# Modeling Techniques in Design-by-Refinement Methodologies (Extended Abstract)

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

[1]Cadence Berkeley Laboratories, Berkeley CA 94704
{jrb,robp}@cadence.com
[2]Department of EECS, University of California at Berkeley, Berkeley CA 94720
alberto@eecs.berkeley.edu

September 25, 2018

**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 introduce such a semantic foundation and describe how it can integrate several 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.

## 1 Introduction

Microscopic devices, powered by ambient energy in their environment, will be able to sense numerous fields, position, velocity, and acceleration, and communicate with appropriate and sometimes substantial bandwidth in the near area. Larger, more powerful systems within the infrastructure will be driven by the continued improvements in storage density, memory density, processing capability, and system-area interconnects as single board systems are eclipsed by complete systems on a chip. Data movement and transformation is of central importance in such applications. Future devices will be network-connected, channeling streams of data into the infrastructure, with moderate processing on the fly. Others will have narrow, application-specific 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 [11, 12]. 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., [3, 24]) 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 ([4]) 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 [4], 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. In particular we are studying the semantic domain of several of the models of computation of interest, and are

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 model of computation. The goal is to have a general framework that encompasses many models of computation, including different models of time and communication paradigms, and yet structured enough to give us results that apply regardless of the particular model in question. At the same time, the framework offers mathematical "tools" to help build new semantic domains from existing ones. Because the framework is based on algebraic structures, the results are independent of any particular design language, and therefore are not just specific to the Metropolis meta-model.

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.

In the following sections we first survey some related work. We then introduce our framework and concentrate on the basic principles underlying the definitions. We refer the reader to our previous publications [21, 14, 20, 23, 6, 7, 2, 8, 22] for an in depth presentation of specific examples of semantic domains.

## 2 Related Work

Several formal models have been proposed over the years (see e.g. [13]) 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 [17]. 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 [11, 12]. 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 de-emphasize executability. Instead, we are more concerned with studying the process of abstraction and refinement in abstract terms. For example, it is easy in our framework to model the non-deterministic behavior that emerges when an abstract model is embedded into a more detailed model. Any executable framework would require an upfront choice that would make the model deterministic, potentially hiding some aspects of the composition.

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 [18, 19] 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 [9, 10]. 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. [15, 16]. 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.

# 3 Overview

This section presents the basic framework we use to construct semantic domains, which is based on trace algebras and trace structure algebras. This overview is intended to highlight the relationships between the concepts. A detailed formal definition can be found in [5, 6, 8].

## 3.1 Traces and Trace Structures

The models of computation in use for embedded concurrent systems represent a design by a collection of agents (processes, actors, modules) that interact to perform a function. For any particular input to the system, the agents react with some particular execution, or behavior. In our framework we maintain a clear distinction between models of agents and models of individual executions. In different models of computation, individual executions can be modeled by very different kinds of mathematical objects. We always call these objects *traces*. A model of an agent, which we call a *trace structure*, consists primarily of a set of traces. This is analogous to verification methods based on language containment, where individual executions are modeled by strings and agents are modeled by sets of strings. However, our notion of trace is quite general and so is not limited to strings.

Traces often refer to the externally visible features of agents: their actions, signals, state variables, etc. We do not distinguish among the different types, and we refer to them collectively as a set of *signals* $W$. Each trace and each trace structure is then associated with an *alphabet* $A \subseteq W$ of the signals it uses.

We make a distinction between two different kinds of behaviors: *complete* behaviors and *partial* behaviors. A complete behavior has no endpoint. 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.

As an example we may consider traces that are suitable for modeling continuous time, synchronous discrete time and transformational (i.e. non-reactive) systems. In the first case, we might define a trace as a mapping that associates to each signal a function from continuous time (the positive reals) to an appropriate set of values (for example, the reals again). Hence, for an alphabet $A$, the traces are defined by

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

A partial trace in this case is a mapping that associates functions from only a closed time interval $[0, \delta]$, $\delta > 0$:

$$\mathcal{B}(A) = A \to ([0, \delta] \to \mathbb{R}).$$

We refer to these traces as *metric-time* traces.

An example of a trace suitable for synchronous discrete time systems is a sequence. In this example we assume the signals represent events that occur at distinct instants in time. The corresponding trace is a sequence whose elements are subsets of the set of signals (events) available from the alphabet:

$$\mathcal{B}(A) = (2^A)^\infty,$$

where the notation $^\infty$ denotes both finite and infinite sequences. Here the partial traces are the finite sequences, while the complete traces are the infinite sequences. We refer to these traces as *synchronous* traces.

Unlike the previous two examples, a transformational system is only concerned with the initial and final state of a computation. If the signals are interpreted as state variables, a corresponding trace may be defined as a pair of mappings associating the state to its initial and final value (from a set of values $V$), respectively:

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

The case of non-termination is modeled by adding a distinctive value $\perp$ to the set $V$. If we denote with $V_\perp = V \cup \{\perp\}$, complete traces can be defined as

$$\mathcal{B}(A) = (A \to V) \times (A \to V_\perp).$$

We refer to these traces as *pre-post* traces.

Note that 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 terminating trace above can represent both a complete behavior that terminates or it can represent a partial behavior.

### 3.2 Trace Algebra and Trace Structure Algebra

In our framework, the first step in defining a model of computation is to construct a trace algebra. The carrier of a trace algebra contains the universe of partial traces and the universe of complete traces for the model of computation. The algebra also includes three operations on traces: *projection*, *renaming* and *concatenation*. 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 second step is to construct a trace structure algebra. Here each element of the algebra is a trace structure, which consists primarily of a set of traces from the trace algebra constructed in the first step. Given a trace algebra, and the set of trace structures to be used as the universe of agent models, a trace structure algebra is constructed in a fixed way. Thus, constructing a trace algebra is the creative part of defining a model of computation. Constructing the corresponding trace structure algebra is much easier. A trace structure algebra includes four operations on agents: *projection*, *renaming*, *parallel composition* and *sequential composition*.

The relationships between trace algebras and trace structure algebras are depicted in Figure 1. This figure also shows the relationships between different algebras that we will discuss later in the paper.

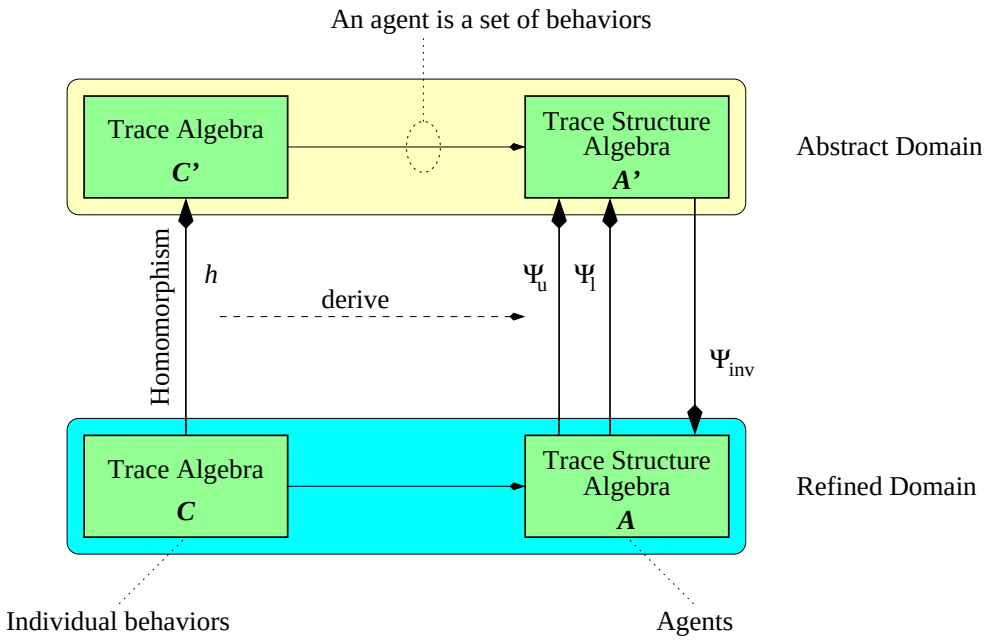


Figure 1: Algebras and their relationships

The first operation is called *projection*, and consists in retaining from a trace only the information related to certain signals. In our framework, the projection operation *proj* takes as argument the set of signals $B$ that should be retained. For metric-time traces, projection corresponds to restricting the mapping from the set $A$ of signals to a subset $B$. This operation is defined similarly for pre-post traces. Conversely, for synchronous traces projections consists of removing the events not in $B$ from each element of the sequence.

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 signals of the agent. In other words, the scope of the hidden signals is limited to the agent they belong to.

The second operation is called *renaming*, and consists in changing the names of the visible elements of a trace or an agent. In our framework, the renaming operation *rename* takes as argument a renaming function $r$ that maps the elements of the alphabet $A$ into a new alphabet $C$, 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.

In all examples, renaming corresponds to a substitution of signals, and is easily defined in terms of $r$ or of its inverse. For example, if $x$ is a synchronous trace $\langle A_1, A_2, \ldots \rangle$, then $\mathit{rename}(r)(x) = \langle r(A_1), r(A_2), \ldots \rangle$, where $r$ is naturally extended to sets of signals.

As for projection, renaming of trace structures can be seen as the natural extension to sets of traces of the corresponding operation on individual traces. The effect is that of a renaming of all the signals in the agent: 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 all trace algebras and all trace structure algebras (and therefore, for all models of computation that fit into our framework) we require that the operations on traces and trace structures 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 signals results in the very same trace or agent. 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.

In the algebra of trace structures we introduce the additional operation of parallel composition. Note that this operation has no counterpart in the trace algebra. Intuitively, parallel composition corresponds to having several agents run concurrently by sharing some common signals. The result of the parallel composition is one agent that alone acts as the combination of the agents being composed. Let $T_1 = (A_1, P_1)$ and $T_2 = (A_2, P_2)$ be two agents that we want to compose, where $A_1$ and $A_2$ are the alphabets, and $P_1$ and $P_2$ the set of traces. The alphabet of the parallel composition $T = T_1 \parallel T_2$ must include all signals from $T_1$ and $T_2$, so that $A = A_1 \cup A_2$. The set $P$ of traces of $T$ must be "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. Formally:

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

Similarly to projection and renaming, parallel composition of agents must satisfy certain properties. For example we require that it be commutative and associative, and that it behaves consistently when used in combination with projection and renaming. Note that these operations are predefined for a trace structure algebra. 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.

While parallel composition is at the basis of concurrent models of computation, in other models 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. In the case of synchronous traces, concatenation corresponds to the usual concatenation on sequences. Similarly we can define concatenation for metric-time traces. For pre-post traces, concatenation is defined only when the final state of the first trace matches the initial state of the second trace. The resulting trace has the initial state of the first component and the final state of the second. Note that the information about the intermediate state is lost.

Similarly to the other operations, concatenation must also satisfy certain properties that ensure that its behavior is consistent with its intuitive interpretation. For example we require that it be associative (but not commutative!), and that it behaves consistently when used in combination with projection and renaming. Concatenation induces a corresponding operation on trace structures that we call *sequential composition* by naturally extending it to sets of traces. A more detailed account of sequential composition can be found in [7].

# 4 Refinement and Conservative Approximations

In verification and design-by-refinement methodologies a specification is a model of the design that embodies all the possible implementation options. Each implementation of a specification is said to *refine* the specification. In our framework, each trace structure algebra (semantic domain of a model of computation) has a refinement order that is based on trace containment. We say that an agent $T_1$ refines an agent $T_2$, written $T_1 \subseteq T_2$, if the set of traces of $T_1$ is a subset of the set of traces of $T_2$. Intuitively, this means that the implementation $T_1$ can be substituted for the specification $T_2$. It is easy to show that the refinement relationships constitutes a (pre)order on the set of trace structures.

Proving that an implementation refines a specification is often a difficult task. Most techniques decompose the problem into smaller ones that are simpler to handle and that produce the desired result when combined. To make this approach feasible, the operations on the agents must be monotonic with respect to the refinement order. The definitions given in the previous section make sure that this is the case for our semantic domains.

An even more convenient approach to the above verification consists of translating the problem into a different, more abstract semantic domain, where checking for refinement of a specification is presumably more efficient. A *conservative approximation* is a mapping of agents from one trace structure algebra to another, more abstract, algebra that serves that purpose. The two trace structure algebras do not have to be based on the same trace algebra. Thus, conservative approximations are a bridge between different models of computation. We will refer to Figure 1 for the rest of the exposition.

A conservative approximation is actually composed of two mappings. The first mapping is 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 $\Psi_u$. 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 $\Psi_l$.

Conservative approximations are abstractions that maintain a precise relationship between verification results in the two trace structure algebras. 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:

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

Thus, when used in combination, the two mappings allow us to relate results in the abstract domain to results in the more detailed domain. The conservative approximation guarantees that this will not

lead to a false positive result, although false negatives are possible. We refer to our previous work for more details ([6]) and for several examples ([7, 8]).

### 4.1 Homomorphisms and Conservative Approximations

Defining a conservative approximations and proving that it satisfies the definition can sometimes be difficult. However, a conservative approximation between trace structure algebras can be derived from a homomorphism between the underlying trace algebras.

A homomorphism $h$ is a function between the domains of two trace algebras that commutes with projection, renaming and concatenation. Consider two trace algebras $\mathcal{C}$ and $\mathcal{C}'$. Intuitively, if $h(x) = x'$ the trace $x'$ is an abstraction of any trace $y$ such that $h(y) = x'$. Thus, $x'$ can be thought of as representing the set of all such $y$. Similarly, a set $X'$ of traces in $\mathcal{C}'$ can be thought of as representing the largest set $Y$ such that $h(Y) = X'$, where $h$ is naturally extended to sets of traces. If $h(X) = X'$, then $X \subseteq Y$, so $X'$ represents a kind of upper bound on the set $X$. Hence, if $\mathcal{A}$ and $\mathcal{A}'$ are trace structure algebras constructed from $\mathcal{C}$ and $\mathcal{C}'$ respectively, we use the function $\Psi_u$ that maps an agent with traces $P$ in $\mathcal{A}$ into the agent with traces $h(P)$ in $\mathcal{A}'$ as the upper bound in a conservative approximation. A sufficient condition for a corresponding lower bound is: if $x \notin P$, then $h(x)$ is not in the set of possible traces of $\Psi_l(T)$. This leads to the definition of a function $\Psi_l(T)$ that maps $P$ into the set $h(P) - h(\mathcal{B}(A) - P)$. The conservative approximation $\Psi = (\Psi_l, \Psi_u)$ is an example of a *conservative approximation induced by* $h$. A slightly tighter lower bound is also possible (see [5]).

It is straightforward to take the general notion of a conservative approximation induced by a homomorphism, and apply it to specific models. Simply construct trace algebras $\mathcal{C}$ and $\mathcal{C}'$, and a homomorphism $h$ from $\mathcal{C}$ to $\mathcal{C}'$. Recall that these trace algebras act as models of individual behaviors. One can construct the trace structure algebras $\mathcal{A}$ over $\mathcal{C}$ and $\mathcal{A}'$ over $\mathcal{C}'$, and a conservative approximation $\Psi$ induced by $h$. Thus, one need only construct two models of individual behaviors and a homomorphism between them to obtain two trace structure models along with a conservative approximation between the trace structure models.

### 4.2 Inverses of Conservative Approximations

Conservative approximations represent the process of abstracting a specification in a less detailed semantic domain. Inverses of conservative approximations represent the opposite process of refinement.

Let $\mathcal{A}$ and $\mathcal{A}'$ be two trace structure algebras, and let $\Psi$ be a conservative approximation between $\mathcal{A}$ and $\mathcal{A}'$. Normal notions of the inverse of a function are not adequate for our purpose, since $\Psi$ is a pair of functions. We handle this by only considering the $T$ in $\mathcal{A}$ for which $\Psi_u(T)$ and $\Psi_l(T)$ have the same value $T'$. Intuitively, $T'$ represents $T$ exactly in this case, hence we define $\Psi_{inv}(T') = T$. When $\Psi_u(T) \neq \Psi_l(T)$ then $\Psi_{inv}$ is not defined.

The inverse of a conservative approximation can be used to embed a trace structure algebra at a higher level of abstraction into one at a lower level. Only the agents that can be represented exactly at the high level are in the image of the inverse of a conservative approximation. We use this as part of our approach for reasoning about heterogeneous systems that use models of computation at multiple levels of abstraction. Assume we want to compose two agents $T_1'$ and $T_2'$ that reside in two different trace structure algebras $\mathcal{A}_1$ and $\mathcal{A}_2$. To make sense of the composition, we first define a third, more detailed trace algebra that has homomorphisms into the other two. Thus we can construct a third, more detailed, trace structure algebra $\mathcal{A}$ with conservative approximations induced by the homomorphisms. The inverse of these conservative approximations are used to map $T_1'$ and

$T_2'$ into their corresponding detailed models $T_1$ and $T_2$. The composition then takes place in the detailed trace structure algebra. Figure 2 illustrates this procedure.

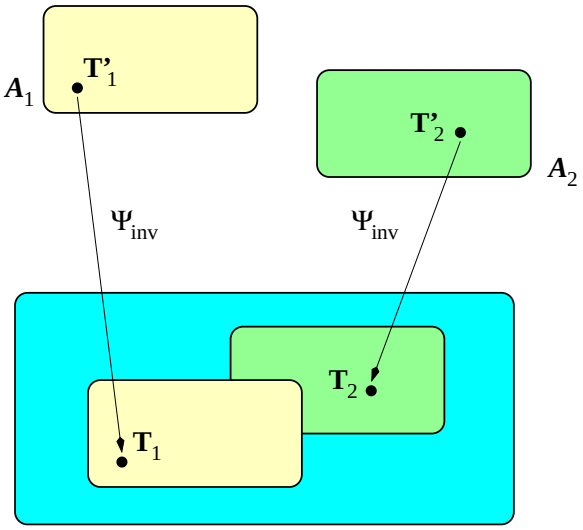


Figure 2: Composition of heterogeneous models

# 5 Conclusions

In this extended abstract with have presented the framework of trace algebras, whose goal is to make it easy to define and to study the relationship between the semantic domains for a wide range of models of computation. 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. Our representation of agents is denotational, in that no rule is given to derive the output from the input. The algebraic infrastructure allows us to to formalize a semantic domain in a way that is close to a natural semantic domain for a model of computation. In addition it introduces additional concepts such as hierarchy, instantiation and scoping in a natural and consistent way.

In particular we are concentrating on using the concept of a conservative approximation to study the problem of heterogeneous interaction. Our current research also focuses on a generalized notion of refinement as substitutability. General results that relate the refinement relationships to the other functions of the algebra, including conservative approximations, will provide powerful tools for the development of verification and synthesis techniques that are independent of the particular model of computation.

# References


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

[2] F. Balarin, J. R. Burch, L. Lavagno, Y. Watanabe, R. Passerone, and A. L. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop*, HLDVT01, pages 129–133, Monterey, CA, November 7–9, 2001. IEEE Computer Society, Los Alamitos, CA, USA.

[3] 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.

[4] F. Balarin, L. Lavagno, C. Passerone, A. L. 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.

[5] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Aug. 1992.

[6] 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.

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

[8] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Modeling techniques in design-by-refinement methodologies. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, June 23-28 2002.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, and A. Sangiovanni-Vincentelli. Intellectual property re-use in embedded system co-design: An industrial case study. In *Proceedings of the* $11^{th}$ *International Symposium on System Synthesis*, ISSS98, pages 37–42, Hsinchu, Taiwan, December 2–3, 1998. IEEE Computer Society, Los Alamitos, CA, USA.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] C. Passerone, C. Sansoé, L. Lavagno, R. McGeer, J. Martin, R. Passerone, and A. L. Sangiovanni-Vincentelli. Modeling reactive systems in Java. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):515–523, October 1998.

[21] R. Passerone. Automatic synthesis of interfaces between incompatible protocols. Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720, December 1997.

[22] R. Passerone, L. d. Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the* $20^{th}$ *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD02, pages 132–139, San Jose, California, November 10–14, 2002. IEEE Computer Society, Los Alamitos, CA, USA.

[23] R. Passerone, J. A. Rowson, and A. L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the* $35^{th}$ *Annual Conference on Design Automation*, DAC98, pages 8–13, San Francisco, CA, USA, June 15–19, 1998. ACM Press, New York, NY, USA.

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