Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002
©2002 Society for Desing and Process Science

Modeling Techniques in Design-by-Refinement Methodologies

Jerry R. Burch[1], Roberto Passerone[1], Alberto L. Sangiovanni-Vincentelli[2]

[1]Cadence Berkeley Laboratories, Berkeley CA 94704
[2]Department of EECS, University of California at Berkeley, Berkeley CA 94720

ABSTRACT: Embedded system design methodologies that are based on the effective use of multiple levels of abstraction hold promise for substantial productivity gains. Starting the design process at a high level of abstraction improves control over the design and facilitates verification and synthesis. In particular, if we use a rigorous approach to link the levels of abstraction, we can establish properties of lower levels from analysis at higher levels. This process goes by the name of "design by refinement". To maximize its benefit, design by refinement requires a formal semantic foundation that supports a wide range of levels of abstraction (e.g., from differential equations describing physical behaviors to high-level requirement specifications). We describe such a semantic foundation, and give examples of how it integrates several well-known models for reactive systems. The semantic foundation allows us to establish the relationships among the different levels of abstractions even when non-homogeneous models are used to describe the design. These relationships are essential to establish properties and to document assumptions about the way the models are used, preventing common errors that are difficult to detect and may otherwise require long redesign cycles.

I. 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 user interfaces. 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:

- 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.
- 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.
- 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 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.
- 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.

A great deal of excellent work has been done in a project centered around Ptolemy, where the issues related to the composition of models of computation and a general simulation framework have been carefully studied [9], [10]. The Ptolemy environment allows the efficient simulation and analysis of functional behavior of heterogeneous systems. In a companion project, we have advocated the introduction of rigorous methodologies for system-level design for years (e.g., [2], [20]) but we feel that there is still much to do. Recently we have directed our efforts to a new companion endeavor that tries to capture the requirements of present day embedded system design: the Metropolis project.

The Metropolis project ([3]) has the objective of providing a *design methodology and the software infrastructure to enable the design of embedded systems*. The focus of the project is in a complete design system for embedded systems from specifications to implementation using methodologies such as platform-based design, communication-based design and successive refinement. The focus is on formal analysis and synthesis techniques for *heterogeneous* embedded systems.

The word heterogeneous highlights the fact that in complex designs that interact with the real world, different parts of a design are likely to be modeled using very different techniques. For this reason, Metropolis is centered around its meta-model of computation, a set of primitives that can be used to construct several different models of computation. The long term objective of this work is to lay the foundations for providing a denotational semantics for the meta-model. Here we begin by studying the semantic domain of several of the models of computation of interest, and by studying how relationships between these models can be established. To do so, we have created a mathematical framework in which to express semantic domains in a form that is close to their natural formulation (i.e. the form that is most convenient for a given domain), and yet structured enough to give us results that apply regardless of the particular domain in question.

An important factor in the design of heterogeneous systems is the ability to flexibly use different levels of abstraction. Different abstractions are often employed for different parts of a design (by way of different models of computation, for instance). Even each individual piece of the design undergoes changes in the level of abstraction during the design process, as the model is refined towards a representation closer to the final implementation. Different levels of detail are also used to perform different kinds of analysis: for example, a high level functional verification versus a detailed electromagnetic interference analysis. Thus, we provide a mathematical framework that allows the user to choose the best abstraction for the particular task at hand, and to formally relate that abstraction to different abstractions used for other tasks. Here, we also recognize that abstraction may come in very different forms that include the model of computation, the scope or visibility of internal structures, or the model of the data.

In our work, we concentrate on semantic domains for concurrent systems and on the relations and functions over those domains. We also emphasize the relationships that can be constructed between different semantic domains. This work is therefore independent of the specific syntaxes and semantic functions employed. Likewise, we concentrate on a formulation that is convenient for reasoning about the properties of the domain. As a result, we do not emphasize finite representations or executable models, which we defer for future work.

The following sections introduce our framework. The exposition will focus on the definition of a natural semantic domain for a specific application, and its formulation in the framework. Later in the paper we consider examples of relationships that can we can build between some of the domains. These relationships are the basic tools to be used in studying the properties of heterogeneous systems.

II. Related Work

Several formal models have been proposed over the years (see e.g. [12]) to capture one or more aspects of computation as needed in embedded system design. Many models of computation can be encoded in the Tagged Signal Model [16]. However, because encoding is necessary, the level of abstraction is effectively changed, so some of the advantages of the original model of computation may be lost. In this work, in contrast, we describe a framework that is less restrictive than the Tagged Signal Model in terms of what can be used to represent behaviors, and we concentrate on building relationships between the models that fit in the framework.

The study of systems in which differnt parts are described using different models of computation (heterogeneous systems) is the central theme of the Ptolemy project [9], [10]. Our work shares the basic principles of providing flexible abstractions and an environment that supports a structured approach to heterogeneity. The approach, however, is quite different. In Ptolemy each model of computation is described operationally in terms of a common executable interface. For each model, a “director” determines the activations of the actors (for some models, the actors are always active and run in their own thread). Similarly, communication is defined in terms of a common interface. The director together with an implementation of the communication interface (a “receiver”) defines the communication scheme and the possible interactions with other models of computation. On the other hand, we base our framework on a denotational representation and deemphasize executability. Instead, we are more concerned with studying the process of abstraction and refinement in abstract terms. For example, it is easy in our framework to model the non-deterministic behavior that emerges when an abstract model is embedded into a more detailed model. Any executable framework would require an upfront choice that would make the model deterministic, potentially hiding some aspects of the composition.

There is a tradeoff between two goals: making the framework general, and providing structure to simplify constructing models and understanding their properties. While our framework is quite general, we have formalized several assumptions that must be satisfied by our domains of agents. These include both axioms and constructions that build process models (and mappings between them) from models of individual behaviors (and their mappings). These assumptions allow us to prove many generic theorems that apply to all semantic domains in our framework. In our experience, having these theorems greatly simplifies constructing new semantic domains that have the desired properties and relationships.

Process Spaces [17], [18] are an extremely general class of concurrency models. However, because of their generality, they do not provide much support for constructing new semantic domains or relationships between domains. For example, by proving generic properties of broad classes of conservative approximations, we remove the need to reprove these properties when a new conservative approximation is constructed.

We introduce the notion of a conservative approximation to relate one domain of agents to another, more abstract, domain. A conservative approximation has two functions. The first, called the lower bound, is used to abstract agents that represent the specification of a design. The second, called the upper bound, is used to abstract agents that represent possible implementations of the specification. A conservative approximation is defined so that if the implementation satisfies the specification in the abstract domain, then the implementation satisfies the specification in the more detailed domain, as well. Our notion of conservative approximation is closely related to the Galois connection of an abstract interpretations [6], [7]. In particular, the upper bound of a conservative approximation roughly corresponds to the abstraction function of a Galois connection. However, the lower bound of a conservative approximation appears to have no analog in the theory of abstract interpretations. To our knowledge, for abstract interpretations a positive verification result in the abstract domain implies a positive verification result in the concrete domain only if there is no loss of information when mapping the specification from the concrete domain to the abstract domain. Thus, conservative approximations allow non-trivial abstraction of both the implementation and the specification, while abstract interpretations only allow non-trivial abstraction of the implementation.

The ability to define domains of agents for different models of computation is also a central concept of the Rosetta language [1]. In Rosetta, a domain is described declaratively as a set of assertions in some higher order logic. Different domains can be obtained by extending a definition in a way similar to the sub-typing relation of a type system. Domains that are otherwise unrelated can be composed by constructing functions, called interactions, that (sometimes partially) express the consequences of the properties and quantities of one domain onto another. This process is particularly useful for expressing and keeping track of constraints during the refinement of the design. In contrast to Rosetta we are not concerned with the definition of a language. In fact, we define the domain directly as a collection of elements of a set, not as the model of a theory. In this sense, the approach taken by Rosetta seems more general. As already discussed, however, the restrictions that we impose on our models allow us to prove additional results that help create and compare the models. A detailed analysis of the relationships between the two frameworks is one of the topics of our current research.

In our framework we define a domain of agents that is suitable for describing the behavior of systems that have both continuous and discrete components. The term hybrid is often used to denote these systems. Many are the models that have been proposed to represent the behavior of hybrid systems. Most of them share the same view of the behavior as composed of a sequence of steps; each step is either a continuous evolution (a flow) or a discrete change (a jump). Different models vary in the way they represent the sequence. One example is the Masaccio model proposed by Henzinger et al. [13], [14]. In Masaccio the representation is based on components that communicate with other components through variables and locations. During an execution the flow of control transitions from one location to another according to a state diagram that is obtained by composing the components that constitute the system. Each transition in the state diagram models a jump or a flow of the system and constrains the input and output variables through a difference or differential equation. The underlying semantic model is based on sequences. legal jumps and flows that can be taken during the sequence of steps.

In our framework we talk about hybrid models in terms of the semantic domain only (which is based on functions of a real variable rather than sequences). This is a choice of emphasis: in Masaccio and other hybrid models the semantic domain is used to describe the behavior of a system which is otherwise represented by a transition diagram. In contrast, in our framework the semantic domain is the only concern and we seek results that are independent of the particular (finite state) representation that is used.

Another related concept that is found in models for hybrid systems is that of refinement. In our framework we must distinguish between two notions of refinement. The first is a notion of refinement within a semantic domain: in our framework this notion is based on pure trace containment, and is analogous to those defined in the majority of hybrid models. The second notion of refinement that is present in our framework has to do with changes in the semantic domain. This notion is embodied in the concept of conservative approximation that relates models at one level of abstraction to models at a different level of abstraction. There is no counterpart of this notion in the hybrid models.

III. Motivating Example

As already mentioned, we require our framework to support multiple models of computation during the design process. In this section we provide a motivating example for this requirement, which we will use throughout the paper. The example, shown in Figure 1, is an abstracted version of the PicoRadio project ([19]), developed at the Berkeley Wireless Research Center.

A PicoRadio is a node in a network that exchanges information with its neighboring nodes. Depending on the application, a PicoRadio may function as the intercom end of a communication system, or as a controller for a set of sensors and actuators. Whatever its function is, the PicoRadio must include several subsystems, as shown in Figure 1. Since communication with neighboring nodes occurs on a wireless link, a Radio Frequency (RF) subsystem is used to interface the design to the channel. Demodulation and decoding is done at the baseband level, after conversion from the high transmission frequency. The data streams obtained from the baseband is interpreted by a protocol stack, which feeds the application that ultimately interfaces with the user.

The design of such systems is complex, not so much in terms of their size, but because of the very stringent constraints on power and because of the intrinsic interactive nature of the nodes. Together, they call for a new design methodology and indeed, developing the new methodology

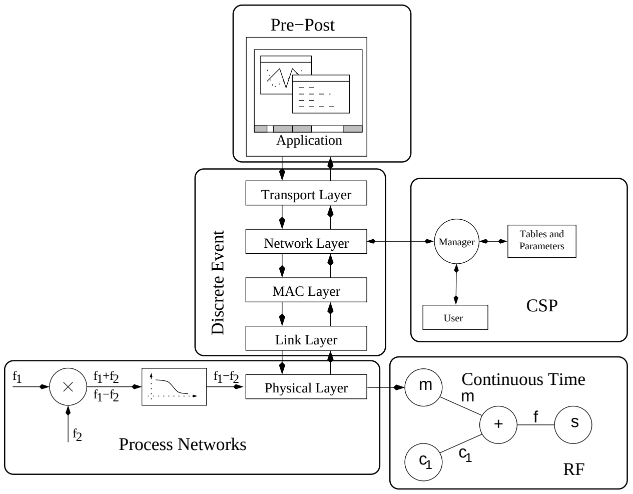

Fig. 1. Full system

was the primary task during the design of the first version of the Picoradio ([8]). Because power concerns are best attacked at the algorithmic level, new protocols are being devised whose primary purpose is to maximize the life-time of the system. Consequently, the interaction between the different subsystems becomes critical.

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

The interaction between different models of computation can be understood when the description of the models is embedded in the same unifying framework. Trace algebras is one such framework. The remaining sections of the paper discuss the details of the framework by examining appropriate models of computation for each of the subsystems above.

IV. Formal Framework

In this section and in those that follow we introduce our formal framework.

Given a model of computation, the task of building the corresponding model in the framework begins with the definition of a set of elements and some functions that operate on them, grouped together into an algebraic structure that we call *trace algebra*. We call each element a *trace*. Traces correspond intuitively to individual executions of agents in the given model of computation. Despite their name, traces are not limited to sequences, but, as we shall see in the examples, they can be any mathematical object that can be defined to satisfy the laws of the algebra. The functions, on the other hand, are analogous to the common operations that can be performed on the executions in the model of computation. The laws of the algebra ensure that the functions behave according to their intuitive sense.

From the trace algebra, we construct a second algebra whose elements are the agents of the model of computation, expressed primarily as a set of traces. We call each agent a *trace structure*, and the corresponding algebra a *trace structure algebra*. Likewise, we construct a set of functions on the agents that satisfy the laws of the algebra and represent operations such as composition and instantiation. The laws are again defined to ensure the proper interpretation of the operations.

For each model of computation we construct a trace algebra and a trace structure algebra. Because different trace algebras have the same structure in terms of the kind and number of operations, one can understand how one model relates to another by constructing mappings that preserve the application of the operation on traces. These mappings are then used to construct approximations (abstractions) on the agents. The purpose of the framework is to make it easy to construct the models and their relationships through the application of general results that derive from the algebraic structure.

In this section we present the basic concepts and definitions by way of an example, i.e. the formalization of a semantic domain suitable for the representation of behaviors in a model of computation that supports continuous time. We first present a formalization that can be considered *natural* for the domain of application, and then show how the same can be cast in terms of a general set of definitions.

A. Traces and Trace Structures

In this section we present both the basic concepts of our framework, and the example of the formalization of a semantic domain for a model of computation based on continuous time.

The model of computation that we have in mind relies on equations to express the relationships between the quantities that occur in the model. More specifically, we are interested in a model of computation where the quantities (variables) are functions over the set of reals. By convention, it is assumed that the set of reals represents time, and we talk about functions over time. Consequently, the equations we are interested in are relations on functions over time, and we denote the independent variable with the letter t.

Consider the following equation:

$$x = 3t. \tag{1}$$

This is an equation in the unknown x. Traditionally, the interpretation of the equation is done in terms of the set of possible solutions. In our case, the set consists of functions that are associated to the variable x. A function $x : \mathbb{R}^{\not=} \to \mathbb{R}$ is a solution of the equation if, when substituted for the unknown, the resulting relation is true. The notation $\mathbb{R}^{\not-}$ denotes the set of non-negative reals (we use non-negative reals because we assume there exists an initial point in time). In this particular case there is only one solution

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

In our framework we want to make the interpretation in terms of the set of solutions more precise. More specifically, we are looking for a set of elements that can form the range of a semantic function that denotes the interpretation of the equation.

In the first place, the existence of variables in the above formulation suggests that an important element in the framework should be the ability to *name* elements that appear in the model. In general, these externally visible features of agents could be actions, signals, state variables, etc. In our formal framework we do not distinguish among the different types, and we refer to them collectively as a set of *signals* W. Each agent is then associated with an *alphabet* $A \subseteq W$ of the signal it uses. An agent is also characterized by a *signature*, which we denote with the symbol γ. The structure of the signature depends on the particular model of computation, and it uses the symbols in the alphabet to model the visible *interface* of the agent.

For the equation in the example above, the alphabet A consists of the names of the variables that appear in the equation:

$$A = \{x\}.$$

Note in particular that t is not included in the alphabet, because of its special role as an independent variable. Note also that the equation simply describes a condition for a function to be a solution. Therefore, when constructing a model for an agent represented by continuous time equations, we do not specify the direction of the signals (input or output), but simply associate a set of signals to each agent. Hence, the signature for agents in the continuous time model of computation simply consists of the set of symbols A:

$$\gamma = A.$$

Consider again the equation 1. As mentioned, we interpret the equation (agent) as the set of its possible solutions. In turn, we may interpret each solution as one possible *behavior* of the agent. We generalize this concept by always maintaining a clear distinction between models of agents (a.k.a. processes) and models of individual executions (a.k.a. behaviors). In different models of computation, individual executions can be modeled by very different kinds of mathematical objects. In our framework, we always call these objects *traces*. In the specific case of functions over time, an individual execution is a set of functions, one for each unknown in the equation (a singleton in our example, since there is only one dependent variable). An agent is a set of sets of functions.

We define traces for the continuous time model to be a close formalization of the natural interpretation of solution. However, we should make the relationship between the solution and the variable precise. Because the definition of a trace must be independent of the particular agent, it must take the alphabet as a parameter. In the case of the continuous time model of computation, we must assign a function over the reals to each of the symbols in the alphabet. For our example, we could use traces of the form

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

where the set V is the range of the functions. In our case we have $V = \mathbb{R}$. A trace thus contains both the solution, and the association of each of the functions in the solution to the variables that appear in the equation. For the example above, the (only) solution can be expressed as a trace $f : A \to (\mathbb{R}^{\not-} \to V)$ where

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

Note that there might be several possible valid definitions of a trace. For example, a trace might associate to each moment in time the values of the functions. The traces are thus functions of the form

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

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

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

$$x = 3t + x_0.$$

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

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

This example shows that, in general, an agent admits several different possible behaviors. Hence, a model of an agent, which we call a *trace structure* in our framework, consists of the signature γ and of a set P of traces. We usually denote the trace structure as the pair $T = (\gamma, P)$. This is analogous to verification methods based on language containment, where individual executions are modeled by strings and agents are modeled by sets of strings. However, our notion of trace is much more general and so is not limited to strings.

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

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

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

$$P = \{f : A \to (\mathbb{R}^{\neq} \to V) : \exists x_0, y_0[f(x) = \lambda t[3t + x_0] \\ \wedge\ [f(y) = \lambda t[4t + y_0]\}$$

Systems of equations can have two interpretations. On the one hand, they represent an agent whose constraints on the variables are expressed by different equations. On the other hand, they may represent the interaction of different agents, each represented by disjoint subsets of the equations in the system. The two views can be reconciled by interpreting the agent as a whole as the result of the interaction of the individual agents. We discuss the details of how this fits into our framework after introducing an example in continuous time that is more relevant to the PicoRadio system introduced in Section III,

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

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

This is a homogeneous second order differential equation in the variables f and s. The solutions of this equation describe an oscillatory behavior. In fact, this equation might be used to model an oscillator that generates a signal (for example a voltage) that we denote by the symbol s, whose frequency is controlled by another signal, denoted by the symbol f. Solutions to this equation are in the form of pairs of functions $s : \mathbb{R}^{\neq} \to V$ and $f : \mathbb{R}^{\neq} \to V$.

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

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

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

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

which represents a constant oscillation with frequency $10\, r/s$.

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

B. Operations on Trace and Trace Structures

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

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

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

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

The second operation is called *renaming*, and consists in changing the names of the visible elements of the agent. For functions over time, this corresponds to a substitution of variables. This, however, must be done carefully to avoid changing the underlying meaning of the equation. In our framework, the renaming operation *rename* takes as argument a renaming function r that maps the elements of the signature A into a new signature C, where A and C are both subsets of the common set of signals W. The function r is required to be a bijection in W to avoid conflicts of names and potentially a change in the behavior of the agents. If r is a renaming function, we define renaming on traces as the corresponding operation on the signals in the signature. Formally:

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

As for projection, renaming of trace structures can be seen as the natural extension to sets of the corresponding operation on individual traces. When applied to a trace structure, the effect is that of a renaming of the variables in the corresponding differential equation. This process corresponds to that of *instantiation* of a master agent into its instances.

Projection and renaming, seen as operators for scoping and instantiation, are common operations that are meaningful to all models of computation. For trace structures, they are always defined as the natural extension to sets of the corresponding operations on traces. The combination of the set of traces x for all alphabets A, and of the operations *proj* and *rename* has the structure of an algebra[1]. We call this algebra a *trace algebra*, and we usually denote it with the symbol $\mathcal{C}$.

For all trace algebras (and therefore, for all models of computation) we require that the operations on traces satisfy certain properties. There are two reasons for doing that. First, the properties ensure that the operations behave as expected given their intuitive meaning. For example, we expect that a projection followed by another projection be the same as the second projection alone, or that a projection that retains all symbols in the trace results in the very same trace. Similarly for renaming and for certain combinations of the operations.

Second, we can use these properties as assumptions to prove results that are independent of the particular model of computation in question. These results provide powerful tools that the designer of the model can use to prove general facts about the model and about its relationships with other models.

Similarly to trace algebra, the combination of all trace structures T and the operations of projection and renaming on trace structures form an algebra, that we call *trace structure algebra*. However, before we formally define a trace structure algebra, we introduce an additional operation on trace structures.

The third operation that we introduce is that of *parallel composition* of agents. A system of equations is an example of a parallel composition in our model of computation based on continuous time. Here, each equation is interpreted as a single agent. The system is also interpreted as an agent, the one that is obtained by composing the individual agents. An example of a system of equations is the following:

$$\begin{aligned} \frac{d^2 s}{dt^2} + f^2 s &= 0 && (2)\\ \frac{d^2 m}{dt^2} + I_1^2 m &= 0 && (3)\\ f &= m + I_2 && (4)\\ I_2 &= 3; && (5)\\ I_1 &= 2. && (6) \end{aligned}$$

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

[1]An algebra is simply a set together with some functions over that set.

We can easily formalize this notion in the framework of trace algebra. Let $T_1 = (\gamma_1, P_1)$ and $T_2 = (\gamma_2, P_2)$ be two trace structures, and denote with $T = T_1 \parallel T_2$ their parallel composition. Clearly, to model this composition, the signature of T must include the signals of both T_1 and T_2. Hence:

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

The set of traces P of T must be such that each trace belongs to both T_1 and T_2. However the traces must first be converted from one alphabet to another. This can be achieved by first extending the set of traces P_1 and P_2 to P_1^e and P_2^e, respectively, which are sets of traces over the alphabet A such that

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

The traces in P_1^e clearly satisfy the system of equations for T_1 (the additional functions are simply ignored), but do not necessarily satisfy that for T_2. Likewise, the traces in P_2^e satisfy the equation for T_2 but do not necessarily satisfy that for T_1. The parallel composition is the set of those traces that satisfy both,

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

Given this definition, it is straightforward to show that parallel composition corresponds to the usual operation of taking the intersection of the solutions of two different equations. Consider again the system of differential equations in Equation 2. This system can be represented as the parallel composition of 5 trace structures [2], as shown in Figure 2, where the rounds represent trace structures, and the connections represent shared signals (functions over time). The signature of the parallel composition is

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

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

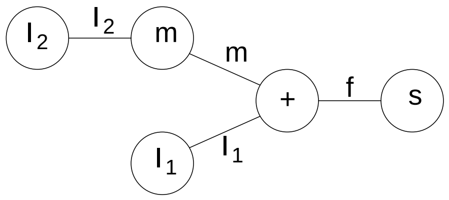

Fig. 2. Parallel composition of agents

The definition of parallel composition of agents shown above for the continuous time model of computation can be

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

generalized in a straightforward way in our framework. Parallel composition corresponds to the concurrent execution of two agents. As discussed above, the parallel composition $T = T_1 \parallel T_2$ is a set of traces in the union A of the alphabets of T_1 and T_2 that is "compatible" with the restrictions imposed by the agents being composed. We can formalize the notion of compatibility by requiring that if x is a trace of T with alphabet A, then its projection $\mathit{proj}(A_1)(x)$ on the alphabet of T_1 is in P_1, and the projection $\mathit{proj}(A_2)(x)$ on the alphabet of T_2 is in P_2. The set of traces in T must be maximal with respect to that property. It can be shown that the previous definition of parallel composition for the continuous time model is equivalent to this formulation.

The combination of the set of trace structures and the operations of projection, renaming and parallel composition of trace structures forms an algebra that we call *trace structure algebra*. Similarly to the algebra for traces, the operations in a trace structure algebra must satisfy certain properties. For example we require that parallel composition be commutative and associative, and that the extensions to sets of projection and renaming also behave consistently. Note that because these operations are predefined, given a trace algebra, and the set of trace structures to be used as the universe of agent models, a trace structure algebra is constructed in a fixed way. A fundamental result of this work is that the properties of the trace algebra are sufficient to ensure that the corresponding trace structure algebra satisfies its required properties.

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

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

The example of this section shows how to formalize within our framework the natural semantic domain of a model of computation based on continuous time and differential equations. It is worth noting how our representation of the agents is completely denotational, in that no rule is given to derive the output from the input. In addition, while our formalization is close to the natural semantic domain of traditional differential equations, the algebraic infrastructure introduces additional concepts such as hierarchy, instantiation and scoping in a natural way. For instance, the trace structures that correspond to the oscillators could be viewed as instantiations of a primitive component obtained by a renaming operation. Also, the frequency modulator that results from the parallel composition outlined above could be used as a primitive component: to that end, it is enough to hide the internal signals $\{m, I_2, f\}$ through a projection operation. These characteristics make our approach suitable as a foundation for the formal semantics of popular design tools, such as Simulink.

C. Summary and Notation

In the previous two sections we have introduced the concept of a trace algebra and a trace structure algebra. Formally a trace algebra $\mathcal{C}$ is a triple:

$$\mathcal{C} = (\mathcal{B}, \mathit{proj}, \mathit{rename}),$$

where we use the symbol $\mathcal{B}$ to denote the set of traces for all possible alphabets. We also use the symbol $\mathcal{B}$ as the function that to each alphabet A associates the set of possible traces with that alphabet. We say that $x \in \mathcal{B}(A)$ to denote that x is a trace in the alphabet A.

A trace structure is a pair

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

where γ is a signature and P a set of traces in the alphabet A of the trace structure ($P \subseteq \mathcal{B}(A)$). A trace structure algebra is a 4-tuple

$$\mathcal{A} = (\mathcal{T}, \mathit{proj}, \mathit{rename}, \parallel),$$

where $\mathcal{T}$ is a set of trace structures. The operations of projection and renaming are obtained from the corresponding operation on traces as extensions to sets. The symbol $\parallel$ denotes parallel composition. If $T = T_1 \parallel T_2$, then the set of traces P of T is obtained from the set of traces P_1 and P_2 of T_1 and T_2 as

$$\begin{aligned} P \quad = \quad & \{x \in \mathcal{B}(A) : \mathit{proj}(A_1)(x) \in P_1 \wedge \\ & \mathit{proj}(A_2)(x) \in P_2\}. \end{aligned}$$

The signature of the parallel composition must also be obtained from that of the agents being composed. Each model of computation must define this operation, as well.

In the sections that follows we will present four more models of computation at different levels of abstraction and provide their formalization in the framework of trace algebra.

D. Concatenation and Sequential Composition

In the case of differential equations, a system corresponds to the parallel composition of agents. In other models of computation, however, the emphasis may be on a sequential "execution" of the agents. This could be seen as a parallel composition where control flows from one agent to another, thus making only one agent active at a time. Nevertheless, this situation is so common that it warrants the introduction of some special operations and notation.

For these models we introduce a third operation on traces called *concatenation*, which corresponds to the sequential composition of behaviors. Similarly to the other operations, concatenation must also satisfy certain properties that ensure that its behavior is consistent with its intuitive interpretation. Other than that, the definition of concatenation depends upon the particular model of computation. Concatenation is also used to define the notion of a prefix of a trace. We say that a trace x is a prefix of a trace z if there exists a trace y such that z is equal to x concatenated with y.

With concatenation, we distinguish between a complete behavior and a partial behavior. A complete behavior has no endpoint. Since a complete behavior goes on forever, it does not make sense to talk about something happening "after" a complete behavior. A partial behavior has an endpoint; it can be a prefix of a complete behavior or of another partial behavior. Every complete behavior has partial behaviors that are prefixes of it; every partial behavior is a prefix of some complete behavior. The distinction between a complete behavior and a partial behavior has only to do with the length of the behavior (that is, whether or not it has an endpoint), not with what is happening during the behavior; whether an agent does anything, or what it does, is irrelevant.

Complete traces and *partial traces* are used to model complete and partial behaviors, respectively. A given object can be both a complete trace and a partial trace; what is being represented in a given case is determined from context. For example, a finite string can represent a complete behavior with a finite number of actions, or it can represent a partial behavior.

Like in the other cases, concatenation induces a corresponding operation on trace structures that we call *sequential composition*. We can illustrate these concepts by extending the example of the previous section to a continuous time model of computation that is suitable to studying hybrid systems, and that includes the operation of concatenation.

A typical semantics for hybrid systems includes continuous *flows* that represent the continuos dynamics of the system, and discrete *jumps* that represent instantaneous changes of the operating conditions. In our model we represent both flows and jumps with single piece-wise continuous functions over real-valued time. The flows are continuous segments, while the jumps are discontinuities between continuous segments. In this paper we assume that the variables of the system take only real or integer values and we defer the treatment of a complete type system for future work. The sets of real-valued and integer valued variables for a given trace are called $V_{\mathcal{R}}$ and $V_{\mathbb{N}}$, respectively.

Traces may also contain actions, which are discrete events that can occur at any time. Actions do not carry data values. For a given trace, the set of input actions is M_I and the set of output actions is M_O.

The signature γ of each agent is a 4-tuple of the above sets of signals:

$$\gamma = (V_{\mathcal{R}}, V_{\mathbb{N}}, M_I, M_O).$$

The sets of signals may be empty, but we assume they are disjoint. The alphabet of γ is

$$A = V_{\mathcal{R}} \cup V_{\mathbb{N}} \cup M_I \cup M_O.$$

The set of partial traces for a signature γ is $\mathcal{B}_P(\gamma)$. Each element of $\mathcal{B}_P(\gamma)$ is as a triple $x = (\gamma, \delta, f)$. The non-negative real number δ is the *duration* (in time) of the partial trace. The function f has domain A. For $v \in V_{\mathcal{R}}$, $f(v)$ is a function in $[0,\delta] \to \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers and the closed interval $[0,\delta]$ is the set of real numbers between 0 and δ, inclusive. This function must be piece-wise continuous and right-hand limits must exist at all points. Analogously, for $v \in V_{\mathbb{N}}$, $f(v)$ is a piece-wise constant function in $[0,\delta] \to \mathbb{N}$, where $\mathbb{N}$ is the set of integers. For $a \in M_I \cup M_O$, $f(a)$ is a function in $[0,\delta] \to \{0,1\}$, where $f(a)(t) = 1$ iff action a occurs at time t in the trace.

The set of complete traces for a signature γ is $\mathcal{B}_C(\gamma)$. Each element of $\mathcal{B}_C(\gamma)$ is as a pair $x = (\gamma, f)$. The function f is defined as for partial traces, except that each occurrence of $[0,\delta]$ in the definition is replaced by $\mathbb{R}^{\not-}$, the set of non-negative real numbers.

To complete the definition of this trace algebra, we must define the operations of projection, renaming and concatenation on traces. The projection operation $\textit{proj}(B)(x)$ is defined iff $M_I \subseteq B \subseteq A$. The trace that results is the same as x except that the domain of f is restricted to B. The renaming operation $x' = \textit{rename}(r)(x)$ is defined iff r is a one-to-one function from A to some $A' \subseteq W$. If x is a partial trace, then $x' = (\gamma', \delta, f')$ where γ' results from using r to rename the elements of γ and $f' = r \circ f$.

The definition of the concatenation operator $x_3 = x_1 \cdot x_2$, where x_1 is a partial trace and x_2 is either a partial or a complete trace, is more complicated. If x_2 is a partial trace, then x_3 is defined iff $\gamma_1 = \gamma_2$ and for all $a \in A$,

$$f_1(a)(\delta_1) = f_2(a)(0)$$

(note that δ_1, δ_2, etc., are components of x_1 and x_2 in the obvious way). When defined, $x_3 = (\gamma_1, \delta_3, f_3)$ is such that $\delta_3 = \delta_1 + \delta_2$ and for all $a \in A$

$$\begin{aligned} f_3(a)(\delta) &= f_1(a)(\delta) \text{ for } 0 \le \delta \le \delta_1 \\ f_3(a)(\delta) &= f_2(a)(\delta - \delta_1) \text{ for } \delta_1 \le \delta \le \delta_3. \end{aligned}$$

Note that concatenation is defined only when the end points of the two traces match. The concatenation of a partial trace with a complete trace yields a complete trace with a similar definition. If $x_3 = x_1 \cdot x_2$, then x_1 is a *prefix* of x_3.

Trace structures in this model have again signature γ. Their definition must be extended to contain a set of complete traces P_C and a set of partial traces P_P. We also denote with $P = P_C \cup P_P$ the set of all traces (consistently with the previous formulation). The sequential composition $T'' = T \cdot T'$ is then defined when $A = A'$, and in that case:

$$\begin{aligned} A'' &= A = A', \\ P''_C &= P_C \cup (P_P \cdot P'_C), \\ P''_P &= P_P \cdot P'_P. \end{aligned}$$

where concatenation is naturally extended to sets of traces.

As for parallel composition, the definition of sequential composition is constructed from equivalent concepts in the trace algebra. Therefore, the trace structure algebra can still be constructed automatically.

V. Models of Computation and Trace Algebras

The sections that follow introduce the formalization of the semantic domain for more models of computation. In all cases we follow the same pattern by first presenting the natural formalization, and then the formalization in terms of trace algebras.

For each model of computation we also sketch an example of its typical applications. Each example is one of the subsystems of the PicoRadio architecture shown before. Later sections of this paper show how we can derive relationships between these models within the framework.

A. CSP

Communicating Sequential Processes were introduced by Hoare [15]. It consists of a collection of agents that interact through the exchange of actions. Actions are shared and must be synchronized: when an agent wishes to perform an action with another agent, it must wait until the other agent is ready to perform the same action.

CSP is particularly well suited to handle cases where a tight synchronization is required or to schedule access to a shared resource. In our example we can use CSP to model a manager subsystem that regulates access to a set of parameters and tables that can be set and read by the user and by the protocol stack. To do this, the manager initially waits to synchronize with either the protocol stack or the user input; once synchronized with one of the two parties, it reserves the shared resource and handles the communication by performing a set of actions (e.g. read, write, update). At the end of the transaction, the manager goes back to its initial state and waits to synchronize again. Figure 3 shows a diagram of this subsystem.

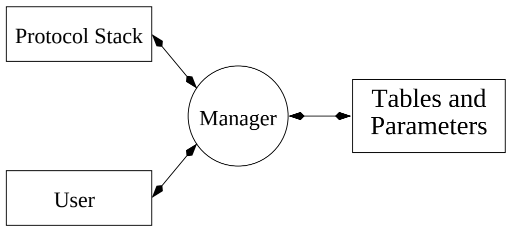

Fig. 3. Table manager and UI interface

Constructing a trace algebra and a trace structure algebra for this model is particularly simple because the communication model fits very easily in our framework. A single execution of an agent (a trace) is simply a sequence of actions from the alphabet A of possible actions. Formally, we define

$$\mathcal{B}(A) = A^{\infty},$$

where the notation A^{∞} includes both the finite and the infinite sequences over A. Projection and renaming are defined as expected: if $x \in \mathcal{B}(A)$ and $B \subseteq A$, then $\mathit{proj}(B)(x)$ is the sequence formed from x by removing every symbol a not in B. More formally, if $x' = \mathit{proj}(B)(x)$, then the length of x' (written $\mathit{len}(x')$) is

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

where $\mathit{len}(x') = \omega$ when the set is infinite. The k-th element of x' corresponds to the k-th element of x that belongs to B. Hence, if $x(n) \in B$, then $x'(k) = x(n)$ where

$$k = |\{j \in \mathbb{N} : 0 \leq j < n \wedge x(j) \in B\}|.$$

Note that any n and k combination is unique.

For renaming, assume without loss of generality that $x \in \mathcal{B}(A)$ is of the form

$$x = \langle a_0, a_1, a_2, \ldots \rangle;$$

then

$$\mathit{rename}(r)(x) = \langle r(a_0), r(a_1), r(a_2), \ldots \rangle.$$

Models of agents are obtained in the standard way, as a collection of sequences. The signature γ of an agent includes a set of input actions I and a set of output actions O. The parallel composition $T = T_1 \parallel T_2$ is defined when O_1 and O_2 are disjoint, and in that case

$$\begin{aligned} I &= (I_1 \cup I_2) - (O_1 \cup O_2), \\ O &= O_1 \cup O_2. \end{aligned}$$

Given the definition of projection, parallel composition clearly requires that trace structures (agents) synchronize on the shared actions.

This model is based solely on actions that bear no value. It is straightforward to extend the model to include a value for each action. We define:

$$\mathcal{B}(A) = (A \times V)^{\infty},$$

where V is the set of possible values. Projection and renaming are extended by having them act only on the first component of the pair. Formally, if $x \in \mathcal{B}(A)$ and $x' = \mathit{proj}(B)(x)$ then the length of x' (written $\mathit{len}(x')$) is

$$\mathit{len}(x') = |\{j \in \mathbb{N} : 0 \leq j < \mathit{len}(x) \wedge x(j) \in (B, V)\}|$$

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

$$k = |\{j \in \mathbb{N} : 0 \leq j < n \wedge x(j) \in (B, V)\}|.$$

Likewise for renaming. Without loss of generality, assume

$$x = \langle (a_0, v_0), (a_1, v_1), \ldots \rangle;$$

then

$$\mathit{rename}(r)(x) = \langle (r(a_0), v_0), (r(a_1), v_1), \ldots \rangle.$$

With this definition we can construct a trace structure that represents the table manager depicted in Figure 3. The signature γ includes inputs and outputs to and from both the protocol stack and the user interface, with actions that set and read the appropriate parameters. For example, the parameters could be a set of virtual connections, specified as pair of addresses (*vci* and *vpi*) and the packet length. Two typical traces for the manager deal with handling requests from the protocol and from the user, as in

$$\begin{aligned} P &= \{< \text{ps\_req}, \text{vci}(10), \text{vpi}(13), \text{ps\_release}, \ldots >, \\ &\quad < \text{user\_req}, \text{length}(1500), \text{vpi}(0), \ldots >, \ldots\}, \end{aligned}$$

where *ps* refers to the protocol stack, and *user* to the user interface. Note that while the manager can non-deterministically choose to serve the protocol stack or the user, it must continue to serve the party that was chosen until the shared resource is released.

Compared to the traditional CSP model, ours differ in some respects. For example, in our model it is possible for several agents to synchronize on the same action, thus making it possible for one agent to *broadcast* an event. In a more traditional model, only one of the listeners is able to react to the event. This is a consequence of our definition of parallel composition.

Another difference is that in our model (and in all other models constructed using trace algebras), the operation of parallel composition and renaming are clearly differentiated. In other words, parallel composition in our model does not *create* the connections, but is limited to constructing an agent whose projections are compatible with the ones being composed. Renaming must be invoked separately (and before the composition) to create the appropriate instances of the agents to be composed.

B. Process Networks

Process networks are collections of agents that operate on infinite streams of data. Streams are traditionally implemented as FIFO queues that connect processes that can produce (write) and consume (read) tokens. Process networks are particularly well suited to modeling digital signal processing applications, given the good match between the typical data model of signal processing and the communication model of process networks.

As an example we might consider a demodulator that uses a local reference to convert an incoming signal from high to base band. The decoder receives a stream of tokens that corresponds to, for instance, the output of the local oscillator described above in Section IV-A. At the same time it receives a stream of data tokens to be demodulated. The demodulator combines the two streams and then applies a filter to retain only the component of interest. A diagram of this subsystem is shown in Figure 4.

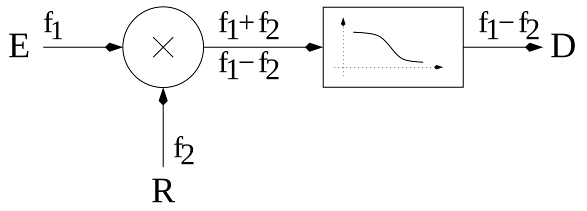

Fig. 4. A signal demodulator

The important property of this model is that the exact time at which tokens arrive at the input is irrelevant, and that only their order within the same stream determines the output stream (together, of course, with their value). The natural domain for this kind of model is then clearly that of a function on streams, which can in turn be formalized as sequences. In the case of our demodulator, if we denote with R and E the reference and the modulated streams, and with D the demodulated stream, we can represent the decoder in the natural domain as a function f from the inputs to the output:

$$D = f(R, E).$$

Parallel composition of agents is defined by composing for each stream s the function whose range is s with the function whose domain is s. This definition becomes circular in the presence of loops in the structure of the parallel composition. In this case, the composition is defined by breaking the loop at some point, and then looking for the fixed points of the function that results. If we do not restrict the range of the possible functions f, the parallel composition may have several fixed points (or even no fixed points at all), and hence exhibit non-deterministic behavior. Because we ultimately want to model physical processes that *are* deterministic, we must impose some constraints on f.

We say that a stream v is a *prefix* of a stream u if v is equal to some initial segment of u. This relation can be extended to sets of streams by requiring that all streams in the first set are a prefix of the corresponding stream in the second set. This relation is easily proved to be a partial order on the streams.

To ensure that a composition of stream functions is determinate, the function f of each of the components must be *continuous* with respect to the prefix ordering on the streams. If that is case, then we are assured that there exists a unique least fixed point, and the parallel composition is defined in terms of that. In addition, continuity implies monotonicity, which in turn ensures that the response of the system to a specific input can be computed incrementally from progressively longer prefixes.

In the following we will show two ways of describing the process networks model in our framework. The first method is closer to the semantic domain based on functions on streams, but falls short in the definition of parallel composition. The second method fixes this problem, at the expense of modeling the traces at a more detailed level of abstraction.

In our initial attempt we follow the natural semantic rather closely. First we define the signature γ of a trace structure. Process networks clearly distinguish between inputs and outputs, hence we define

$$\gamma = (I, O),$$

where I and O are disjoint sets of input and output signals, respectively. In the example above, we have

$$\begin{aligned} I &= \{R, E\}, \\ O &= \{D\}. \end{aligned}$$

Given a stream function, a trace is a single application from a set of input streams to a set of output streams. If we define the alphabet of a trace to be the set $A = I \cup O$, and formalize streams as the finite and infinite sequences over a value domain V, denoted by V^{∞}, then the set of all possible traces is

$$\mathcal{B}(A) = A \rightarrow V^{\infty}.$$

A trace structure is simply the signature together with a set of traces, i.e. $T = (\gamma, P)$ where $P \subseteq A \rightarrow V^{\infty}$. If we separate the contributions of the inputs and the outputs, the set P of traces can be seen as (is isomorphic to) a subset of

$$(I \rightarrow V^{\infty}) \times (O \rightarrow V^{\infty}),$$

that is, as a function on streams.

In order to comply with the process network model, we also insist that the functions so identified have the necessary continuity and monotonicity properties with respect to the prefix ordering defined on the sequences. In other words, not all set of traces may form a trace structure.

We define a *functional* trace structure as one that associates at most one output stream to each input stream. More formally, the condition is simply

$$\mathit{proj}(I)(x) = \mathit{proj}(I)(y) \Rightarrow x = y.$$

To define monotonicity we first need a partial order on traces. We say that a trace $x \in \mathcal{B}(A)$ is a *prefix* of a trace $y \in \mathcal{B}(A)$, written $x \sqsubseteq y$, if $x(a)$ is a prefix of $y(a)$ for all $a \in A$. Let $T = (\gamma, P)$ be a trace structure. Then T is monotonic if for all $x, y \in P$,

$$\mathit{proj}(I)(x) \sqsubseteq \mathit{proj}(I)(y) \Rightarrow \mathit{proj}(O)(x) \sqsubseteq \mathit{proj}(O)(y).$$

Note that, in particular, this also implies

$$\mathit{proj}(I)(x) \sqsubseteq \mathit{proj}(I)(y) \Rightarrow x \sqsubseteq y.$$

Finally we define the process network trace structure algebra as the algebra that contains all and only the functional and monotonic trace structures.

The operations of projection and renaming on traces are easily defined. If $x \in \mathcal{B}(A)$, $B \subseteq A$ and r is a renaming function, then

$$\begin{aligned} \mathit{proj}(B)(x) &= \lambda a \in B[x(a)], \\ \mathit{rename}(r)(x) &= \lambda a \in A[x(r(a))]. \end{aligned}$$

Parallel composition on trace structures is defined as usual in terms of the projection operation. Note that the trace structure obtained from a composition contains all the traces that are compatible with the agents being composed; in particular, it will contain all the fixed-points in a composition that involves a feedback loop. Figure 5 illustrates the point. Here two instances of the trace structure I are composed so that the input of one corresponds to the output of the other. The trace structures are also defined to be the identity function on streams, i.e. they contain all pairs of identical input and output streams. It is easy to show that also the composition contains all pairs of identical streams. This is a problem, as it doesn't faithfully represent the semantics of the original formulation of process networks, that in this case includes only empty streams, the least of the fixed-points in the composition. The problem with our model is that whether a trace is included in the composition or not depends exclusively on whether its projections are part of the individual components. In order to include only the least fixed-point, we would also need to check whether other traces (more specifically, prefixes) are also included in the composition.

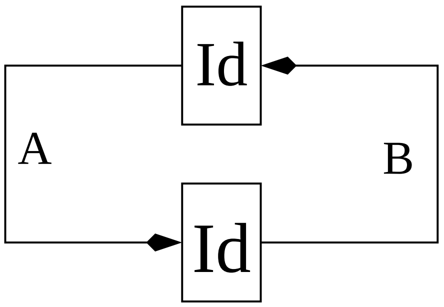

Fig. 5. Parallel composition with feedback

Our solution to this problem avoids changing the definition of parallel composition (which is common to all trace structure algebras), but requires us to develop a new semantic domain at a more detailed level of abstraction. The additional information is enough the determine the result of the parallel composition exactly.

In the new formalization, each trace is a totally ordered sequence of events. Formally we have:

$$\mathcal{B}(A) = (A \times V)^{\infty}.$$

Note that this is exactly the definition that we have for the semantic domain for communicating sequential processes. The definition of projection and renaming also parallels the definitions given in Section V-A, and will not be repeated here. The signature of the trace structures is again a pair of disjoint sets $\gamma = (I, O)$ as before. Despite the similarities with CSP, this formulation results in a different model of computation because the class of trace structures that we construct must satisfy some additional conditions, as was also the case in our initial formalization of process networks.

In the new formulation, the traces in the trace algebra carry order information for all events. This means that we can tell whether an input (or an output) event occurred before or after another input or output event. Because the semantics of process networks is independent of this ordering, a trace structure must contain traces that represent all orderings of inputs and outputs that are compatible with a particular stream function. The word "compatible" here has two meanings. First we must only include those orderings that result in monotonic functions. Second, inputs and outputs can not occur arbitrarily ordered in a trace: output tokens should never precede the input tokens that caused them. The rest of this section makes these two requirements more precise.

It is easy to construct a homomorphism h to the previous trace algebra that loses the ordering information. Given a trace x in the alphabet A, we isolate the sequence relative to a signal a using a projection operation, and then construct the appropriate function. More formally:

$$h(x) = \lambda a \in A \lambda n \in \mathbb{N}[(\mathit{proj}(\{a\})(x))(n))_v],$$

where the subscript v denotes the second component of a pair in $A \times V$. This function is a homomorphism in that it commutes with the application of the other operations on traces, projection and renaming.

The functionality and monotonicity conditions are best expressed in the domain of stream functions, as we don't want the particular order of a trace to affect the prefix relation. A functional trace structure can be defined as follows. For all $x, y \in P$, the following condition must be satisfied:

$$h(\mathit{proj}(I)(x)) = h(\mathit{proj}(I)(y)) \Rightarrow h(x) = h(y).$$

Similarly for monotonicity. If $T = (\gamma, P)$, then T is monotonic if for all $x, y \in P$,

$$h(\mathit{proj}(I)(x)) \sqsubseteq h(\mathit{proj}(I)(y)) \Rightarrow \\ h(\mathit{proj}(O)(x)) \sqsubseteq h(\mathit{proj}(O)(y)).$$

In order to include all orderings in the trace structures, we might be tempted to state that if $x \in P$, then any other trace y such that $h(y) = h(x)$ should be in P. Doing this would remove all information regarding the ordering of inputs and outputs. As a result, the composition would again suffer from the same problem (inclusion of all of the fixed-points) that we had with the previous model. Instead, we must strengthen this condition.

We do this in two steps. Given a trace structure $T = (\gamma, P)$, we first look for a subset $P_0 \subseteq P$ of only those traces that can be characterized as *quiescent*, in the sense that all the outputs relative to the inputs have been produced. In fact, we are looking for the set P_0 with the added property that the outputs occur in the sequence as soon as possible. This is similar to the *fundamental model* assumption in asynchronous design. In the formalization that follows, we will assume that tokens have no value to simplify the notation. Under this assumption, P_0 can be formalized as follows:

$$\mathcal{P}_0 = \{z \in P : \forall x, y \in \mathcal{B}(A)\ \forall b \in I \\ z = x\langle b\rangle y \Rightarrow x \in P\},$$

where the notation $\langle b\rangle$ denotes the sequence made of only the symbol b. The intuition behind this definition is as follows. Assume that a trace $z \in P$ can be written as the concatenation $x\langle b\rangle y$ with $x \in P$. Then, since T is functional, for any trace x' such that $h(\mathit{proj}(I)(x')) = h(\mathit{proj}(I)(x))$, we have $h(\mathit{proj}(O)(x')) = h(\mathit{proj}(O)(x))$. So, in particular, none of the output tokens that are contained in the suffix y ever occur before the input token b in any other trace in P with the same inputs as x. If y starts with an output token c, this condition tells us that c does not appear any sooner in any other trace, and therefore that z outputs c as soon as possible. The universal quantification on x, y and b extends the property to the entire trace z.

Since P_0 is the "fastest" subset of P, we can now construct a new set that includes all possible delays of the output. We construct this set by induction. Given a set X of traces, we define a function F that adds all traces where each output that precedes an input is delayed by one position. Formally:

$$F(X) = X \cup \{x\langle b, c\rangle y \in \mathcal{B}(A) : \\ x\langle c, b\rangle y \in X \wedge b \in I \wedge c \in O\}. \qquad (7)$$

Intuitively we would like to repeatedly apply this function starting from P_0 until we reach a fixed-point. This function is monotonic relative to set containment (given $X_1 \subseteq X_2$, $F(X_2)$ will add at least the traces that $F(X_1)$ adds, plus possibly some more). In addition, F creates progressively larger sets, i.e.

$$\forall X[X \subseteq F(X)].$$

When this is the case, we say that F is *inflationary* at X. These two properties are enough to guarantee the existence of a fixed-point. In fact, they guarantee the existence of a *least* fixed-point greater than or equal to P_0, the minimal set that contains P_0 and all the traces with delayed outputs[3]. Let's denote with $P_{\mathrm{fp}}(P)$ the fixed-point obtained by starting the recursion with the P_0 associated to P. Then we define the trace structure algebra for process networks as the one that contains only those trace structures such that

$$P = P_{\mathrm{fp}}(P).$$

The system shown in Figure 5 now results in a correct composition. In fact, the bottom trace structure I will require that the input at A appear before any output on B in all its traces. Likewise, the top trace structure will require its input, which corresponds to B, to occur before the output A. This contradiction will rule out all traces except the empty one, as dictated by the least fixed-point semantics.

C. Discrete Event

A discrete event system consists of agents that interact by exchanging *events* on a set of signals. Each event is labeled with a *time stamp* that denotes the time at which the event occurred. The notion of time is global to the entire system, so that if any two events have the same time stamp then they are considered to occur at the same time. The set of time stamps is often taken to be the set of positive integers or real numbers, ordered by the usual order. The order is then extended to the events so that events with smaller time stamps precede events with higher time stamps. The model is called *discrete* because it is required that for each signal the set of time stamps is not dense in the reals.

Examples of discrete event systems abound, as both Verilog and VHDL use this model as their underlying simulation semantics. For our example, we might consider the subsystem that implements the protocol stack that handles the data stream after it has been demodulated. The stack includes functions that modify and depend on the tables

[3]Technically it is the greatest lower bound of the set of fixed-points of F that are greater than or equal to P_0

and parameters managed by the subsystem described in the section on CSP (V-A). In addition, the protocol stack interacts with the physical layer at the lower levels, and then unpacks and delivers the raw data to the application. The non-recurring nature of these operations, their unpredictable timing and the dependency of the protocol behavior on their timing make a discrete event model more suitable than, say, a data-flow model. A typical protocol stack of four layers is shown in Figure 6.

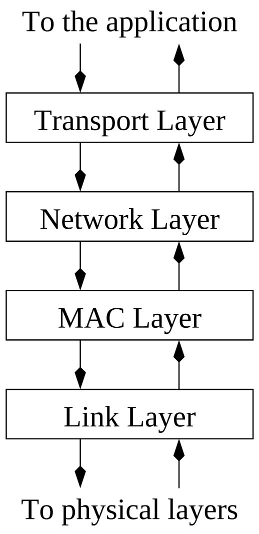

Fig. 6. Protocol Stack

In the natural semantic domain, each behavior of an agent can be characterized as a sequence of events, each associated to an increasing time stamp. Note that events that occur in unrelated parts of the system are still ordered by their time stamps. Different events may occur with the same time stamp. In most cases, if two discrete event models differ at all, they differ in the way events with the same time stamp are handled. For the purpose of simulating such systems, some models define a notion of a *delta cycle* that orders the events with identical time stamps. Others don't define any specific way to handle this occurrence, leading to non-determinism.

It is natural to construct a semantic domain in our framework based on the interpretation of a behavior as a sequence of events with time stamps. If A is the set of signals, V the set of values and $\mathbb{R}^{\neq}$ the set of non-negative reals, we define the traces as follows:

$$\mathcal{B}(A) = (A \times V \times \mathbb{R}^{\neq})^{\infty}.$$

Two conditions must be imposed on the time stamps of a trace. First, the time stamps in the sequence must be non-decreasing, i.e. if x is a trace and n and m are two natural numbers such that $n, m < \mathit{len}(x)$, then

$$n < m \Rightarrow x(n)_t \leq x(m)_t,$$

where the subscript t denotes the time stamp of the event. Second, the time stamps of an infinite sequence x must be divergent, i.e. for all $t \in \mathbb{R}^{\neq}$, there is an event in x with time stamp greater than t. Discreteness can be enforced by requiring that for all non-negative reals $t \in \mathbb{R}^{\neq}$, there is only a finite number of events in x such that $x(n)_t < t$. Projection and renaming are defined similarly to the functions defined for CSP in Section V-A.

The signature γ of a trace structure distinguishes between the set of inputs I and the set of output O, that together form the alphabet A. Trace structures are then built as a signature with a set of traces in a way similar to the models that we have already presented. Constraints can be imposed on the set P of traces of a trace structure, analogous to the monotonicity and continuity requirements for process networks.

As an example from our protocol stack, one of the layers may include, among others, two traces, one for a successful operation, and the other for the occurrence of a timeout. The discrete event model is required in this case, as the process network model is unable to handle timeouts.

In some discrete event models, a new event occurs on a signal if and only if the corresponding value for that signal has changed since the previous occurrence. Traces that have this property are called *stutter free*. If this is the case, it is convenient in our framework to define the set of traces as the subset of stutter free traces. We can do this by defining a function that, given a trace, produces its unique stutter free equivalent by removing the unnecessary events. We call this process *stutter removal*. Note that discrete traces result in discrete traces after stutter removal.

D. Pre-Post

In this last model of computation that we present we are concerned with modeling non-interactive constructs, such as the ones that occur in a programming language. In this case we are interested only in an agents possible final states given an initial state. This semantic domain could therefore be considered as a denotational representation of an axiomatic semantics.

In our example, this model may be appropriate for the higher levels of the protocol stack, and in particular for the application layer where most of the functionality can be described as non interactive procedure calls. Note how this model of computation differs from those that were introduced in the previous sections, all of which included some notion of "evolution" of the system. Nontheless, traces don't necessarily require that notion, and we can easily fit this model in our framework.

Traditionally, the semantics for this kind of models is constructed by first defining a *state* as a set of variables $S = \{s_i\}$, and then indicating the rules according to which each construct in the programming language modifies this state. A natural semantic domain for describing the constructs is therefore a set of pairs of initial and final state, one for each possible initial state.

The formulation in the framework of trace algebra is almost identical to the natural domain. The signature γ of the agents is simply the set of variables A the agent depends on and writes to. Each trace is made of pairs of states. A state s is a function with domain A that to each variable $a \in A$ associates a value $s(a)$ from a set of values V. We also define a degenerate, undefined state $\perp$. Given an alphabet A a trace is simply a pair of states

$$\mathcal{B}(A) = (s_i, s_f),$$

where $s_i, s_f : A \to V$ denote the initial and the final state, respectively. Here, the initial state must be non-degenerate. A degenerate final state denotes constructs whose final state is either undefined, or that fail to terminate.

If $s : A \to V$ is a state, we can define projection and renaming on states as follows:

$$\begin{aligned} \textit{proj}(B)(s) &= \lambda a \in B[s(a)], \\ \textit{rename}(r)(s) &= \lambda a \in A[s(r(a))]. \end{aligned}$$

Then, if $x = (s_i, s_f)$ is a trace, we define projection and renaming by the obvious extension:

$$\begin{aligned} \textit{proj}(B)(x) &= (\textit{proj}(B)(s_i), \textit{proj}(B)(s_f)), \\ \textit{rename}(r)(x) &= (\textit{rename}(r)(s_i), \textit{rename}(r)(s_f)). \end{aligned}$$

A trace structure is easily constructed as a set of traces. As usual, the notion of parallel composition arises automatically given the definition of projection. However, in this particular model, parallel composition is not the main operation of interest, since we are modeling the behavior on non-interacting constructs. In fact, handling shared variables of concurrent programs is problematic with these definitions, and we define parallel composition to be undefined when the signatures of two agents overlap. Instead, we concentrate on the concatenation operation which is relevant to define the concept of sequential composition.

As mentioned in Section IV-D, we must distinguish between complete and partial traces. The above definition of a trace can be interpreted either way, depending on whether we consider the behavior to be completed or not. A non-terminating trace could be considered as a partial trace, assuming that non-termination occurs within a bounded amount of time. This is quite unusual: it may occur, for example, if the duration of an infinite loop decreases exponentially from one iteration to the other.

If $x = (s_i, s_f)$ and $x' = (s'_i, s'_f)$ are traces, the concatenation operation $x'' = x \cdot x'$ is defined if and only if x is a partial trace, the signature A and A' are the same, and the final state of x is identical to the initial state of x'. As expected, when defined, x'' has alphabet $A'' = A = A'$ and contains the initial state of x and the final state of x':

$$x'' = (s_i, s'_f).$$

Trace structures in this model have signature A and they provide the semantics of statements in a programming language. The signature A indicates the variables accessible in the scope where the statement appears. For example, the traces in the trace structure for an assignment to variable v are of the form (s_i, s_f), where s_i is an arbitrary initial state, and s_f is identical to s_i except that the value of v is equal to the value of the right-hand side of the assignment statement evaluated in state s_i (we assume the evaluation is side-effect free).

The semantics of a procedure definition is given by a trace structure with an alphabet $\{v_1, \ldots, v_r\}$ where v_k is the k-th argument of the procedure. The semantics of a procedure call `proc(a, b)` is the result of renaming $v_1 \to a$ and $v_2 \to b$ on the trace structure for the definition of `proc`. The parameter passing semantics that results is *value-result* (i.e. no aliasing or references) with the restriction that no parameter can be used for both a value and result. More realistic (and more complicated) parameter passing semantics can also be modeled.

To define the semantics of `if-then-else` we introduce a function $\textit{init}(x, c)$ that is true if and only if the predicate c is true in the initial state of trace x. The formal definition depends on the particular trace algebra being used. For pre-post traces, $\textit{init}(x, c)$ is false for all c if x has $\perp_*$ as its initial state.

For the semantics of `if-then-else`, let c be the conditional expression and let P_T and P_E be the sets of possible traces of the `then` and `else` clauses, respectively. The set of possible traces of the `if-then-else` is

$$P = \{x \in P_T : \textit{init}(x, c)\} \cup \{x \in P_E : \neg\textit{init}(x, c)\}$$

that is, we choose the traces from one or the other clause according to the truth value of the condition. Notice that this definition can be used for any trace algebra where $\textit{init}(x, c)$ has been defined, and that it ignores any effects of the evaluation of c not being atomic. The semantics of other, more complicated, constructs like loops could also be defined using similar techniques. We refer the interested reader to [5].

VI. Conservative Approximations and Relationships Among Models of Computation

In the previous section we have presented the formalization of several models of computation at different levels of abstraction, and how they can all be described in the framework of trace algebra. For each model we have suggested a particular application in the context of a system similar to the PicoRadio project. The whole system is depicted in Figure 1. In order to understand the behavior and the properties of the whole system, we need to understand the interplay between the different subsystems. We can accomplish this by relating the semantic domains that we have developed in the previous section and study how the different notions of computation fit together.

In what follows we develop a concept that we call *conservative approximation*. Intuitively a conservative approximation is a mapping of agents from one semantic domain to another, more abstract, domain. We actually employ two such mappings. The first mapping represents an upper bound of the agent: the abstract agent represents all of the possible behaviors of the agent in the more detailed domain, plus possibly some more. This mapping is usually denoted by Ψ_u. The second is a lower bound: the abstract agent represents only possible behaviors of the more detailed one, but possibly not all. We denote it by Ψ_l.

When used in combination, these two mappings allow us to relate results in the abstract domain to results in the more detailed domain. In particular, a conservative approximation is defined to preserve results related to trace containment, such that if T_1 and T_2 are trace structures, then:

$$\Psi_u(T_1) \subseteq \Psi_l(T_2) \Rightarrow T_1 \subseteq T_2.$$

We refer to our previous work for more details ([4]).

Where the lower and the upper bound coincide we can talk about an inverse of the approximation, by assigning to the agent in the abstract domain the unique inverse image of the two mappings. Where this function is defined, it can be interpreted as a refinement or concretization of the abstract semantic domain into the detailed one. One direction in our research currently focuses on characterizing the properties of this inverse function.

A. DE to PN

In this section we will explore the relationships between the discrete event model presented in Section V-C and the process network model presented in Section V-B. We will use the simple version of DE that consists of a sequence of events.

During the presentation we will refer to Figure 7 and Figure 8. Figure 7 depicts the mappings that relate traces in the different domains. Figure 8 shows the corresponding mappings when applied to the domains of trace structures (sets of traces).

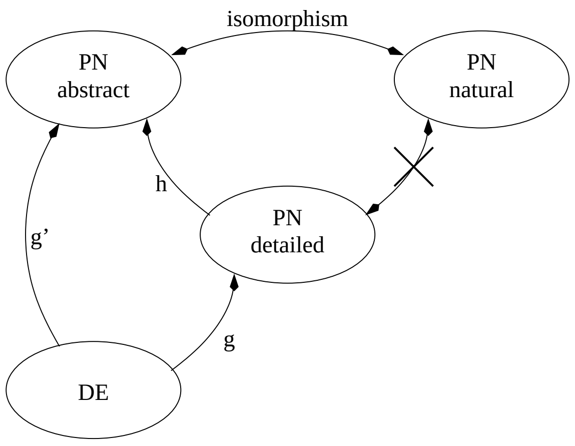

Fig. 7. Relations between trace algebras

We have already pointed out that the natural domain is that of functions on streams. Our initial abstract formalization is a model of traces that is isomorphic to the set of streams. However, the corresponding formalization in terms of trace structures led to a problem with the composition operator: in the original model, composition is defined so that it includes only the least fixed-point of the functions that satisfy a certain equation; in our model, instead, composition includes all the fixed-points. Thus we are unable to find an isomorphism between the trace structures of our formalization and the agents in the natural domain, that is a one-to-one mapping that preserves composition.

We have then developed a more detailed domain, in which sequences are used to emphasize the order relationships between inputs and outputs that allows us to build the

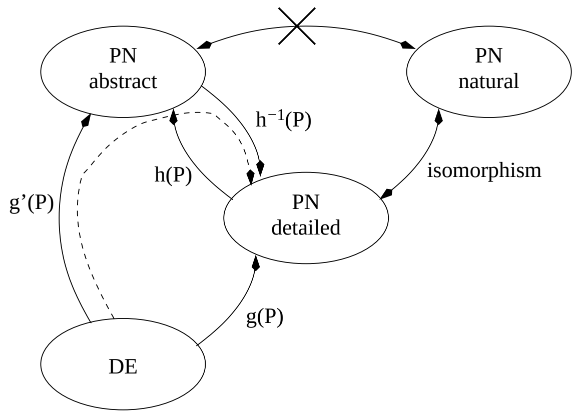

Fig. 8. Relations between trace structure algebras

fixed-point in the composition. By doing this we abandon the isomorphism of the traces with the domain of streams. To be classified as process network agents, trace structures in this formalization must satisfy constraints that ensure that a function on stream is in fact being constructed. The discussion then suggests that there is an isomorphism (which preserves the operation of composition) between the detailed model of trace structures and the agents in the natural semantic domain.

Recall that traces in the abstract process network algebra belong to the set:

$$\mathcal{B}(A) = A \rightarrow V^{\infty}.$$

Traces in the more detailed algebra belong to the other set:

$$\mathcal{B}(A) = (A \times V)^{\infty}.$$

As shown in Section V-B, traces in this more detailed model can be mapped into traces in the more abstract model by virtue of a homomorphism h that removes the order relationships across signals. When naturally extended to trace structures (i.e. to set of traces), h maps agents in the detailed domain into agents in the abstract domain. The homomorphism on individual traces is obviously not one-to-one. However, when considered as a mapping of trace structures from the (restricted set of agents in the) detailed trace structure algebra into the more abstract algebra, the function is a one-to-one mapping. In fact, if two trace structures T_1 and T_2 map into the same trace structure in the abstract algebra, then they must have the same fundamental mode representation P_0. The inductive construction of Equation 7 then shows that $T_1 = T_2$. Because h is one-to-one when applied to agents, there is an inverse function h^{-1} from the abstract trace structure algebra into the detailed algebra.

The relationships between the discrete event and the process network model of computation can be described as a mapping to one of the two formulations. Recall that traces in the discrete event model are of the form:

$$\mathcal{B}(A) = (A \times V \times \mathbb{R}^{\neq})^{\infty}.$$

A straightforward mapping can be constructed from the discrete event traces to the detailed process network traces. The mapping is a function g that simply removes the time stamp from the sequence. In other words, if

$$x = \langle (a_0, v_0, t_0), (a_1, v_1, t_1), \ldots \rangle$$

is a discrete event trace, then

$$g(x) = \langle (a_0, v_0), (a_1, v_1), \ldots \rangle.$$

This mapping is a homomorphism on traces, in that it commutes with the operations of projection and renaming. In other words, if x is a discrete event trace, then

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

The natural extension to sets of traces $g(P)$ of the homomorphism g is a function that maps discrete event agents into process network agents. This function is an upper bound Ψ_u of a conservative approximation:

$$\Psi_u(T) = (\gamma, g(P)).$$

For the lower bound Ψ_l we must map to a restricted set of traces. Namely, the inverse image of $\Psi_l(P)$ should map to traces that are *only* in P. This can be accomplished using the homomorphism g as follows:

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

where $\mathcal{B}(A) - P$ is the complement of P with respect to the universe of traces. This lower bound can be made tighter by considering only the traces that occur in the agents that form the trace structure algebra.

It can be shown that the two mappings so defined form a conservative approximation. This formulation can be generalized. In fact, nothing in the derivation of Ψ_u and Ψ_l depends on the particular models of computation considered. Hence, whenever there is a homomorphism g between the sets of traces of two different models of computation, we can construct a conservative approximation using the same formulation. We refer the reader to [4] for more details on this technique.

What does this mapping look like? Consider for example the inverter shown in Figure 9. It has an input a and an output b. If we assume the inverter has a constant positive delay δ, then a possible trace of the agent in the discrete event model might look like the following:

$$x = \langle (a, 0, 0), (b, 1, \delta), (a, 1, 3.5), (b, 0, 3.5 + \delta), \ldots \rangle,$$

assuming that $\delta < 3.5$. The corresponding trace in the process networks model is

$$x' = \langle (a, 0), (b, 1), (a, 1), (b, 0), \ldots \rangle.$$

This trace is included in the upper bound computed by Ψ_u. If the agent does not contain a trace for any possible delay δ, then this trace is not included in the lower bound Ψ_l. In fact, a trace y with a similar sequence of events, but different delay, would be in $\mathcal{B}(A)$ but not in P; because g discards the delays, $g(x) = g(y)$ and, by definition of Ψ_l above, x is removed from the mapping. In other words, the process network model does not distinguish between agents with different delays and we are indeed computing an approximation.

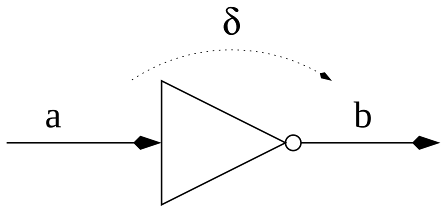

Fig. 9. Inverter agent with delay δ

It is interesting to consider the inverse of this conservative approximation. The inverse mapping corresponds to trying to embed an agent of the process networks model into a discrete event context. Here we must find agents T such that $\Psi_u(T) = \Psi_l(T) = T'$. Because of the particular abstraction we have employed, this occurs whenever the agent T has non-deterministic delay. In this case, given a trace x, all other traces y with the same sequence of events but different delay are included in the set of possible traces of the agent, and therefore retained in the computation of the lower bound. Hence, for every agent T' in the process network model of computation, there exists an agent $T = \Psi_{inv}(T')$ in the discrete event model, where T has the same behaviors as T' and chooses non-deterministically the delay of the outputs. Any deterministic implementation of this embedding will therefore have to make an upfront choice regarding the timing of the agent.

The functions Ψ_u and Ψ_l that we have just defined certainly constitute an abstraction. However, in this particular case the abstraction does not ensure that the corresponding trace structure in the process network algebra satisfies the constraints for that model defined in Section V-B involving equation 7. In fact, for each trace in the discrete event model there should correspond several (possibly infinitely many) traces in the process network model that include all possible delayed outputs. It is possible to consider only a restricted version of the discrete event trace structures that maps correctly in the detailed process network algebra. To simplify this task, we will take an alternative route and use the abstract process network algebra as an intermediate step.

Notice that the abstract process network trace structure algebra requires that agents be monotonic and functional. This requirement must still be satisfied by the discrete event agent that we want to abstract. An equivalent constraint that can be imposed at the discrete event level is that of receptiveness. Intuitively, a trace structure is receptive if it can't constrain the value of its inputs. The technical definition of receptiveness (see [11]) requires the device of infinite games: an agent is receptive if it can always respond to an input with outputs that make the trace one of its possible traces.

We can show that if a discrete event agent T is both receptive and functional, then it is also monotonic (where the prefix order corresponds to the usual prefix on sequences). In fact, assume it is not monotonic. Then there are traces x and y in T such that $\mathit{proj}(I)(x) \sqsubseteq \mathit{proj}(I)(y)$, but $\mathit{proj}(O)(x) \not\sqsubseteq \mathit{proj}(O)(y)$. But if T is receptive, then x can be extended to a trace x' such that $\mathit{proj}(I)(x') = \mathit{proj}(I)(y)$ and $x' \in T$. By the functionality assumption, $x' = y$. But $x \sqsubseteq x'$, a contradiction. Hence T must be monotonic.

A homomorphism g' between the discrete event traces and the abstract process network traces is given by the composition of g and h. The natural extension to sets gives us a mapping $g'(P)$ on trace structures and a corresponding conservative approximation. An approximation from the discrete event trace structure algebra to the detailed process networks trace structure algebra can now be constructed by taking the composition of the mapping g' and h^{-1} as shown in Figure 8.

B. CT to DE

To construct an approximation from the continuous time to the discrete event model of computation we must first define the notion of an event at the level of the continuous time traces. Abstraction, in this case, can be done in several ways. One, for example, is to consider an event as the snapshot of the state at certain regular intervals. Another technique consists of abstracting the value domain, and identify an event whenever the signals cross certain discrete thresholds. In this paper we take yet another approach, and identify an event whenever any of the signals changes with respect to its previous value. Here, the notion of "previous" value must be made precise, since in general there is no immediate predecessor in the continuos real time.

In the continuous time model signals may change value simultaneously. In the discrete event model, on the other hand, events are totally ordered, even when they have the same time stamp. Hence, after identifying an event, we must also decide how to order simultaneous events in the same time stamp. Because there is no obvious choice, we map each event in continuous time to the set of all possible orderings in discrete event. This choice implies that for each trace in the continuous time model there correspond several traces in the discrete event model. Consequently, the approach based on the homomorphism on traces outlined in the previous section will not work.

To formalize these notions we start from the concept of an event. Intuitively, an event occurs when any of the signals changes value. It is easier, however, to define the opposite condition, i.e. when all signals are constant. If $f : \mathbb{R}^{\not-} \to V$ is a function over the reals, we say that f is *stable at* $t_0 \in \mathbb{R}^{\not-}$ if and only if there exists an $\epsilon > 0$ such that f is constant on the interval $(t_0 - \epsilon, t_0]$. Recall now the definition of a continuous time trace:

$$\mathcal{B}(A) = \mathbb{R}^{\not-} \to (A \to V).$$

We say that a trace $x \in \mathcal{B}(A)$ is *stable on signal* $a \in A$ *at* $t_0 \in \mathbb{R}^{\not-}$ if and only if the function $f(t) = x(t, a)$ is stable at t_0. We say that a continuous time trace x *has an event for signal* $a \in A$ *at* t_0 whenever x is not stable on a at t_0. Because at time $t_0 = 0$ there is no left interval, we always assume that a trace has an event at time 0 for all signals.

To construct a trace in the discrete event model we must create a sequence where each element corresponds to an event for some signal at some time in continuous time. To simplify the task, we introduce two additional, and somewhat more elaborate, trace algebras for the discrete event model.

In the first trace algebra, we construct a "sequence" by taking the set of reals as an index set, and by mapping the index set to sequences of events that represent the delta cycles for each particular time stamp. An empty sequence of delta cycles denotes the absence of events for the particular time stamp. Formally, we define the set of possible traces as:

$$\mathcal{B}(A) = \mathbb{R}^{\not-} \to (A \times V)^{\infty},$$

where A is, as usual, the set of signals, and V is the corresponding set of possible values. This formulation clearly includes systems that are not discrete: imagine, for instance, that the sequence corresponding to the delta cycles is non-empty for every $t \in \mathbb{R}^{\not-}$. Thus we must further restrict the set of possible traces to only those whose set of non-empty time stamps is discrete, as was discussed in Section V-C.

Projection and renaming are defined as expected. Their formal definition gives us the opportunity to introduce a construction theorems that allows one to build new trace algebras from existing ones. In this particular case, note how the set of traces is defined as a function whose range is the set of traces defined in Section V-A for the CSP model. The following theorem proves that when projection and renaming are defined appropriately, the result is always another trace algebra.

Theorem 1. *Let $\mathcal{C}' = (\mathcal{B}'(A), \mathit{proj}, \mathit{rename})$ be a trace algebra and let Z be a set. Then the trace algebra $\mathcal{C}$ such that:*

$$\begin{aligned} \mathcal{B}(A) &= Z \to \mathcal{B}'(A), \\ \mathit{proj}(B)(x) &= \lambda d \in Z[\mathit{proj}(B)(x(d))], \\ \mathit{rename}(r)(x) &= \lambda d \in Z[\mathit{rename}(r)(x(d))], \end{aligned}$$

is a trace algebra.

In our particular case we let $\mathcal{B}'(A) = (A \times V)^{\infty}$, $Z = \mathbb{R}^{\not-}$ and projection and renaming as defined in Section V-A. Hence for a trace $x \in \mathbb{R}^{\not-} \to (A \times V)^{\infty}$ we have

$$\begin{aligned} \mathit{proj}(B)(x) &= \lambda t \in \mathbb{R}^{\not-}[\mathit{proj}(B)(x(t))], \\ \mathit{rename}(r)(x) &= \lambda t \in \mathbb{R}^{\not-}[\mathit{rename}(r)(x(t))]. \end{aligned}$$

A trace structure has again signature $\gamma = (I, O)$ and is otherwise obtained as usual as a set of traces.

The second trace algebra that we introduce is similar to the one just presented, but without ordering information within a time stamp. Then we build a mapping from each of the new traces to a set of discrete event traces, that contain all possible interleavings of the events.

Recall (see above) that traces in the discrete event model of computation are of the form:

$$\mathcal{B}(A) = \mathbb{R}^{\not<} \to (A \times V)^{\infty}.$$

The ordering information in the sequence of delta cycles can be removed by considering the more abstract set of traces:

$$\mathcal{B}'(A) = \mathbb{R}^{\not<} \to 2^{A \times V}.$$

It is easy to construct a function h from $\mathcal{B}$ to $\mathcal{B}'$ that removes the ordering information. If $x \in \mathcal{B}(A)$ is of the form $x = x(t, n)$, we define $x' = h(x)$ as the trace $x' = x'(t)$ such that for all $t \in \mathbb{R}^{\not<}$

$$x'(t) = \{(a, v) \in A \times V : \exists n \in \mathbb{N}[x(t, n) = (a, v)\}.$$

It is easy to show that h is well defined, and that it is onto. However h is not one-to-one, so that its inverse h^{-1} maps a single trace $x' \in \mathcal{B}'(A)$ to a set of traces in $\mathcal{B}(A)$. This set of traces corresponds to all possible interleavings of the set of pairs of signals and values, with or without repetitions.

It is now easy to define a function g from traces in the continuous time to traces in the discrete event model without ordering. If $y = y(t, a)$ is a continuous time trace, then define $x' = g(y)$ as the trace $x' = x'(t)$ such that for all $t \in \mathbb{R}^{\not<}$

$$\begin{aligned} x'(t) \quad = \quad & \{(a, v) \in A \times V : x \text{ has an event on} \\ & \text{signal } a \text{ at time } t \wedge x(t, a) = v\}. \end{aligned}$$

We can now define an approximation between the continuous time and the discrete event model based on the functions g and h.

Let $T = (\gamma, P)$ be a trace structure in the continuous time model. To build an upper bound we naturally extend the functions g and h to sets of traces as follows:

$$\Psi_u(T) = (\gamma, h^{-1}(g(P))).$$

A lower bound could be constructed in several ways. Note, however, that without any further constraint the discrete event model can represent continuous functions exactly. In other words, since our mapping on trace structures is actually one-to-one, it does not constitute an abstraction. The obvious choice in this case is therefore to simply have

$$\Psi_l(T) = \Psi_u(T),$$

for all T.

The key to getting a real abstraction is that of defining exactly the conditions that make the discrete event model *discrete*. This can be done by replacing the set of reals in the definition of the trace algebra with a different set D. The result is a parametrized trace algebra

$$\mathcal{B}(A) = D \to (A \times V)^{\infty}.$$

Depending on the choice of D different kinds of abstractions are possible.

VII. Conclusions

We presented the theoretical foundation of the Metropolis project whose goal is to build an environment where the design of complex systems will be a matter of days versus the many months needed today.

All the models of importance “reside” in a unified framework 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 this unified approach is 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.

We used trace algebra to provide the underlying mathematical machinery. In particular, we showed how to formalize within our framework the natural semantic domain of a model of computation based on continuous time and differential equations. Then we introduced the formalization of the semantic domain for four more commonly used models of computation. In all cases, we followed the same pattern by first presenting the natural formalization, and then the formalization in terms of trace algebras. For each model of computation we also sketched an example of its typical applications.

For each model we have also suggested a particular application in the context of a system similar to the PicoRadio project. In order to understand the behavior and the properties of the entire system, we needed to understand the interplay between the different subsystems. We accomplished this by relating the semantic domains that we have developed in this paper and studied how the different notions of computation fit together.

We believe that this framework is 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.

References

[1] The rosetta web site. http://www.sldl.org.

[2] 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.

[3] F. Balarin, L. Lavagno, C. Passerone, A. S. Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogeneous systems. In J. Cortadella and A. Yakovlev, editors, *Advances in Concurrency and System Design*. Springer-Verlag, 2002.

[4] J. R. Burch, R. Passerone, and A. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In M. Koutny and A. Yakovlev, editors, *Application of Concurrency to System Design*, 2001.

[5] J. R. Burch, R. Passerone, and A. Sangiovanni-Vincentelli. Using multiple levels of abstraction in embedded software design. In M. A. Henzinger and C. M. Kirsch, editors, *First International Workshop, EMSOFT 2001*, volume 2211 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[7] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[8] J. L. da Silva Jr., j. Shamberger, M. J. Ammer, C. Guo, S. Li, R. Shah, T. Tuan, M. Sheets, J. M. Rabaey, B. Nikolic, A. L. Sangiovanni-Vincentelli, and P. Wright. Design methodology for picoradio networks. In *Proceedings of the Design Automation and Test in Europe*, Munich, Germany, March 2001.

[9] 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.

[10] J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. 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. 2001.

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

[12] 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.

[13] T. Henzinger. Masaccio: a formal model for embedded components. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *TCS 00: Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 549–563. Springer-Verlag, 2000.

[14] T. Henzinger, M. Minea, and V. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In M. di Benedetto and A. Sangiovanni-Vincentelli, editors, *HSCC 00: Hybrid Systems—Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 275–290. Springer-Verlag, 2001.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[16] 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.

[17] R. Negulescu. *Process Spaces and the Formal Verification of Asynchronous Circuits*. PhD thesis, University of Waterloo, Canada, 1998.

[18] R. Negulescu. Process spaces. In C. Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[19] J. Rabaey, M. Ammer, J. S. jr., D. Patel, and S. Roundy. Picoradio supports ad hoc ultra-low power wireless networking. *IEEE Computer Magazine*, July 2000.

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