin{aligned} &\mathit{proj}(r(B))(\mathit{rename}(r)(T)) = \\ &\quad \mathit{rename}(r\mid_{B \to r(B)})(\mathit{proj}(B)(T)). \end{aligned}$$

We want to use trace structure algebras as the basis for our approach, which requires defining what it means for an "implementation to satisfy a specification" when both are given by trace structures. Our notion of satisfaction is based on trace set containment: an implementation satisfies a specification iff it is *contained* by the specification.

Definition 18. We say $T \subseteq T'$ (read T is contained in T') iff $\gamma = \gamma'$ and $P \subseteq P'$.

The operations of parallel composition, renaming and projection are monotonic with respect to trace structure containment, *e.g.,*

$$T \subseteq T' \Rightarrow \mathit{proj}(B)(T) \subseteq \mathit{proj}(B)(T').$$

The monotonicity of parallel composition is important for using trace structure algebras as a basis for hierarchical verification techniques.

4.1 Examples

Let us consider how some of the example trace algebras discussed in section 3.1 can be used to construct trace structures, and how the different definitions of projection on traces lead to different notions of parallel composition of trace structures.

Consider trace structures over the trace algebra $\mathcal{C}_C^I$. The set of possible traces of a trace structure with alphabet A is a subset of $\mathcal{B}_C(A)$, which in this case is A^∞. Consider the trace structures

$$\begin{aligned} T &= ((\{a,b\}, \emptyset), \{abab\}) \\ T' &= ((\{b,c\}, \emptyset), \{bcb\}). \end{aligned}$$

By the definition of parallel composition in a trace structure algebra, the set of possible traces of $T'' = T \parallel T'$ is

$$\begin{aligned} P'' &= \{x \in \mathcal{B}_C(\{a,b,c\}) : \mathit{proj}(\{a,b\})(x) \in P \wedge \\ &\qquad \mathit{proj}(\{b,c\})(x) \in P'\} \\ &= \{abacb, abcab\}. \end{aligned}$$

This example illustrates how parallel composition results in nondeterminism in this model.

However, parallel composition does not lead to nondeterminism when the underlying trace algebra is the one with $\mathcal{B}_C(A) = (2^A)^\omega$ described in section 3.1. Let

$$\begin{aligned} T &= ((\{a,b\}, \emptyset), \{\langle\{a,b\}, \{a\}, \{b\}\rangle\}) \\ T' &= ((\{b,c\}, \emptyset), \{\langle\{b\}, \{c\}, \{b\}\rangle\}) \end{aligned}$$

Here the set of possible traces of $T'' = T \parallel T'$ is the singleton set

$$P'' = \{\langle\{a,b\}, \{a,c\}, \{b\}\rangle\}.$$

The relevant difference between this model and the interleaving model is that here each trace provides more information about the time of occurrence of events. As a result, the order of events is fully determined when "merging" together two local traces to form a global trace of a composition. Global traces are also fully determined in the cases where traces over an alphabet A are elements of $2^{A \times \Re^{\not -}}$, $(A \mapsto V)^\omega$ or $\Re^{\not -} \mapsto (A \mapsto V)$.

Another case where parallel composition does lead to nondeterminism is the one described in section 3.1 where $\mathcal{B}_C(A) = (2^A - \{\emptyset\})^\infty$. In this case, for T and T' defined as above, the set of possible traces of $T'' = T \parallel T'$ is

$$\begin{aligned} P'' = \{&\langle\{a,b\}, \{a\}, \{c\}, \{b\}\rangle, \\ &\langle\{a,b\}, \{a,c\}, \{b\}\rangle, \\ &\langle\{a,b\}, \{c\}, \{a\}, \{b\}\rangle\}. \end{aligned}$$

The definitions of $\mathcal{C}_C^K$, $\mathcal{C}_C^S$, $\mathcal{C}_C^D$ also lead to the definition of the corresponding trace structure algebras $\mathcal{A}_C^K$, $\mathcal{A}_C^S$, $\mathcal{A}_C^D$. The definition of the projection operation induces the definition of parallel composition; in the case of $\mathcal{A}_C^K$ the definitions ensure that the streams of connected Kahn Processes are the same; for synchronous models they ensure that composition is truly synchronous; and for discrete time models that all processes share the same notion of time.

In addition we might want to impose constraints on the set of trace structures that we consider in the trace structure algebra. For example, we could restrict our model of Kahn Process Networks to only those processes that are *functional* in the sense that the same output sequences is obtained from the same input sequences. Note that in our definition of trace algebra the alphabet A doesn't distinguish between inputs and outputs. Once this distinction is enacted, it is easy to define functional processes by requiring that if two trace structures T and T' have the same input traces, then $T = T'$. Likewise for the synchronous model: in case we want to represent only finite state systems, we might require that only the regular sequences be considered and that the output be determinate once the input is.

4.2 Constructing Trace Structure Algebras

In a trace structure algebra $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$, the set of trace structures $\mathcal{T}$ is required to be closed under the operations on trace structures. This section states three theorems that make it easier to prove closure.

The first theorem states that if $\mathcal{T}$ is equal to the set of all trace structures over $\mathcal{C}_C$, then $\mathcal{T}$ is closed under the operations on trace structures, so $\mathcal{A}_C$ is a trace structure algebra. Recall that the alphabet of a trace structure need not be a finite set. The second theorem shows that the set of all trace structures with finite alphabets is closed under the operations on trace structures.

0-7695-1071-X/01 $10.00 © 2001 IEEE

For the third theorem, let $(\mathcal{C}_C, \mathcal{T})$ be a trace structure algebra, where $\mathcal{T}$ is some subset of the set of trace structures over $\mathcal{C}_C$. For every alphabet B, let $\mathcal{L}(B)$ be a class of sets of complete traces over B, that is, $\mathcal{L}(B) \subseteq 2^{\mathcal{B}_C(B)}$. Assume that $\mathcal{L}$ is closed under intersection, renaming, projection and "inverse projection" (this is formalized below). Let $\mathcal{T}'$ be the set of trace structures $(\gamma, P) \in \mathcal{T}$ such that P is in $\mathcal{L}(A)$. Then $\mathcal{T}'$ is closed under the operations on trace structures, so $(\mathcal{C}_C, \mathcal{T}')$ is a trace structure algebra.

Let $\mathcal{T}^I$ be the set of all trace structures over $\mathcal{C}_C^I$. By the first theorem, $\mathcal{A}_C^I = (\mathcal{C}_C^I, \mathcal{T}^I)$ is a trace structure algebra. Let $\mathcal{T}^{IR}$ be the set of all trace structures (γ, P) over $\mathcal{C}_C^I$ for which γ has a finite alphabet and P is a mixed regular set of sequences (that is, P is the union of a regular set and an ω-regular set). By the second and third theorems, $\mathcal{A}_C^{IR} = (\mathcal{C}_C^I, \mathcal{T}^{IR})$ is also a trace structure algebra.

The remainder of this section formalizes these results.

Theorem 19. If $\mathcal{C}_C$ is a trace algebra and $\mathcal{T}$ is the set of all of the trace structures over $\mathcal{C}_C$, then $\mathcal{T}$ is closed under the operations on trace structures, so $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ is a trace structure algebra.

Theorem 20. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ be a trace structure algebra. Let $\mathcal{T}'$ be the set of trace structures $T \in \mathcal{T}$ such that the alphabet of T is a finite set. Then $\mathcal{A}'_C = (\mathcal{C}_C, \mathcal{T}')$ is a trace structure algebra.

Definition 21. Let $\mathcal{T}$ be a set of trace structure over some trace algebra $\mathcal{C}_C$. The set of *alphabets of* $\mathcal{T}$ is the set of alphabets A of a signature γ in the set

$$\{\gamma : \exists P[(\gamma, P) \in \mathcal{T}]\}.$$

Theorem 22. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ be a trace structure algebra. For every alphabet B of $\mathcal{T}$, let $\mathcal{L}(B)$ be a subset of $2^{\mathcal{B}_C(B)}$. Let $\mathcal{T}'$ be the set of trace structures $T \in \mathcal{T}$ such that P is in $\mathcal{L}(A)$. Then $\mathcal{A}'_C = (\mathcal{C}_C, \mathcal{T}')$ is a trace structure algebra if the following requirements are satisfied for every alphabet B of $\mathcal{T}$.

L1. $\mathcal{L}(B)$ is closed under intersection.

L2. If $B' \subseteq B$ and $X \in \mathcal{L}(B)$, then $\mathit{proj}(B')(X) \in \mathcal{L}(B')$.

L3. If $B \subseteq B'$ and $X \in \mathcal{L}(B)$, then

$$\{x \in \mathcal{B}_C(B') : \mathit{proj}(B)(x) \in X\} \in \mathcal{L}(B').$$

L4. If r is a renaming function with domain B and $X \in \mathcal{L}(B)$, then $\mathit{rename}(r)(X) \in \mathcal{L}(r(B))$.

Definition 23. We define $\mathcal{A}_C^I$ to be the pair $(\mathcal{C}_C^I, \mathcal{T}^I)$, where $\mathcal{T}^I$ is the set of all trace structures over $\mathcal{C}_C^I$. By theorem 19, $\mathcal{A}_C^I$ is a trace structure algebra.

Definition 24. We define $\mathcal{T}^{IR}$ to be the set of all trace structures $T = (\gamma, P)$ over $\mathcal{C}_C^I$ for which γ has a finite alphabet and P is a mixed regular set of sequences. Also, $\mathcal{A}_C^{IR}$ is the ordered pair $(\mathcal{C}_C^I, \mathcal{T}^{IR})$. By theorem 20 and theorem 22, $\mathcal{A}_C^{IR}$ is a trace structure algebra.

5 Conservative Approximations

A conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ is an ordered pair $\Psi = (\Psi_l, \Psi_u)$, where Ψ_l and Ψ_u are functions from $\mathcal{T}$ to $\mathcal{T}'$. For a given trace structure T in $\mathcal{A}_C$, the trace structure $\Psi_l(T)$ is a kind of lower bound of T, while $\Psi_u(T)$ is an upper bound (relative to the '$\subseteq$' ordering on trace structures). Here we require that $\Psi_l(T)$ and $\Psi_u(T)$ have the same signature as T; it is also possible to allow conservative approximations that can change the signature of a trace structure, but that is beyond the scope of this paper.

As an example, consider the verification problem

$$\mathit{proj}(A)(T_1 \parallel T_2) \subseteq T,$$

where T_1, T_2 and T are trace structures in $\mathcal{T}$. This corresponds to checking whether an implementation consisting of two components T_1 and T_2 (along with some internal signals that are removed by the projection operation) satisfies the specification T. By definition, if Ψ is a conservative approximation, then showing

$$\mathit{proj}(A)(\Psi_u(T_1) \parallel \Psi_u(T_2)) \subseteq \Psi_l(T)$$

is sufficient to show that the original implementation satisfies its specification. Thus, the verification can be done in $\mathcal{A}'_C$, where it is presumably more efficient than in $\mathcal{A}_C$. A conservative approximation guarantees that doing the verification in this way will not lead to a false positive result, although false negatives are possible depending on how the approximation is chosen. The following definition formalizes the notion of a conservative approximation.

Definition 25. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras, and let Ψ_l and Ψ_u be functions from $\mathcal{T}$ to $\mathcal{T}'$. We say $\Psi = (\Psi_l, \Psi_u)$ is a *conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$* iff the following conditions are satisfied.

- For all $T \in \mathcal{T}$, the signature of $\Psi_l(T)$ and $\Psi_u(T)$ is γ.
- Let E be an arbitrary expression potentially involving parallel composition, projection and renaming of trace structures in $\mathcal{T}$. Let E' be formed from E be replacing every instance of each trace structure T with $\Psi_u(T)$. If T_1 is a trace structure in $\mathcal{T}$, and $E' \subseteq \Psi_l(T_1)$, then $E \subseteq T_1$.

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

The simplest example of a conservative approximation is $\Psi = (\Psi_l, \Psi_u)$ is

$$\begin{aligned}\Psi_l(T) &= (\gamma, \emptyset)\\ \Psi_u(T) &= (\gamma, \mathcal{B}'_C(A)).\end{aligned}$$

This definition of Ψ clearly satisfies the first condition of definition 25. To see that it satisfies the second condition, notice that the set of possible traces of E' and $\Psi_l(T_1)$ will be the universal set and the empty set, respectively; thus, it is never true that $E' \subseteq \Psi_l(T_1)$. This particular conservative approximation is not useful, however, because it always leads to a negative verification result; it cannot be used to show that an implementation satisfies a specification. In section 5.2, we will show how a conservative approximation can be constructed using a homomorphism from one trace algebra to another.

The remainder of this section states theorems that provide sufficient conditions for showing that some Ψ is a conservative approximation. The first theorem can be understood by recalling the example verification problem described above, and by considering the following chain of implications:

$$\begin{aligned}
&\mathit{proj}(A)(\Psi_u(T_1) \parallel \Psi_u(T_2)) \subseteq \Psi_l(T)\\
&\quad \text{assuming } \Psi_u(T_1 \parallel T_2) \subseteq \Psi_u(T_1) \parallel \Psi_u(T_2)\\
&\quad \Rightarrow \quad \mathit{proj}(A)(\Psi_u(T_1 \parallel T_2)) \subseteq \Psi_l(T)\\
&\quad \text{assuming } \Psi_u(\mathit{proj}(A)(T')) \subseteq \mathit{proj}(A)(\Psi_u(T'))\\
&\quad \Rightarrow \quad \Psi_u(\mathit{proj}(A)(T_1 \parallel T_2)) \subseteq \Psi_l(T)\\
&\quad \text{assuming } \Psi_u(T') \subseteq \Psi_l(T) \text{ implies } T' \subseteq T\\
&\quad \Rightarrow \quad \mathit{proj}(A)(T_1 \parallel T_2) \subseteq T.
\end{aligned}$$

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

In addition, we show that if $\Psi' = (\Psi'_l, \Psi'_u)$ provides looser lower and upper bounds than a conservative approximation Ψ (*i.e.,* $\Psi'_l(T) \subseteq \Psi_l(T)$ and $\Psi_u(T) \subseteq \Psi'_u(T)$ for all T), then Ψ' is also a conservative approximation. Also, the functional composition of two conservative approximations yields another conservative approximation.

Theorem 26. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras, and let Ψ_l and Ψ_u be functions from $\mathcal{T}$ to $\mathcal{T}'$. Assume that for all $T \in \mathcal{T}$, the signature of $\Psi_l(T)$ and $\Psi_u(T)$ is γ. If the following propositions A1 through A4 are satisfied for all trace structures T, T_1 and T_2 in $\mathcal{T}$, then Ψ is a conservative approximation.

A1. $\Psi_u(T_1 \parallel T_2) \subseteq \Psi_u(T_1) \parallel \Psi_u(T_2)$.

A2. $\Psi_u(\mathit{proj}(B)(T)) \subseteq \mathit{proj}(B)(\Psi_u(T))$.

A3. $\Psi_u(\mathit{rename}(r)(T)) \subseteq \mathit{rename}(r)(\Psi_u(T))$.

A4. If $\Psi_u(T_1) \subseteq \Psi_l(T_2)$, then $T_1 \subseteq T_2$.

Theorem 27. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras, and let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$. If $\Psi' = (\Psi'_l, \Psi'_u)$ is such that $\Psi'_l(T) \subseteq \Psi_l(T)$ and $\Psi_u(T) \subseteq \Psi'_u(T)$ for all $T \in \mathcal{T}$, then Ψ' is a conservative approximation.

Theorem 28. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$, $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ and $\mathcal{A}''_C = (\mathcal{C}''_C, \mathcal{T}'')$ be trace structure algebras. Also, let $\Psi = (\Psi_l, \Psi_u)$ and $\Psi' = (\Psi'_l, \Psi'_u)$ be conservative approximations from $\mathcal{A}_C$ to $\mathcal{A}'_C$ and from $\mathcal{A}'_C$ to $\mathcal{A}''_C$, respectively. Then $\Psi'' = (\Psi''_l, \Psi''_u)$ is a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}''_C$, where

$$\begin{aligned}\Psi''_l(T) &= \Psi'_l(\Psi_l(T))\\ \Psi''_u(T) &= \Psi'_u(\Psi_u(T)).\end{aligned}$$

5.1 Homomorphisms on Trace Algebras

We can define the notions of homomorphisms and isomorphisms between trace algebras. A homomorphism commutes with *rename* and *proj*; also, if x is a trace with alphabet A, then a homomorphism maps x to a trace with alphabet A. Thus, our definition of a homomorphism is quite standard. We will show in the next section how homomorphisms can be used to construct conservative approximations. An isomorphism is a homomorphism that is also a bijection. It is also possible to allow homomorphisms that can change the alphabet of a trace, but that is beyond the scope of this paper.

Definition 29. Let $\mathcal{C}_C$ and $\mathcal{C}'_C$ be trace algebras. Let h be a function from $\mathcal{B}_C$ to $\mathcal{B}'_C$ such that for all alphabets A, if $x \in \mathcal{B}_C(A)$, then $h(x) \in \mathcal{B}'_C(A)$. The function h is a *homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$* iff

$$\begin{aligned}h(\mathit{rename}(r)(x)) &= \mathit{rename}(r)(h(x)),\\ h(\mathit{proj}(B)(x)) &= \mathit{proj}(B)(h(x)).\end{aligned}$$

Here is a simple example of a homomorphism between trace algebras. It involves two of the trace algebras described in section 3.1. For all alphabets A, let h map traces in A^∞ to traces in 2^A such that

$$h(x) = \{a : \exists n\,[a = x(n)]\}.$$

It is easy to show that h is a homomorphism. Applying h to a trace abstracts away information about the order of events; all that remains is the set of actions that occurred one or more times.

Definition 30. A homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$ is an *isomorphism* iff it is a bijection. $\mathcal{C}_C$ are $\mathcal{C}'_C$ *isomorphic* iff there exists an isomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$.

Clearly if h is an isomorphism, then so is h^{-1}. Also, an isomorphism on trace algebras induces an isomorphism on trace structure algebras, as follows.

Corollary 31. Let h be an isomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras such that

$$\begin{aligned} (\gamma, P) \in \mathcal{T} &\Rightarrow (\gamma, h(P)) \in \mathcal{T}' \\ (\gamma, P') \in \mathcal{T}' &\Rightarrow \exists (\gamma, P) \in \mathcal{T}[P' = h(P)]. \end{aligned}$$

Then $\mathcal{A}_C$ and $\mathcal{A}'_C$ are isomorphic.

As examples of homomorphisms between trace algebras we show how to use the discrete time model $\mathcal{C}^D_C$ as a common refinement for both the Kahn Process Networks model $\mathcal{C}^K_C$ and the synchronous model $\mathcal{C}^S_C$.

Let's first consider the abstraction from discrete time processes to synchronous processes. In discrete time, we must first define how the sampled time relates to the interval in the sequences of the synchronous model. To do that, the discrete time models needs a notion "clock tick": for example, we might introduce an additional signal (the clock) in the signature of the process that can assume only two values, denoted $\top$ and $\bot$; we interpret these two values to indicate whether the time stamp corresponds to a synchronous instant, or whether the corresponding value should be discarded because it represents a non-visible behavior between synchronous instants. Then the homomorphism between the traces can be defined as the function that takes a trace $\mathcal{N}^{\neq} \mapsto (\overline{A} \mapsto V)$ in the extended signature $\overline{A}$ (that includes the clock signal) into the trace $(\overline{A} \mapsto V)^{\infty}$ where the additional signal (which, as noted in the definition of $\mathcal{C}^S_C$, is always present) is empty and the sequence is constructed by only considering the instants in the discrete model where the clock signal is (for example) $\top$. We denote with h' the function that takes a discrete time trace and produces the corresponding synchronous trace. It is also easy to show that this function preserves the operation of projection and renaming. Figure 3 shows an example of the application of this abstraction.

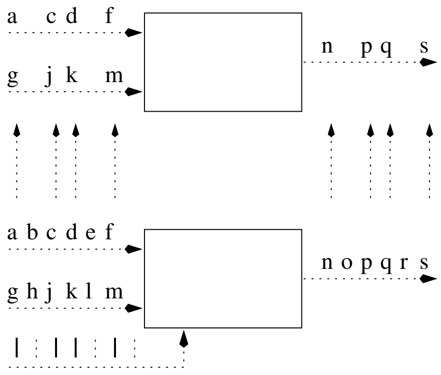

Figure 3. Abstraction of discrete into synchronous time

Let's now consider the correspondence between discrete time traces and Kahn Process traces. Here the discrete time model needs a notion of arrival of a new piece of data. In this example we will take the convention that a new piece of data (or token) arrives whenever the value on a particular signal changes. In this way the value itself encodes the information that a new element of a stream is present at the input (or is generated at the output). Here the advancement of the computation (the addition of a new token in a stream) is not centrally regulated but depends on the individual signals. We can therefore define a mapping that takes a trace $\mathcal{N}^{\neq} \mapsto (A \mapsto V)$, call this function $m(i, a)$, into the trace $A \mapsto V^{\infty}$, call this $f(a)$, where for each $a \in A$, the function $f(a)$ is obtained by dropping the repeated values of the corresponding sequence $m(i, a)$ over the index variable i. We denote with h'' the function that takes a discrete time trace and produces the corresponding trace in the Kahn Process model. Again, it is also easy to show that this function preserves the operation of projection and renaming. An example is shown in Figure 4.

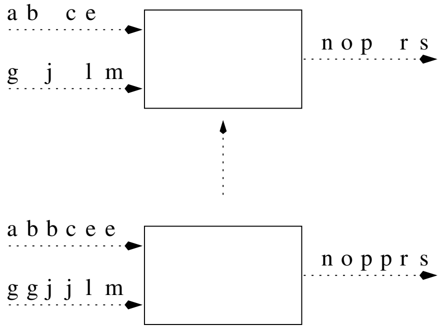

Figure 4. Abstraction of discrete into Kahn Processes

The operations defined in this section are not the only possible abstractions of discrete time into synchronous or Kahn Process models. Indeed, we could have defined different mappings with different properties. As we will see later, different mappings give rise to different notion of compositions.

5.2 Homomorphisms and Approximations

Let h be a trace algebra homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$, and let x and x' be traces in $\mathcal{C}_C$ and $\mathcal{C}'_C$, respectively, such that $h(x) = x'$. Intuitively, the trace x' is an abstraction of any trace y such that $h(y) = x'$. Thus, x' can be thought of as representing the set of all such y. Similarly, a set X' of traces in $\mathcal{C}'_C$ can be thought of as representing the largest set Y such that $h(Y) = X'$, where h is naturally extended to sets of traces. If $h(X) = X'$, then $X \subseteq Y$, so X' represents a kind of upper bound on the set X. This motivates using the function Ψ_u such that

$$\Psi_u(T) = (\gamma, h(P))$$

as the upper bound in a conservative approximation from a trace structure algebra over $\mathcal{C}_C$ to a trace structure algebra over $\mathcal{C}'_C$. A sufficient condition for a corresponding lower bound is: if $x \notin P$, then $h(x)$ is not in the set of possible traces of $\Psi_l(T)$. This leads to the definition

$$\Psi_l(T) = (\gamma, h(P) - h(\mathcal{B}_C(A) - P)).$$

The conservative approximation $\Psi = (\Psi_l, \Psi_u)$ is an example of a *conservative approximation induced by h,* which is formalized in the definition below using a slightly tighter lower bound for Ψ_l. Using this concept, if one proves that h is a homomorphism between two trace algebras (which is often quite easy), then one obtains a conservative approximation between trace structures with no additional effort. A conservative approximation induced by a homomorphism h is closely related to homomorphisms on ω-automata [8].

Definition 32. Let h be a homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$, and let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras. We naturally extend h to sets of traces. Assume Ψ_u and Ψ_l are functions from $\mathcal{T}$ to $\mathcal{T}'$ such that

$$\begin{aligned}\Psi_u(T) &\supseteq (\gamma, h(P))\\ \Psi_l(T) &\subseteq (\gamma, h(P) - h(Y - P)),\end{aligned}$$

where Y is the union of the $X \subseteq \mathcal{B}_C(A)$ such that

$$(\gamma, X) \in \mathcal{T} \wedge h(X) \subseteq h(P).$$

It can be shown that $\Psi = (\Psi_l, \Psi_u)$ is a conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$, which we call a *conservative approximation induced by h from $\mathcal{A}_C$ to $\mathcal{A}'_C$.* If the two set inequalities above are replaced by equalities, then Ψ is called the *tightest* conservative approximation induced by h from $\mathcal{A}_C$ to $\mathcal{A}'_C$.

Notice that $h(P) - h(\mathcal{B}_C(A) - P)$ is a subset of $h(P) - h(Y - P)$, so

$$\begin{aligned}\Psi_u(T) &= (\gamma, h(P))\\ \Psi_l(T) &= (\gamma, h(P) - h(\mathcal{B}_C(A) - P))\end{aligned}$$

(as described at the beginning of this section) is an example of a conservative approximation induced by h. This conservative approximation is independent of $\mathcal{T}$; the tightest conservative approximation induced by h depends on both h and $\mathcal{T}$.

Definition 32 defines both the class of conservative approximations induced by a homomorphism h and a distinguished approximation in that class, which we call the tightest conservative approximation induced by h. It is obvious that this distinguished approximation is in fact the tightest approximation within the class we defined. That is, if Ψ is the tightest conservative approximation induced by h and Ψ' is any conservative approximation in induced by h, then $\Psi'_l(T) \subseteq \Psi_l(T)$ and $\Psi_u(T) \subseteq \Psi'_u(T)$ for any trace structure T.

However, it is not immediately clear that class of approximations we defined includes all conservative approximations that might intuitively be "induced" by h. If there is a larger class of conservative approximations "induced" by h, then it might include an approximation that is tighter then the tightest one given in definition 32. We provide evidence that this is not the case in section 5.3, where we consider the *inverse* of a conservative approximation. This result depends on the particular set Y used in definition 32, and would not be true if we replaced Y by a simpler expression such as $\mathcal{B}_C(A)$.

A trace x is in Y iff it is contained in a trace structure $T_1 \in \mathcal{T}$ such that $h(P_1) \subseteq h(P)$. If $T_1 \nsubseteq T$, it is required that $\Psi_u(T_1) \nsubseteq \Psi_l(T)$. In this case, there exists a trace $x \in P_1 - P$, which implies $x \in Y - P$. Thus, $h(x)$ is in $\Psi_u(T_1)$ but not in $\Psi_l(T)$, so the requirement is satisfied. On the other hand, if there is no $T_1 \in \mathcal{T}$ such that $h(P_1) \subseteq h(P)$ and $T_1 \nsubseteq T$, then $Y = P$. In this case, $\Psi_l(T) = \Psi_u(T)$, showing that the particular definition of Y in definition 32 makes Ψ a tighter conservative approximation than it would otherwise be.

It is straightforward to take the general notion of a conservative approximation induced by a homomorphism, and apply it to specific models. Simply construct trace algebras $\mathcal{C}$ and $\mathcal{C}'$, and a homomorphism h from $\mathcal{C}$ to $\mathcal{C}'$. Recall that these trace algebras act as models of individual behaviors. Using the results described so far in this section (without any additional proofs), one can construct the trace structure algebras $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ and $\mathcal{A}' = (\mathcal{C}', \mathcal{T}')$, and a conservative approximation Ψ induced by h (where $\mathcal{T}$ and $\mathcal{T}'$ are the sets of all trace structures over $\mathcal{C}$ and $\mathcal{C}'$, respectively). Thus, one need only construct two models of individual behaviors and a homomorphism between them to obtain two trace structure models along with a conservative approximation between the trace structure models.

5.3 Inverses of Conservative Approximations

Let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. Let $T \in \mathcal{T}$ and $T' \in \mathcal{T}'$ be such that $T' = \Psi_u(T)$. As we have discussed, T' represents a kind of upper bound on T. It is natural to ask whether there is a trace structure in $\mathcal{T}$ that is represented exactly by T' rather than just being bounded by T'. If no trace structure in $\mathcal{T}$ can be represented exactly, then Ψ is abstracting away too much information to be of much use. If every trace structure in $\mathcal{T}$ can be represented exactly, then Ψ_l and Ψ_u are equal and are isomorphisms from $\mathcal{A}_C$ to $\mathcal{A}'_C$. These extreme cases illustrate that the amount of abstraction in Ψ is related to what trace structures T are represented exactly by $\Psi_u(T)$ and $\Psi_l(T)$.

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 $T \in \mathcal{T}$ for which $\Psi_l(T)$ and $\Psi_u(T)$ have the same value, call it T'. Intuitively, T' represents T exactly in this case; the key property of the inverse of Ψ (written Ψ_{inv}) is that $\Psi_{inv}(T') = T$. If $\Psi_l(T) \neq \Psi_u(T)$, then T is not represented exactly in $\mathcal{A}'_C$. In this case, T is not in the image of Ψ_{inv}. Characterizing when $\Psi_{inv}(T')$ is defined (and what its value is) helps to show what trace structures in $\mathcal{T}$ can be represented exactly (not just conservatively) by trace structures in $\mathcal{T}'$. The remainder of this section formalizes the idea of the inverse of a conservative approximation, and characterizes the inverse of the tightest conservative approximation induced by a homomorphism h.

Lemma 33. Let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. For every $T' \in \mathcal{T}'$, there is at most one $T \in \mathcal{T}$ such that $\Psi_l(T) = T'$ and $\Psi_u(T) = T'$.

Definition 34. Let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. Let $\mathcal{T}_1$ be the set of $T \in \mathcal{T}$ such that $\Psi_l(T) = \Psi_u(T)$. Let $\mathcal{T}'_1$ be the image of $\mathcal{T}_1$ under Ψ_l. The *inverse of* Ψ is the partial function Ψ_{inv} with domain $\mathcal{T}'$ and codomain $\mathcal{T}$ that is defined for all $T' \in \mathcal{T}'_1$ so that $\Psi_{inv}(T') = T$, where T is the unique (by lemma 33 and the definition of $\mathcal{T}'_1$) trace structure such that $\Psi_l(T) = T'$ and $\Psi_u(T) = T'$.

Theorem 35. Let h be a trace algebra homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$, and let $\Psi = (\Psi_l, \Psi_u)$ be the tightest conservative approximation induced by h from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. If $T' \in \mathcal{T}'$ is such that the set

$$Z = \{X \subseteq \mathcal{B}_C(A') : (\gamma', X) \in \mathcal{T} \wedge h(X) \subseteq P'\},$$

contains a unique maximal (by inclusion) element P for which $P' = h(P)$, then $\Psi_{inv}(T') = (\gamma', P)$; otherwise, $\Psi_{inv}(T')$ is undefined.

The above theorem completely characterizes the inverse of any tightest conservative approximation induced by a homomorphism h. The final theorem of this section specializes this result to trace structures algebras that are *closed under finite and infinite unions,* a property enjoyed by many of the trace structure algebras we consider. This specialization results in a simpler characterization of when Ψ_{inv} is defined. In particular, $\Psi_{inv}(T')$ is defined iff there exists a $T \in \mathcal{T}$ such that $\Psi_u(T) = T'$. This is a strong result. Clearly the existence of such a T is a necessary condition for the inverse of any conservative approximation to be defined on T'; when $\mathcal{T}$ is closed under finite and infinite unions, and Ψ is the tightest conservative approximation induced by a homomorphism, it is also a sufficient condition.

Definition 36. Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ be a trace structure algebra. We say $\mathcal{A}_C$ is *closed under finite (infinite) unions* iff for every signature γ the set

$$\{P \subseteq \mathcal{B}_C(A) : (\gamma, P) \in \mathcal{T}\}$$

is closed under finite (infinite) unions.

Theorem 37. Let h be a trace algebra homomorphism from $\mathcal{C}_C$ to $\mathcal{C}'_C$, and let $\Psi = (\Psi_l, \Psi_u)$ be the tightest conservative approximation induced by h from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$. Assume $\mathcal{A}_C$ is closed under finite and infinite unions. If $T' \in \mathcal{T}'$ is such that $\Psi_u(T) = T'$ for some $T \in \mathcal{T}$, then $\Psi_{inv}(T')$ is defined and its set of possible traces is

$$\bigcup\{X \subseteq \mathcal{B}_C(A') : (\gamma', X) \in \mathcal{T} \wedge h(X) \subseteq P'\};$$

otherwise, $\Psi_{inv}(T')$ is undefined.

6 Modeling Heterogeneous Systems

Our method for modeling heterogeneous systems makes use of the inverses of conservative approximations. If an inverse is not defined for a particular trace structure, then the lower level model is not adequate to represent the higher level. In addition, we must also insure that the inverse of the approximation preserves the operations of trace structure algebra.

Given agents in two different models of computation, a formal semantics of their parallel composition can be constructed as follows.

1. Construct trace algebras $\mathcal{C}'_C$ and $\mathcal{C}''_C$ (def. 5) appropriate for the two different models of computation. These algebras are models of individual behaviors (or executions), rather than agents. Thus, they should be relatively easy to construct.

2. Construct trace structure algebras $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ and $\mathcal{A}''_C = (\mathcal{C}''_C, \mathcal{T}'')$ (def. 14). These algebras are models of agents. The generic theorems of section 4.2 simplify their construction from the corresponding trace algebras $\mathcal{C}'_C$ and $\mathcal{C}''_C$.

3. Within the trace structure algebras $\mathcal{A}'_C$ and $\mathcal{A}''_C$, construct trace structures T' and T'' for the two agents. Notice that these first three steps are only necessary to bring the agent models into our framework. If the agent models were originally constructed within the framework as trace structures, then we can begin with step 4.

4. Construct a third trace algebra $\mathcal{C}_C$ that is at a lower level of abstraction than $\mathcal{C}'_C$ and $\mathcal{C}''_C$ and can serve as the basis for a unifying model of computation.

5. Use the theorems of section 4.2 to construct a trace structure algebra $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ from the trace algebra $\mathcal{C}_C$.

6. Construct trace algebra homomorphisms h' from $\mathcal{C}_C$ to $\mathcal{C}'_C$ and h'' from $\mathcal{C}_C$ to $\mathcal{C}''_C$ (def. 29).

7. The homomorphisms h' and h'' induce conservative approximations Ψ' and Ψ'' (def. 32), which map trace structures from $\mathcal{A}$ to $\mathcal{A}'$ and $\mathcal{A}''$, respectively.

8. The inverses of the above conservative approximations, Ψ'_{inv} and Ψ''_{inv} (def. 34), map trace structures from $\mathcal{A}'$ and $\mathcal{A}''$, respectively, to the unifying semantic domain $\mathcal{A}$. There, the parallel composition of the original agent models is

$$\Psi'_{inv}(T') \mid\mid \Psi''_{inv}(T'').$$

As an example we now consider the problem of composing two objects: a trace structure T' of the synchronous model $\mathcal{A}^S_C$ and a trace structure T'' of the Kahn Process model $\mathcal{A}^K_C$. Because these trace structures are drawn from different algebras we don't have a definition of a composition. And in fact, there are several different, and all valid, ways that a composition could be defined. We prefer to define this operation in terms of the common refined discrete time model where composition is defined. As noted above, a different refinement would lead to a different composition of the two models. We have already defined the homomorphism h' and h'' on trace algebras. They, in turn, induce conservative approximations Ψ' and Ψ'' whose inverses Ψ'_{inv} and Ψ''_{inv} will be used in the composition.

We proceed as follows. Let T_1 be the discrete time model such that $\Psi'_{inv}(T') = T_1$, and let T_2 be the discrete time model such that $\Psi''_{inv}(T'') = T_2$. Because T_1 and T_2 are both discrete time models, we can obtain their composition $V = T_1 \mid\mid T_2$.

We now want to reflect the effect of this composition back to the higher levels of abstraction. To do so, we consider the minimal $V_1 \subseteq T_1$ and $V_2 \subseteq T_2$ such that the composition of V_1 and V_2 is V (minimality is inteded with respect to trace containment). These objects represent the behaviors of T_1 and T_2 that are mutually compatible and concur in creating a behavior of the compound object V. In other words, V_1 and V_2 represent the behaviors of T_1 and T_2 as constrained by the composition. This procedure is represented graphically in Figure 5.

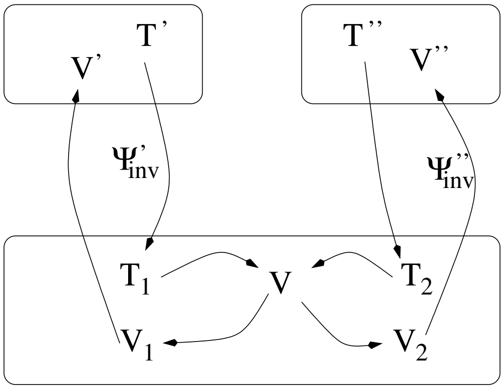

Figure 5. Example of composition

We can now abstract these two objects and obtain $\Psi'_u(V_1) = V' \subseteq T'$ and $\Psi''_u(V_2) = V'' \subseteq T''$. These objects represent at the higher level of abstraction the constrained behaviors that are due to the effect of the composition.

The result of applying this procedure is not to obtain a new compound object, as such an object might not be defined in either the synchronous or the Kahn Process domain alone. Rather we act as to 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.

It is instructive to note that a more direct relation between synchronous trace structures and Kahn Process trace structures could be obtained by observing that both models actually operate on sequences of values. The intuitive way of combining these two models would therefore be to convert each sequence in one model into a corresponding, identical, sequence in the other. This straighforward conversion however fails to acknowledge the fact that computation in the Kahn Process model advances on individual signals, while the synchronous model advances on all signals at the same time. The result is a different notion of composition. This is not to say that this different notion of composition is wrong or not valid; it shows that there are possibly many different ways of combining heterogeneous models that result in different underlying properties. In this section we have shown a methodology that derives a notion of composition that is consistent with the chosen notion of refinement to a common model.

The use of inverses of conservative approximation can also help us define what it means for one agent model to be a refinement of another. Within a single trace structure algebra, the relation of one agent implementing a specification is simply represented by trace containment. Let T' and T be trace structures in a trace structure algebra $\mathcal{A}_C$. Then T' is a refinement of T iff $T' \subseteq T$ (def. 18).

A more interesting case is when T' and T are in different trace structure algebras $\mathcal{A}'_C$ and $\mathcal{A}_C$, respectively. This can happen, for example, if $\mathcal{A}'_C$ is a functional model and $\mathcal{A}_C$ is a more detailed model used to introduce timing and/or power constraints. Given an inverse conservative approximation Ψ_{inv} from $\mathcal{A}'_C$ to $\mathcal{A}_C$, then T is a refinement of T' iff

$$T \subseteq \Psi_{inv}(T').$$

The simple example of composition shown in this section can be carried forward to include a set of different models. This generates a framework of models in which the notion of refinement through inverses of conservative approximations is used to determine the interaction of those that are not directly related. That is, if we can always find a common refinement. Because the relation of refinement is a partial order in this set, we can organize our models in their order of implementation details; by lifting this set with a model that can refine (perhaps transitively by going through other models) all other models, we satisfy that requirement and we can define a composition that is driven by the refinement to a common model for all pairs of models.

In addition it is also easy to provide a model that contains no information into which every other model could be abstracted (for example by employing the simple trace algebra that contains only one trace). By doing this we generate a lattice structure, as the one shown in Figure 6. In a lattice structure we can transitively apply the techniques presented above. For example, the composition driven by refinement can be carried out on models that can only indirectly be put in relation to a common refinement by recursively applying the transformation until the greatest lower bound is found.

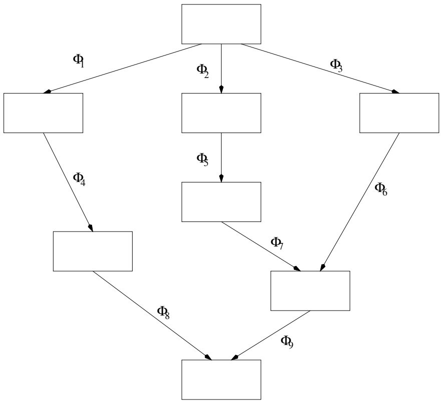

Figure 6. A lattice of models

7 Summary and Future Work

In our framework, each agent is represented by a *trace structure,* which is an ordered pair of a signature γ (the interface of the agent) and a set P of *possible traces.* Each trace in P represents a possible behavior of the agent. Both implementations and specifications are represented by trace structures. One trace structure satisfies the specification given by another trace structure iff the set of possible traces of the first is contained in the set of possible traces of the second. This notion of *trace set containment* is a generalization of standard verification techniques based on *language containment.*

The above description of trace structures does not say what kinds of mathematical objects are used as traces. In normal language containment methods, a trace is a finite or infinite sequence, so a set of traces is a formal language. We want to be much more general than this, because we do not want our use of trace structures to limit the kinds of models we can consider. On the other hand, we do not want to allow completely arbitrary traces because we want to have general theorems that are true of all trace structures (so the theorems do not have to be reproven every time a new class of trace structures is constructed).

We satisfy these constraints by using the idea of a *trace algebra.* A trace algebra (def. 5) is an abstract algebra with a set of traces as its domain, where each trace is interpreted as an abstraction of a physical behavior. There are two operations in a trace algebra: projection and renaming. These operations must satisfy axioms T1 through T8, the *axioms of trace algebra.* Other than these axioms, no other restrictions are placed on what kinds of mathematical objects can be used as traces in a trace algebra.

Once trace algebra is formalized, it is possible to formalize trace structures. The set of *trace structures* (def. 12) over a trace algebra $\mathcal{C}$ is the set of ordered pairs (γ, P), where γ is a signature and P is a subset of the traces of $\mathcal{C}$ with the same alphabet (*i.e.,* set of signals) as γ. A *trace structure algebra* is an ordered pair $\mathcal{A} = (\mathcal{C}, \mathcal{T})$, where $\mathcal{C}$ is a trace algebra and $\mathcal{T}$ is a subset of the set of trace structures over $\mathcal{C}$. The operations of parallel composition, projection and renaming are defined on trace structures in $\mathcal{T}$ using the operations of projection and renaming on individual traces in $\mathcal{C}$ (def. 15, def. 16 and def. 17). The set of trace structures $\mathcal{T}$ must be closed under these operations. The axioms of trace algebra are quite weak, but they are strong enough to guarantee that the operations on trace structures satisfy several useful identities.

Using these ideas to construct agent models only requires constructing a domain of traces, along with projection and renaming operations, and proving that they satisfy the axioms of trace algebra. A trace structure algebra can be constructed from the trace algebra without having to prove any additional theorems. Thus, our general results greatly simplify the task of constructing new agent models.

One of the uses of being able to easily build new agent models is to study the relationships between models that can be efficiently mechanized and models that accurately represent physical reality. Ideally, correctness proofs (of trace set containment) in the efficient model would be logically equivalent to correctness proofs in the accurate model, but this is rarely the case. The best we can usually do is to have correctness in the efficient model imply correctness in the accurate model. This is formalized by using a *conservative approximation* from the accurate model to the efficient model (def. 25). Let $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ and $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$ be trace structure algebras. A conservative approximation from $\mathcal{A}_C$ to $\mathcal{A}'_C$ is an ordered pair $\Psi =\Psi(\quad _l, \Psi_u)$, where Ψ_l and Ψ_u are functions from $\mathcal{T}$ to $\mathcal{T}'$. For a given trace structure T in $\mathcal{A}_C$, the trace structure $\Psi_l(T)$ is a kind of lower bound of T, while $\Psi_u(T)$ is an upper bound (relative to trace set containment). By definition, if a verification problem in $\mathcal{C}_C$ is converted into a verification problem in $\mathcal{C}'_C$ by applying a conservative approximation Ψ, then a correctness proof in the latter problem implies a correctness result in the former problem.

A general method for constructing conservative approximations involves *homomorphisms on trace algebras* (def. 29). A homomorphism from $\mathcal{C}$ to $\mathcal{C}'$ is just a function from the traces of $\mathcal{C}$ to the traces of $\mathcal{C}'$ that satisfies the standard homomorphism laws for the operations of trace algebra. A *conservative approximation induced by* h (def. 32) is a conservative approximation from $\mathcal{A}_C = (\mathcal{C}_C, \mathcal{T})$ to $\mathcal{A}'_C = (\mathcal{C}'_C, \mathcal{T}')$, for appropriate $\mathcal{T}$ and $\mathcal{T}'$. To take advantage of these results we need only construct the appropriate trace algebras and homomorphisms; the trace structure algebras and the conservative approximations are obtained without any additional effort.

We also showed how our framework can be used to model heterogeneous systems. Inverses of conservative approximations (def. 34) can be used to embed traces structures from two different models of computation into a unifying trace structure algebra, and to check refinement between different models of computation.

The most important area of future work for our framework is to apply it to more examples of models of computation. This includes not only different models of functional behavior, but also models of lower-level issues such as timing, power and noise. Conservative approximations should be generalized to allow for $\Psi_u(T)$ and $\Psi_l(T)$ to have different signatures than T. This makes it possible to remove signals that are not relevant for the more abstract agent model. Also, the signatures of trace structures should be generalized to allow explicit modeling of data types and bidirectional communication. More theoretical possible extensions include handling branching time models and integrating Dill's [6] approach for distinguishing *success* and *failure* traces.

References

[1] 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.

[2] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Aug. 1992. Technical Report CMU-CS-92-179.

[3] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. 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.

[4] J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. ojun Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong. Heterogeneous concurrent modeling and design in java. Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley, Mar. 1986.

[5] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[6] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[7] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, Mar. 1997.

[8] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.

[9] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 17(12):1217–1229, Dec. 1998.

[10] A. Mazurkiewicz. Basic notions of trace theory. In de Bakker et al. [5].

[11] V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, Feb. 1986.

[12] J. Rowson and A. Sangiovanni-Vincentelli. Felix initiative pursues new co-design methodology. *Electronic Engineering Times*, pages 50, 51, 74, June 1998.