Approximating Behaviors in Embedded System Design

Roberto Passerone[1] and Alberto L. Sangiovanni-Vincentelli[2]

[1] Dipartimento di Ingegneria e Scienza dell'Informazione,
University of Trento, Trento, Italy
[2] Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley, CA

Abstract. Embedded systems are electronic devices that function in the context of a physical environment, by sensing and reacting to a set of stimuli. To simplify the design of embedded systems, different parts are best described using different notations and analyze with different techniques, i.e., the system is said to be *heterogeneous*. We informally refer to the notation and the rules that are used to specify and verify the elements of heterogeneous systems and their collective behavior as a *model of computation*. In this paper, the use of *conservative approximations* (recently introduced by the authors) is reviewed to establish relationships between different models of computation in a design. After presenting the basic definitions, we propose three different models at different levels of abstraction for describing a system and the progression towards its implementation. Then, we derive associated conservative approximations starting from simple homomorphisms between sets of behaviors of the different models.

1 Introduction

Embedded systems are electronic devices that function in the context of a physical environment, by sensing and reacting to a set of stimuli. To simplify the design of an embedded system, its different parts are best described using different notations and analyzed with different techniques. In this case, we say that the system is *heterogeneous*. For example, the model of the software application that runs on a distributed collection of nodes in a sensor network is often concerned only with the initial and final state of the behavior of a reaction. In contrast, the particular sequence of actions of the reaction could be relevant to the design of one instance of a node. Likewise, the notation used in reasoning about a resource management subsystem is often incompatible with the handling of real time deadlines, typical of communication protocols. This form of heterogeneity is also reflected in the structure of the design teams, which increasingly consist of highly specialized groups that focus on the solution of a particular task, under the direction of system architects.

Designers benefit from this separation. First, the system is naturally partitioned into smaller and more manageable parts. Secondly, and more importantly, designers are free to select for each subsystem the rules that are used to specify its behavior as a hierarchical collection of modules (*composition*), and to verify that such behavior conforms to a specification (*refinement verification*). These rules vary widely across different modeling domains, such as the ones outlined above. The restrictions and the intrinsic properties of these rules, which we collectively refer to as a *model of computation*, are the

P. Degano et al. (Eds.): Montanari Festschrift, LNCS 5065, pp. 721–742, 2008.

basis of domain specific techniques that can be used to guarantee the correctness of the implementation in an easier way.

While specified separately, subsystems must eventually interact to form the system behavior, and will in fact do so in the physical implementation. However, system designers are typically interested in not waiting until the final stages of the implementation to validate the system functionality and performance metrics, because the cost of fixing design and specification errors increases dramatically in the later phases of the design flow as amply documented for electronic systems, software and integrated circuits. The costs associated with late discovery of errors and, in particular, of integration errors, have risen to a point that they are no longer sustainable. To witness, consider the recent recalls by Mercedes-Benz of 1.5 million cars for problems with the braking subsystem. Consequently, the ability of the system designer to specify, manage, and verify the functionality and performance of *concurrent behaviors*, within *and across* heterogeneous boundaries, is essential. Most design methodologies that address these problems are based on the processes of abstraction and refinement, that is, of the application of maps that convert and relate different models of computation. However, crossing the boundaries between abstraction levels by abstracting and refining a specification is often not trivial. The most common pitfalls include mishandling of corner cases and inadvertently misinterpreting changes in the communication semantics.

These problems arise because of the poor understanding and the lack of a precise definition of the abstraction and refinement maps used in the flow, which are therefore likely to provide little, if any, guarantee of satisfying a given set of constraints and specifications, without resorting to extensive simulation or tests on prototypes. However, in the face of growing complexity, this approach will have to yield to more rigorous methods. In addition, abstraction and refinement should be designed to preserve, whenever possible, the properties of the design that have already been established. This is essential to increase the value of early, high level models and to guarantee a speedier path to implementation.

In this paper we review abstraction and refinement relationships in the form of *conservative approximations* [3,17,18] introduced by the authors to approach the problem of abstraction and refinement from a formal standpoint. Conservative approximations are closely related to abstract interpretations, and, in addition, preserve refinement verification from an abstract to a concrete model while avoiding the occurrence of false positive results. This property of an abstraction is useful because, presumably, refinement verification is more efficient at the abstract level than it is at the concrete. In this paper we show how to derive models of computation and the corresponding abstraction and refinement maps starting from simple models of behavior. We focus in particular on models that include both continuous and discrete behaviors, and are therefore appropriate for the design of hybrid systems [4].

The rest of the paper is structured as follows. Section 2 gives an overview of our methodology and formal framework and introduces the basic terminology. A set of different agent models for embedded systems are presented in Section 3. Then, we construct relationships between these models and give a general recipe for deriving conservative approximations in Section 4. Section 5 surveys related work and discusses other forms of abstraction. In all cases, the specific abstraction is either an instance of

an abstract interpretation (and is therefore unsound for refinement verification), or is a particular case of conservative approximations. Finally, Section 6 provides directions for our future research.

2 Methodology Overview

Before we can investigate the notion of an abstraction, we must provide a way to describe its domain and range, namely the models of computation. In general, an abstraction transforms a block of computation in one model into a block of computation in another model. For example, it may transform a module written in a discrete event language (such as Verilog or VHDL) into a transaction level module that ignores the precise time at which events occur, such as a dataflow language. We therefore represent models of computation at the granularity of the module, or block. In other words, a model of computation is simply the set of blocks that can be expressed in the model. For instance, we represent a model of computation based on finite state machines as the set of finite state machines, or a dataflow model of computation as the set of dataflow actors. However, the representation of the blocks need not be in the form of a programming language. In fact, to simplify the task of defining abstraction functions, we typically represent blocks as the *set of behaviors*, or *traces* that they can exhibit. The nature of these traces obviously depends on the particular model of computation: for instance, they may consist of sequences of values (as in the case of synchronous models), functions of real variables (for more accurate continuous time models), or sets of values representing certain performance metrics (power models, constraints). Because we use traces, we will refer to blocks in any model of computation generically as *agents*, and they will be denoted by the letters p and q. 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 agent is then associated with an *alphabet* $A \subseteq W$ of the signals it uses.

We make a distinction between two different kinds of traces: *complete* traces and *partial* traces. A complete trace has no endpoint. A partial trace has an endpoint; it can be a prefix of a complete trace or of another partial trace. Every complete trace has partial traces that are prefixes of it; every partial trace is a prefix of some complete trace. The distinction between a complete trace and a partial trace has only to do with the length of the trace (that is, whether or not it has an endpoint), not with what is happening during the trace. For example, a finite string can represent a complete behavior with a finite number of actions, or it can represent a partial behavior.

In our framework, the first step in defining a model of computation is to construct an algebra of traces $\mathcal{C}$. The 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*. Intuitively, these operations correspond to encapsulation, instantiation and sequential composition, respectively. Projection removes all references to a specified set of signals in a trace, hiding them from an external observer, while renaming is used to change the names of the signals, emulating the replacement of actual for formal parameters in function instantiation. Concatenation, which joins two behaviors at their ends, can be used to

define the notion of a prefix of a trace. We say that a trace x is a prefix of a trace z if there exists a trace y such that z is equal to x concatenated with y.

Agents in a model of computation do not exist in isolation. For instance, agents can be combined with the operation of parallel composition (denoted by the symbol $\|$) to yield a new agent in the *same* model of computation. An *agent algebra* $\mathcal{Q}$ is used to define these operators, together with their set of agents. For trace-based agent models, an agent algebra is constructed in a fixed way from the algebra of the corresponding traces, where agents are simply sets of traces. For composition, the new agent combines the behaviors of the original components in such a way that the new behaviors are consistent with those of the components when projected onto their alphabets. Other operations are derived by simply extending to sets of traces the operators of the trace algebra. We will show examples of these derivations later in Section 3.

Different means can often be used to achieve the same goal. Likewise, agents with different behavior may sometimes yield the same result when applied to a particular context. In particular, if an agent p can always be replaced for an agent q in any context without materially changing the outcome of the composition, then we say that p *refines* q. In the rest of this paper we use the symbol $\preceq$ to denote this refinement relationship, and we write $p \preceq q$ whenever p refines q. We also refer to q as a *specification*, and to p as an *implementation* of q. For trace-based models, refinement can be reduced to checking containment between the trace set of agents, and is therefore analogous to verification methods based on language containment.

2.1 Refinement Preserving Abstractions

The choice of levels of abstraction, or models, in a heterogeneous design methodology is obviously very important. Each model must in fact be capable of supporting the desired techniques, and must be detailed enough to provide answers to the specific questions under consideration for the particular subsystems it applies to. An equally important choice has to do with the way the levels of abstraction are connected, or, in other words, with the abstraction and refinement functions that are used to relate the models. In general, many forms of abstraction and refinement are possible. In practice, only those that preserve certain properties of interest are useful. In particular, we are interested in abstractions that preserve the refinement relationship $\preceq$ when moving from a more abstract model to a more concrete one. More formally, assume p and q are agents in a model $\mathcal{Q}$, and that p' and q' are the corresponding abstractions in a model $\mathcal{Q}'$. Then we say that the abstraction preserves the refinement relationship from the abstract to the concrete model if $p' \preceq q'$ implies that also $p \preceq q$.

This property is useful for several reasons. First, refinement verification can be used to establish that an agent satisfies some requirement by comparing it to a specification. It would therefore be at best inconvenient if the result of this verification were lost during a refinement step of the methodology. In the worst case, it could lead to incorrect designs. A second advantage has to do with the efficiency of refinement verification. The process is in fact potentially more efficient at the abstract level because of the lesser amount of information included in the model. An abstraction that preserves the refinement relationship can thus be used to translate a complex verification problem at the concrete level to a simpler problem at the abstract level. This translation is

conservative: while the loss of information may make it impossible to establish a refinement relationship between the abstracted agents, it ensures that when the relationship is indeed established it also holds at the concrete level. In other words, false positive results are ruled out.

Conservative approximations are constructed using two abstraction functions, instead of just one. The first function, usually denoted Ψ_u, is applied to the implementation p, while the second function, denoted Ψ_l, is applied to the specification q. The pair (Ψ_l, Ψ_u) forms a conservative approximation whenever $\Psi_u(p) \preceq \Psi_l(q)$ implies $p \preceq q$. Thus, by definition, a conservative approximation always preserves the refinement relationship from the abstract to the concrete model. In the rest of this paper, we first introduce a number of models of interest for the development of embedded systems, and then show how to relate them using conservative approximations, and their inverses, obtained from simple homomorphisms on traces.

The notion of a conservative approximation is independent of the use of traces as an underlying agent model [18]. In particular, it could be used in other contexts, such as branching-time logics, where refinement and equivalence are expressed in terms of simulations. Our motivations for developing a trace-based model is the ease with which conservative approximations can be derived starting from simple homomorphic functions on behaviors.

3 Models of Embedded System Behavior

In this section we will present three models at progressively higher levels of abstraction, by defining a trace algebra and a corresponding agent algebra. We develop our models in the context of hybrid systems [4], a particular kind of heterogeneous systems that combine behaviors expressed as a continuous evolution with the occurrence of instantaneous discrete events. These two aspects of a behavior are often called the *flows* and the *jumps* of the system. Hybrid formalisms are particularly useful when designing embedded control systems, which require modeling the physical behavior of environments that undergo sudden mode changes. The hybrid model, in addition, is necessary for an accurate evaluation of a control strategy based on discrete computations.

The first model that we present, called *metric time*, is intended to represent exactly the evolutions (the flows and the jumps) of a system as a function of global real time. With the second we abstract away the metric while maintaining the total order of occurrence of events. This model is used to define the untimed semantics of embedded applications. Finally, the third trace algebra further abstracts away the information on the event occurrences by only retaining initial and final states and removing the intermediate steps. This simpler model can be used to describe the semantics of some programming language constructs. Later, we will use homomorphisms on trace sets to derive conservative approximations.

3.1 Metric Time

A typical semantics for hybrid embedded systems includes continuous *flows* that represent the continuous dynamics of the system, and discrete *jumps* that represent instantaneous changes of the operating conditions. The system is modeled by its state

variables. In our formalization, the evolution of the state variables takes the form of a single piece-wise continuous function over real-valued time, where the continuous segments represent the flows, while the discontinuities between the segments model the jumps. In this paper we assume that the variables of the system take only real or integer values. Real-valued variables are used, for instance, to model quantities such as position and speeds, while integer variables are more appropriate for modalities and other discrete quantities. The sets of real-valued and integer valued variables for a given trace are called $V_{\mathbb{R}}$ and $V_{\mathbb{Z}}$, respectively. Traces may also contain actions, which are discrete events that can occur at any time. Actions do not carry data values. For a given trace, the set of input actions is M_I and the set of output actions is M_O. Actions could be, for example, the commands issued by a user, or signals generated by an embedded controller.

Each trace has a signature γ which is a 4-tuple of the above sets of signals:

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

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

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

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

The set of complete traces for a signature γ is $B_C(\gamma)$. Each element of $B_C(\gamma)$ is 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 $\mathit{proj}(B)(x)$ is defined iff $M_I \subseteq B \subseteq A$, where B is the set of signals that must be *retained*. The trace that results is the same as x except that the domain of f is restricted to B. The renaming operation $x' = \mathit{rename}(r)(x)$ 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). Concatenation is defined only when the end points of the two traces match. By doing so, jumps must be

modeled explicitly by a trace, and do not arise as a byproduct of concatenation. When defined, $x_3 = (\gamma_1, \delta_3, f_3)$ is such that $\delta_3 = \delta_1 + \delta_2$ and for all $a \in A$,

$$f_3(a)(\delta) = f_1(a)(\delta) \quad \text{for } 0 \leq \delta \leq \delta_1,$$
$$f_3(a)(\delta) = f_2(a)(\delta - \delta_1) \quad \text{for } \delta_1 \leq \delta \leq \delta_3.$$

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.

3.2 Non-metric Time

In the definition of this trace algebra we are concerned with the order in which events occur in the system, but not in their absolute distance or position. This is useful if we want to describe the semantics of a programming language for embedded systems that abstracts from a particular real time implementation. Although we want to remove real time, we want to retain the global ordering on events induced by time. In particular, to simplify the abstraction from metric time to non-metric time described below, we would like to support the case of an uncountable number of events[1]. Sequences are clearly inadequate given our requirements. Instead we use a more general notion of a partially ordered multiset to represent the trace. We quote the original definition given by Pratt [19], and due to Gischer, which begins with the definition of a labeled partial order. We then specialize this notion to our needs.

Definition 1 (Labeled partial order, Pratt [19]). *A* labeled partial order *(lpo) is a 4-tuple* $L = (V, \Sigma, \leq, \mu)$ *consisting of*

1. *a* vertex set V*, typically modeling* events*;*
2. *an* alphabet Σ *(for symbol set), typically modeling* actions *such as the arrival of integer 3 at port Q, the transition of pin 13 of IC-7 to 4.5 volts, or the disappearance of the 14.3 MHz component of a signal;*
3. *a* partial order $\leq$ *on* V*, with* $e \leq f$ *typically being interpreted as event* e *necessarily preceding event* f *in time; and*
4. *a* labeling function $\mu : V \to \Sigma$ *assigning symbols to vertices, each labeled event representing an* occurrence *of the action labeling it, with the same action possibly having multiple occurrence, that is,* μ *need not be injective.*

A *pomset* (partially ordered multiset) is then the isomorphism class of an lpo, denoted $[V, \Sigma, \leq, \mu]$. By taking lpo's up to isomorphism we confer on pomsets a degree of abstraction equivalent to that enjoyed by strings (regarded as finite linearly ordered labeled sets up to isomorphism), ordinals (regarded as well-ordered sets up to isomorphism), and cardinals (regarded as sets up to isomorphism).

This representation is suitable for the above mentioned infinite behaviors: the underlying vertex set may be based on an uncountable total order that suits our needs. For our application, we do not need the full generality of pomsets. Instead, we restrict ourselves to pomsets where the partial order is total, which we call *tomsets*.

[1] In theory, such Zeno-like behavior is possible, for example, for an infinite loop whose execution time halves with every iteration.

Traces have the same form of signature as in metric time:

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

Both partial and complete traces are of the form $x = (\gamma, L)$ where L is a tomset. When describing the tomset L of a trace, we will in fact describe a particular lpo, with the understanding that L is the isomorphism class of that lpo. An action $\sigma \in \Sigma$ of the lpo is a function with domain A such that for all $v \in V_{\mathbb{R}}$, $\sigma(v)$ is a real number (the value of variable v resulting from the action σ); for all $v \in V_{\mathbb{Z}}$, $\sigma(v)$ is an integer; and for all $a \in M_I \cup M_O$, $\sigma(v)$ is 0 or 1. The underlying vertex set V, together with its total order, provides the notion of time, a space that need not contain a metric. For both partial and complete traces, there must exist a unique minimal element $min(V)$. The action $\mu(min(V))$ that labels $min(V)$ should be thought of as giving the initial state of the variables in $V_{\mathbb{R}}$ and $V_{\mathbb{Z}}$. For each partial trace, there must exist a unique maximal element $max(V)$ (which may be identical to $min(V)$). As defined above, the set of partial traces and the set of complete traces are not disjoint. It is convenient, in fact, to extend the definitions so that traces are labeled with a bit that distinguishes partial traces from complete traces, although we omit the details.

By analogy with the metric time case, it is straightforward to define projection and renaming on actions $\sigma \in \Sigma$. This definition can be easily extended to lpo's and, thereby, traces. The concatenation operation $x_3 = x_1 \cdot x_2$ is defined iff x_1 is a partial trace, $\gamma_1 = \gamma_2$ and $\mu_1(max(V_1)) = \mu_2(min(V_2))$. When defined, the vertex set V_3 of x_3 is a disjoint union:

$$V_3 = V_1 \uplus (V_2 - min(V2)),$$

ordered such that the orders of V_1 and V_2 are preserved and such that all elements of V_1 are less than all elements of V_2. The labeling function is such that for all $v \in V_3$

$$\mu_3(v) = \mu_1(v) \text{ for } min(V_1) \leq v \leq max(V_1),$$
$$\mu_3(v) = \mu_2(v) \text{ for } max(V_1) \leq v.$$

3.3 Pre-post Time

The third and last trace algebra is concerned with modeling non-interactive constructs of 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.

We cannot model communication actions at this level of abstraction, so signatures are of the form $\gamma = (V_{\mathbb{R}}, V_{\mathbb{Z}})$ and the alphabet of γ is $A = V_{\mathbb{R}} \cup V_{\mathbb{Z}}$. A non-degenerate state s is a function with domain A such that for all $v \in V_{\mathbb{R}}$, $s(v)$ is a real number (the value of variable v in state s); and for all $v \in V_{\mathbb{Z}}$, $s(v)$ is an integer. We also have a degenerate, undefined state $\perp_*$.

A partial trace $B_P(\gamma)$ is a triple (γ, s_i, s_f), where s_i and s_f are the initial and final states. A complete trace $B_C(\gamma)$ is of the form $(\gamma, s_i, \perp_\omega)$, where $\perp_\omega$ indicates non-termination. This trace algebra is primarily intended for modeling terminating behaviors, which explains why so little information is included on the traces that model non-terminating behaviors.

The operations of projection and renaming are built up from the obvious definitions of projection and renaming on states. The concatenation operation $x_3 = x_1 \cdot x_2$ is defined iff x_1 is a partial trace, $\gamma_1 = \gamma_2$ and the final state of x_1 is identical to the initial state of x_2. As expected, when defined, x_3 contains the initial state of x_1 and the final state of x_2.

3.4 Construction of Agent Models

Our models of agent are constructed in a fixed way from models of traces by considering the set of behaviors that an agent exhibits. An agent over a given trace algebra is a pair (γ, P), where γ is a signature and P is a subset of the traces for that signature. The set P represents the set of possible behaviors of an agent.

An agent algebra has a set of agents over a given trace algebra as its domain. Operations of projection, renaming, parallel composition and serial composition on agents are defined using the operations of the trace algebra, as follows.

Projection and renaming are the simplest operations to define. When they are defined depends on the signature of the agent in the same way that definedness for the corresponding trace algebra operations depends on the signature of the traces. The signature of the result is also analogous. Finally, the set of traces of the result is defined by naturally extending the trace algebra operations to sets of traces. For instance, if $p = (\gamma, P)$ is an agent, then $\mathit{proj}(B)(p) = (\gamma', \mathit{proj}(B)(P))$, where γ' is obtained by γ by retaining only the elements of B. Sequential composition is defined in terms of concatenation in an analogous way. The only difference from projection and renaming is that sequential composition requires two agents as arguments, and concatenation requires two traces as arguments.

Parallel composition of two agents is defined only when all the traces in the agents are complete traces, and the set of output actions of the two agents are disjoint. Let the agent p'' be the parallel composition of p and p'. Then the components of p'' are as follows (M_I and M_O are omitted in pre-post traces):

$$
\begin{aligned}
V''_{\mathbb{R}} &= V_{\mathbb{R}} \cup V'_{\mathbb{R}} \\
V''_{\mathbb{Z}} &= V_{\mathbb{Z}} \cup V'_{\mathbb{Z}} \\
M''_O &= M_O \cup M'_O \\
M''_I &= (M_I \cup M'_I) - M''_O \\
P'' &= \{x \in \mathcal{B}_C(\gamma'') : \mathit{proj}(A)(x) \in P \wedge \mathit{proj}(A')(x) \in P'\}.
\end{aligned}
$$

Here, the variables of the composite p'' are the union of the variables of the components p and p'. The actions of the composite are also the union of the actions of the components. An action is regarded as an output of the composite if it is an output of either component. However, an action is an input of the composite if it is an input of one of the components, and it is not at the same time an output of the other component, so that an input can only be connected to one output. The definition of P'' ensures that the behaviors of the composite are all and only the behaviors consistent with the components.

4 Relations Between Models

The three trace algebras defined above cover a wide range of levels of abstraction. The first step in formalizing the relationships between those levels is to define homomorphims between the trace algebras. Trace algebra homomorphisms induce corresponding conservative approximations between the agent algebras, as we shall see.

4.1 Homomorphisms

From metric to non-metric time. A homomorphism from metric time to non-metric time should abstract away detailed timing information. This requires characterizing events in metric time and mapping those events into a non-metric time domain. Since metric time trace algebra is, in part, value based, some additional definitions are required to characterize events at that level of abstraction.

Let x be a metric trace with signature γ and alphabet A such that

$$\gamma = (V_{\mathbb{R}}, V_{\mathbb{Z}}, M_I, M_O),$$
$$A = V_{\mathbb{R}} \cup V_{\mathbb{Z}} \cup M_I \cup M_O.$$

We define the homomorphism h by defining a non-metric time trace $y = h(x)$. This requires building a vertex set V and a labeling function μ to construct an lpo. The trace y is the isomorphism class of this lpo. For the vertex set we take all reals such that an event occurs in the trace x, where the notion of event is formalized in the next several definitions.

Definition 2 (Stable function). *Let f be a function over a real interval to $\mathbb{R}$ or $\mathbb{Z}$. The function is* stable at t *iff there exists an $\epsilon > 0$ such that f is constant on the interval $(t - \epsilon, t]$.*

Definition 3 (Stable trace). *A metric time trace x is* stable at t *iff for all $v \in V_{\mathbb{R}} \cup V_{\mathbb{Z}}$ the function $f(v)$ is stable at t; and for all $a \in M_I \cup M_O$, $f(a)(t) = 0$.*

In other words, a trace is stable at a time t if it is possible to find a left neighborhood of t (i.e., an interval $(t - \epsilon, t]$ for $\epsilon > 0$) where the trace is constant and no action occurs. When a trace is not stable at t, then we say that the trace has an event at t.

Definition 4 (Event). *A metric time trace x has an* event at $t > 0$ *if it is not stable at t. Because a metric time trace doesn't have a left neighborhood at $t = 0$, we always assume the presence of an event at the beginning of the trace. If x has an event at t, the* action label σ *for that event is a function with domain A such that for all $v \in A$, $\sigma(a) = f(a)(t)$, where f is a component of x as described in the definition of metric time traces.*

Now we construct the vertex set V and labeling function μ necessary to define y and, thereby, the homomorphism h. The vertex set V is the set of reals t such that x has an event at t. While it is convenient to make V a subset of the reals, remember that the tomset that results is an isomorphism class. Hence the metric defined on the set of reals is lost. The labeling function μ is such that for each element $t \in V$, $\mu(t)$ is the action label for the event at t in x.

Note that if we start from a partial trace in the metric trace we obtain a trace in the non-metric trace that has an initial and final event. It has an initial event by definition. It has a final event because the metric trace either has an event at δ (the function is not constant), or the function is constant at δ, and then there must be an event that brought the function to that constant value (which, in case of identically constant functions, is the initial event itself).

To show that h does indeed abstract away information, consider the following situation. Let x_1 be a metric time trace. Let x_2 be same trace where time has been "stretched" by a factor of two (i.e., for all $v \in A_1$, $x_1(a)(t) = x_2(a)(2t)$). The vertex sets generated by the above process are isomorphic (the order of the events is preserved), therefore $h(x_1) = h(x_2)$.

From non-metric to pre-post time. The homomorphism h from the non-metric time traces to pre-post traces requires that the signature of the agent be changed by removing M_I and M_O. Let $y = h(x)$. The initial state of y is formed by restricting $\mu(min(V))$ (the initial state of x) to $V_{\mathbb{R}} \cup V_{\mathbb{Z}}$. If x is a complete trace, then the final state of y is $\perp_\omega$. If x is a partial trace, and there exists $a \in M_I \cup M_O$ and time t such that $f(a)(t) = 1$, the final state of y is $\perp_*$. Otherwise, the final state of y is formed by restricting $\mu(max(V))$.

4.2 Conservative Approximations

As discussed in the introduction, we are interested in relating different models that describe systems at different levels of abstraction. We can accomplish this by deriving a conservative approximation from a homomorphism between trace algebras. 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. For instance, a non-metric time trace x' can be thought of the abstraction of all possible stretched versions y in the metric time model. This is easily extended to sets of traces, and therefore to agents. Hence, if $\mathcal{Q}$ and $\mathcal{Q}'$ are agent algebras over $\mathcal{C}$ and $\mathcal{C}'$ respectively, we use the function Ψ_u that maps an agent $p = (\gamma, P)$ in $\mathcal{Q}$ into the agent $(\gamma, h(P))$ in $\mathcal{Q}'$ 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 $\Psi_l(p)$. This leads to the definition of a function $\Psi_l(p)$ that maps P into the set $h(P) - h(\mathcal{B}(\gamma) - P)$, where $\mathcal{B}(\gamma)$ is the set of all traces with alphabet γ. For instance, the lower bound of a metric time agent p into non-metric time includes a trace x' if and only if p contains all its possible concretizations (time stretched versions). 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 [3]).

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 agent algebras $\mathcal{Q}$ over $\mathcal{C}$ and $\mathcal{Q}'$ over $\mathcal{C}'$, and a conservative approximation Ψ induced by h. Thus, one need only construct two models of individual behaviors and a homomorphism between them to obtain two agent models along with a conservative approximation between the individual agents of the models.

This same approach can be applied to the three trace algebras, and the two homomorphisms between them, that were defined in Section 3, giving conservative approximations between process models at three different levels of abstraction. The application of the upper bound is straightforward, since it is a natural extension to sets of the homomorphism on behaviors. The lower bound, on the other hand, provides complementary information. For instance, the lower bound of a metric-time agent contains all those behaviors for which the agent has an analogous behavior for any possible "stretching" of the time-axis. Thus, the lower bound identifies those behaviors in an agent that are in a sense *speed independent*. Similarly, the conservative approximation from non-metric time to pre-post traces identifies the subset of behaviors of an agent which depend exclusively on the initial and final state of the computation.

4.3 Inverse Approximations

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

To formalize what it means to be represented exactly in the context of conservative approximations, we define the inverse Ψ_{inv} of the conservative approximation Ψ. The inverse of an approximation is a function from the abstract model $\mathcal{Q}'$ to the concrete model $\mathcal{Q}$ that, as we shall see in this section, completes the relationships between $\mathcal{Q}$ and $\mathcal{Q}'$ by establishing a refinement map across the models. Normal notions of the inverse of a function are not adequate for constructing the inverse of a conservative approximation Ψ, since Ψ is a pair of functions. Our notion of an inverse is thus based on the following result.

Lemma 1. *Let $\mathcal{Q}$ and $\mathcal{Q}'$ be models of computation, and let (Ψ_l, Ψ_u) be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. For all p_1 and p_2 in $\mathcal{Q}$, if $\Psi_l(p_1) = \Psi_u(p_1) = p'$ and $\Psi_l(p_2) = \Psi_u(p_2) = p'$, then $p_1 = p_2$.*

Lemma 1 shows that when the upper and the lower bound coincide for a particular agent p, then, intuitively, the abstraction p' is an exact representation of p. To put it another way, p does not use any of the additional information provided by the concrete level, since it can be determined uniquely from its abstraction p'. It is therefore natural to define $\Psi_{inv}(p') = p$, where p is the agent in $\mathcal{Q}$ such that $\Psi_u(p) = \Psi_l(p) = p'$. If $\Psi_l(p) \neq \Psi_u(p)$, then p is not represented exactly in $\mathcal{Q}'$. In this case, p is not in the image of Ψ_{inv}.

Definition 5 (Inverse of a Conservative Approximation). *Let $\Psi = (\Psi_l, \Psi_u)$ be a conservative approximation from $\mathcal{Q}$ to $\mathcal{Q}'$. For $p' \in \mathcal{Q}'$, the inverse $\Psi_{inv}(p')$ is defined and is equal to p if and only if $\Psi_l(p) = \Psi_u(p) = p'$.*

It follows from the definition that, when Ψ_{inv} is defined, the following identity holds:

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

The function Ψ_{inv} need not be defined for all p'. This may happen, for example, if the model $\mathcal{Q}'$ includes information that cannot be expressed exactly in $\mathcal{Q}$. In that case, Ψ_{inv} is a partial function and is only defined for the agents that have an exact representation in both models. When an agent has an exact representation in $\mathcal{Q}$ and $\mathcal{Q}'$, we say that it can be used *indifferently* in the two models, or that it is *polymorphic*. This is because the agent makes no assumption regarding its behavior based on information that can be expressed exclusively in either model of computation. However, the representation of the agent in $\mathcal{Q}$ and $\mathcal{Q}'$ is, in general, different. Thus, this notion extends our ability to reuse agents across models that employ different representations.

For our examples, the inverse of the approximation from metric to non-metric time agent is always defined, and translates a non-metric time agent to a corresponding metric time agent which non-deterministically chooses a given timing for any of its behaviors. This non-determinism is typical of our approach, and is useful to expose the degrees of freedom that are available in a design-by-refinement methodology. Similarly, the inverse of the conservative approximation that goes from pre-post to non-metric time agents builds a concretization where each pair of initial and final states is non-deterministically computed by reordering actions along the time axis.

A conservative approximation thus induces its own inverse in the form of a (possibly partial) refinement map. The inverse is uniquely determined, and, because of the defining properties of a conservative approximation, Ψ_{inv} is one-to-one, and, when restricted to the image of Ψ_{inv}, the functions Ψ_l and Ψ_u are equal and are the inverse of Ψ_{inv}. In addition, when defined, Ψ_{inv} is always monotonic and, if either Ψ_l or Ψ_u is also monotonic, it preserves the ordering of the agents in both directions. Hence, the inverse embeds the abstract model of computation (or at least the part of it where it is defined) into the more concrete model, in a way that is consistent with the chosen abstractions. Different conservative approximations between the same models may therefore induce different embeddings. This is again an indication of the importance of choosing the right abstraction for the problem at hand. The nature of the embedding, in fact, determines how one model is interpreted in terms of the other, and quantifies the amount of information lost during the abstraction.

4.4 Modeling Constructs in Embedded Software

Using Pre-Post Traces. One of the fundamental features of embedded software is that it interacts with the physical world. Conventional axiomatic or denotational semantics of sequential programming languages only model initial and final states of terminating programs. Thus, these semantics are inadequate to fully model embedded software. However, much of the code in an embedded application does computation or internal communication, rather than interacting with the physical world. Such code can be adequately modeled using conventional semantics, as long as the model can be integrated with the more detailed semantics necessary for modeling interaction. Pre-post agents are quite similar to conventional semantics. As described earlier, we can also embed pre-post agents into more detailed models. Thus, we can model the non-interactive parts of an embedded application at a high level of abstraction that is simpler and more natural, while also being able to integrate accurate models of interaction, real-time constraints and continuous dynamics.

As an example we consider the problem of developing software to control an engine in the cutoff region [2]. Here, the behaviors of an automobile engine are divided into regions of operation, each characterized by appropriate control actions to achieve a desired result. The cutoff region is entered when the driver releases the accelerator pedal, thereby requesting that no torque be generated by the engine. In order to minimize power train oscillations that result from suddenly reducing torque, a closed loop control damps the oscillations using carefully timed injections of fuel. The control problem is therefore hybrid, consisting of a discrete (the fuel injection) and a continuous (the power train behavior) systems tightly linked.

```
01. void control_algorithm( void ) {
02.     // state definition
03.     struct state { double x1; double x2; double omega_c; } current_state;
04.     // Init the past three injections (assume injection before cutoff)
05.     double u1, u2, u3 = 1.0;
06.
07.     loop forever {
08.         await( action_request );
09.         read_current_state( current_state );
10.         compute_sigmas( sigma_m, sigma_0, current_state, u1, u2, u3 );
11.         // update past injections
12.         u1 = u2; u2 = u3;
14.         // compute next injection signal
15.         if ( sigma_m < sigma_0 ) {
16.             action_injection( );
17.             u3 = 1.0;
18.         } else {
19.             action_no_injection( );
20.             u3 = 0.0;
21.         }
22.     }
23. }
```

Fig. 1. An embedded control algorithm

Figure 1 shows the top level routine of the control algorithm. Although we use a C-like syntax, the semantics are simplified, as described later. The controller is activated by a request for an injection decision (this happens every full engine cycle). The algorithm first reads the current state of the system (as provided by the sensors on the power train), predicts the effect of injecting or not injecting on the future behavior of the system, and finally controls whether injection occurs. The prediction uses the value of the past three decisions to estimate the position of the future state. The control algorithm involves solving a differential equation, which is done in the call to `compute_sigmas` (see [2] for more details). A nearly optimal solution can be achieved without injecting intermediate amounts of fuel (i.e., either inject no fuel or inject the maximum amount). Thus, the only control inputs to the system are the actions `action_injection` (maximum injection) and `action_no_injection` (zero injection).

The semantics of each statement of the programming language is given by an agent. To simplify the semantics, we assume that inter-process communication is done through shared actions rather than shared variables. A pre-post agent has a signature γ of the form $(V_{\mathbb{R}}, V_{\mathbb{Z}})$. For the semantics of a programming language statement, γ indicates the variables accessible in the scope where the statement appears. For a block that declares local variables, the agent for the statement in the block includes in its signature the local variables. The agent for the block is formed by projecting away the local variables from the agent of the statement.

The sequential composition of two statements is defined as the concatenation of the corresponding agents: the definition of concatenation ensures that the two statements agree on the intermediate state. The traces in the agent for an assignment to variable v are of the form (γ, 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 an agent with an alphabet $\{v_1, \ldots, v_r\}$ where v_k is the k-th argument of the procedure (these signal names do not necessarily correspond to the names of the formal variables). We omit the details of how this agent is constructed from the text of the procedure definition. More relevant for our control algorithm example, the semantics of a procedure call `proc(a, b)` is the result of renaming $v_1 \rightarrow a$ and $v_2 \rightarrow b$ on the agent 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` and `while` loops we define a function $\mathit{init}(x, c)$ to be 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. In particular, for pre-post traces, $\mathit{init}(x, c)$ 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

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

Notice that this definition can be used for any trace algebra where $\mathit{init}(x, c)$ has been defined, and that it ignores any effects of the evaluation of c not being atomic.

In the case of `while` loops we first define a set of traces E such that for all $x \in E$ and traces y, if $x \cdot y$ is defined then $x \cdot y = y$. For pre-post traces, E is the set of all traces with identical initial and final states. If c is the condition of the loop, and P_B the set of possible traces of the body, we define $P_{T,k}$ and $P_{N,k}$ to be the set of terminating and non-terminating traces, respectively, for iteration k, as follows:

$$\begin{aligned} P_{T,0} &= \{x \in E : \neg \mathit{init}(x, c)\} \\ P_{N,0} &= \{x \in E : \mathit{init}(x, c)\} \\ P_{T,k+1} &= P_{N,k} \cdot P_B \cdot P_{T,0} \\ P_{N,k+1} &= P_{N,k} \cdot P_B \cdot P_{N,0} \end{aligned}$$

The concatenation of $P_{T,0}$ and $P_{N,0}$ at the end of the definition ensures that the final state of a terminating trace does not satisfy the condition c, while that of a non-terminating trace does. Clearly the semantics of the loop should include all the terminating traces. For non-terminating traces, we need to introduce some additional notation. A sequence $Z =< z_0, \ldots >$ is a non-terminating execution sequence of a loop if, for all k, $z_k \in P_{N,k}$ and $z_{k+1} \in z_k \cdot P_B$. This sequence is a chain in the prefix ordering. The initial state of Z is defined to be the initial state of z_0. For pre-post traces, we define $P_{N,\omega}$ to be all traces of the form $(\gamma, s, \perp_\omega)$ where s is the initial state of some non-terminating execution sequence Z of the loop. The set of possible traces of the loop is therefore

$$P = (\bigcup_k P_{T,k}) \cup P_{N,\omega}.$$

Using Non-Metric Time Traces. Using an inverse conservative approximation, as described earlier, the pre-post trace semantics outlined above can be embedded into the non-metric time agent model. However, this is not adequate for two of the constructs used in Figure 1: `await` and the non-terminating loop. These constructs must be describe directly at the lower level of abstraction provided by non-metric time traces.

As used used in Figure 1, the `await(a)` simply delays until the external action `a` occurs. Thus, the possible partial traces of `await` are those where the values of the state variables remain unchanged and the action `a` occurs exactly once, at the endpoint of the trace. The possible complete traces are similar, except that the action `a` must never occur.

To give a more detailed semantics for non-terminating loops, we define the set of extensions of a non-terminating execution sequence Z to be the set $ext(Z) = \{x \in \mathcal{B}(\gamma) : \forall k[z_k \in pref(x)]\}$. For any non-terminating sequence Z, we require that $ext(Z)$ be non-empty, and have a unique maximal lower bound contained in $ext(Z)$, which we denote $\lim(Z)$. In the above definition of the possible traces of a loop, we modify the definition of the set of non-terminating behaviors $P_{N,\omega}$ to be the set of $\lim(Z)$ for all non-terminating execution sequences Z.

Using Metric Time Traces. Analogous to the embedding discussed in the previous subsection, non-metric time agents can be embedded into the metric-time agent model. Here continuous dynamics can be represented, as well as timing assumptions about programming language statements. Also, timing constraints that a system must satisfy can be represented, so that the system can be verified against those constraints.

5 Related Work

Abstract interpretations [6,7] are a widely used means of relating different domains of computation for the purpose of facilitating the analysis of a system. An abstract interpretation between two domains of computation consists of an abstraction function and of a concretization function that form a Galois connection. The distinguishing feature of an abstract interpretation is that the concretization of the evaluation of an expression using the operators of the abstract domain of computation is guaranteed to be an

upper bound of the corresponding evaluation of the same expression using the operators of the concrete domain. Hence, a conservative evaluation can be carried out at the more abstract level, where it is potentially computationally more efficient. Refinement verification, however, is unsound: a positive refinement verification result at the abstract level does not guarantee a corresponding refinement verification result at the concrete level. Conservative approximations overcome this problem because they employ two separate abstraction functions, one for the implementation and one for the specification. Our study shows that this is a necessary condition for the preservation of refinement, and one that is not satisfied by a Galois connection [18]. Conservative approximations and abstract interpretations are however strongly related, in that a pair of Galois connections can be used to construct a conservative approximation [18]. This result is important because it extends the rich field of abstract interpretations to refinement verification.

The study of heterogeneous systems is also a central theme of both the Metropolis [1] and the Ptolemy [11] projects. In Metropolis, a system is composed of processes that communicate over media expressed in a meta-model of computation. Their combination, and their relationships, implicitly determine the interaction semantics. Because of its generality, the underlying meta-model fabrics can be used to promote reuse of diverse components. The communication media, however, must be carefully designed to resolve possible incompatibilities. Our work can be thought of as the theory base for the use of the meta-model to represent heterogeneous systems. In addition, conservative approximations have been used to make the process of platform-based design advocated in the Metropolis project precise, and their application in this area is part of our current work [17].

Similarly, Ptolemy consists of several executable domains of computation that can be mixed in a hierarchy controlled by a global scheduler. Ptolemy does not currently provide a notion of abstraction between the different models in the system. However, an important innovative concept in the design of the Ptolemy II infrastructure is the notion of *domain polymorphism* [12]. An actor (agent) is domain polymorphic if it can be used indifferently, i.e., without modification, in several domains of computation. To check whether an actor can be used in a particular domain, the authors set up a type system based on state machines, which is used to describe the assumptions of each model and each actor relative to an abstract semantics.

Conservative approximations offer a formal way of defining a similar concept of polymorphism, even though they do not rely upon a common underlying semantics, as in the case of Ptolemy. A distinctive feature of conservative approximations is their ability to determine which parts of the models are unaffected by the application of the abstraction. This information is useful because it identifies the elements of the models that can be expressed indifferently under the interpretation of either model, without changing their meaning. Our interpretation of this notion is, however, broader than that introduced in Ptolemy II. In particular, an actor (agent) is polymorphic in our framework when it makes no assumption regarding its behavior based on information that cannot be expressed in the other model. When this is the case, reuse of subsystems can be extended across the boundaries of heterogeneous models. This leads to the notion of the *inverse* of a conservative approximation, which is a refinement map that is used

to embed one model into another. The embedding provides us with an interpretation of agents across different models which is consistent with the corresponding abstraction. *An agent is polymorphic precisely when this interpretation is exact*. This has the advantage of making the process of abstraction and refinement of an agent explicit. Elements that do not fall in the range of the inverse can only be approximated by the other model.

Another example of approximation is the *homomorphic reduction* proposed by Kurshan [9,10]. This technique can be applied to models of behavior that consist of languages (sets of sequences) that are recognized by a class of ω automata called L-automata, which are able to express both safety and fairness constraints. Here, each automaton P constructed over a set of symbols L (an L-automaton) accepts a language $\mathcal{L}(P) \subseteq L^\omega$, where L^ω denotes the set of all infinite sequences of symbols from L. Verification in this context is the process of determining whether the language $\mathcal{L}(P)$ recognized by an implementation automaton P is contained in the language $\mathcal{L}(T)$ accepted by the specification automaton T, i.e., that $\mathcal{L}(P) \subseteq \mathcal{L}(T)$. This problem can be reduced to a more abstract language L' by verifying that $\mathcal{L}(P') \subseteq \mathcal{L}(T')$, for appropriate abstract L'-automata P' and T'. The main result[2] states that $\mathcal{L}(P') \subseteq \mathcal{L}(T')$ implies $\mathcal{L}(P) \subseteq \mathcal{L}(T)$ provided there exists a *language homomorphism* $\Phi : L^\omega \mapsto L'^\omega$ such that $\Phi(\mathcal{L}(P)) \subseteq \mathcal{L}(P')$ and $\Phi(\mathcal{L}(T^\#)) \subseteq \mathcal{L}(T'^\#)$. In this case, Φ is said to be *co-linear*[3] for $(P, T; P', T')$. In the co-linearity condition above, the notation $T^\#$ denotes the *dual* automaton[4] of T, which is closely related to language complementation.

We have argued before that one function on languages is not sufficient to guarantee the preservation of such verification result. The apparent contradiction with the use of just one language homomorphism Φ can be reconciled by accounting for the use of the dual automaton in the co-linearity condition. Effectively, if Φ is co-linear for $(P, T; P', T')$, then it can be shown that not only is $\Phi(\mathcal{L}(P)) \subseteq \mathcal{L}(P')$, but also that $\mathcal{L}(T') \subseteq \overline{\Phi(\overline{\mathcal{L}(T)})}$, where the overline bar denotes language complementation. Hence, the language of the specification T is transformed according to a different abstraction functions, namely $\Theta(\mathcal{L}(T)) = \overline{\Phi(\overline{\mathcal{L}(T)})}$. Interestingly, Φ and Θ form the upper and lower bound of a conservative approximation that is closely related (and under certain conditions equal) to the conservative approximation induced by a homomorphism (see Section 4). Co-linearity of Φ thus simply ensures that $\mathcal{L}(P')$ and $\mathcal{L}(T')$ provide looser bounds, a condition that still guarantees soundness in the verification. Conservative approximations generalize the technique of homomorphic reduction to arbitrary agent models, and can therefore be applied to models that are not described by automata.

Model checking techniques based on abstraction/refinement is also a well studied related field of application for abstraction mappings [5], and is a typical application of the framework of abstract interpretations. The technique consists of first deriving an over-approximation of a state-based model using, for instance, predicate abstraction. The abstract model is constructed in a way that ensures that the property to be verified can be represented exactly (by, for example, an appropriate choice of the predicates). Therefore, if the property is verified in the abstract domain, it is also verified in the

[2] Theorem 8.5.2 in [9].
[3] Definition 8.5.1 in [9].
[4] Definitions 6.2.19 and 6.2.26 in [9].

concrete domain. If not, a counter-example is generated and used to refine the abstract domain until the satisfaction of the property is determined. The approach based on conservative approximations differs because, as explained, it allows non-trivial abstraction of the specification, as well as of the implementation. Model checking techniques also exist that use under-approximations, rather than over-approximations, to derive an abstracted model [16]. This is similar to our use of the lower bound function. However, unlike our use of the lower bound, the under-approximation is applied to the implementation, rather than to the specification. This corresponds again to using a Galois connection, one that goes in the reverse direction. By doing this, if the abstract model violates the property under verification, then it can be concluded that also the concrete model violates the property. Instead, if the abstract model satisfies the property, the verification is inconclusive and the abstract model must be refined until the property is proved incorrect, or the abstraction becomes exact. This approach may be useful when the interest lies in finding true counter-examples and bug traces.

Another formalization of abstraction is based on *theory interpretations* [14]. Here, an abstract architecture description and a concrete architecture description are both translated to theories in a logical language (typically first-order logic). The concrete architecture is correct relative to the abstract architecture if there is a theory interpretation I from the abstract theory Θ to the concrete theory Θ'; that is, for every formula F, $F \in \Theta \Rightarrow I(F) \in \Theta'$. In addition, it may be required that I be faithful: $F \notin \Theta \Rightarrow I(F) \notin \Theta'$. Our approach does not interpret architectures, or other agents, as logical theories. Instead, they are directly modeled as mathematical objects. This can be thought of as a model based approach, as opposed to a theory based approach. In a model based approach, within a given model of computation, the refinement relation is just a binary relation on objects in the model. This notion of refinement is easier to reason about than theory interpretations, but it is less flexible for comparing agents in different models of computation. This can be addressed by introducing abstract interpretations or conservative approximations.

Process Spaces [15] is a very general class of concurrency models, and it compares quite closely to trace-based agent models [17]. Given a set of executions $\mathcal{E}$, a Process Space $\mathcal{S}_{\mathcal{E}}$ consists of the set of all the processes (X, Y), where X and Y are subsets of $\mathcal{E}$ such that $X \cup Y = \mathcal{E}$. The sets of executions X and Y of a process are not necessarily disjoint, and they represent the assumptions (Y) and the guarantees (X) of the process with respect to its environment. As in trace-based agent models, executions are abstract objects. Different sets of abstract executions $\mathcal{E}_1$ and $\mathcal{E}_2$ induce different Process Spaces $\mathcal{S}_{\mathcal{E}_1}$ and $\mathcal{S}_{\mathcal{E}_2}$. The notion of process abstraction from $\mathcal{S}_{\mathcal{E}_1}$ to $\mathcal{S}_{\mathcal{E}_2}$ in Process Spaces is related to the notion of conservative approximation. In particular, process abstractions are defined as the Galois connections between process spaces that are derived from a relation on the set of abstract executions. The connections are obtained as axialities [8]. A process abstraction is classified as optimistic or pessimistic according to whether it preserves certain verification results from the concrete to the abstract or from the abstract to the concrete model. These two kinds of abstraction can be used in combination to preserve verification results both ways. However, in that case, the two models are isomorphic since there is effectively no loss information. Optimistic and pessimistic process abstractions roughly correspond to the two abstraction functions of conservative

approximations. However, our use of these functions is significantly different, since we apply them in combination (one for the specification, the other for the implementation). Consequently, our models need not be isomorphic, so that we obtain stronger preservation results without sacrificing the abstraction.

Winskel et al. [20] propose a framework based on category theory that is related to ours. In their formalism, each model of computation is turned into a category where the objects are the agents, and the morphisms represent a refinement relationship based on *simulations* between the agents. The authors study a variety of different models that are obtained by selecting arbitrary combinations of three parameters: behavior vs. system (e.g., traces vs. state machines), interleaving vs. non-interleaving (e.g., state machines vs. event structures) and linear vs. branching time. The common operations in a model are derived as universal constructions in the category. Relationships can be constructed by relating the categories corresponding to different models by means of functors, which are homomorphisms of categories that preserve morphisms and their compositions. When categories represent models of computation, functors establish connections between the models in a way similar to abstraction maps and semantic functions. In particular, when the morphisms in the category are interpreted as refinement, functors become essentially monotonic functions between the models, since preserving morphisms is equivalent to preserving the refinement relationship.

In [20], the authors thoroughly study the relationships between the eight different models of concurrency above by relating the corresponding categories through functors. In addition, these functors are shown to be components of *reflections* or *co-reflections*. These are particular kinds of adjoints, which are pairs of functors that go in opposite directions and enjoy properties that are similar to the order preservation of the abstraction and concretization maps of a Galois connection. When the morphisms are interpreted as refinement, reflections and co-reflections generalize the concept of Galois connection to preorders. In fact, the relationships between categories based on adjoints are similar in nature to the abstractions and refinements obtained by abstract interpretations and conservative approximations. However, as described above for abstract interpretations, conservative approximations use independent abstractions for the implementation and the specification in order to derive a stronger result in terms of preservation of the refinement relation, and avoidance of false positive verification results. Indeed, we require two Galois connections, instead of one, to determine a single conservative approximation. In the work presented in [20], this translates in two adjoints per pair of categories.

6 Conclusions

We presented the use of abstraction and refinement functions between models of computation for the verification and design of heterogeneous systems. We compared conservative approximations to abstract interpretations and we showed that, unlike abstract interpretations, conservative approximations always preserve refinement verification results from an abstract to a concrete model, while avoiding false positives. Therefore, conservative approximations are better suited for heterogeneous design methodologies, i.e., methodologies that use several models of computation. In particular, because they always guarantee correctness, conservative approximations provide more flexibility in

choosing the verification strategy and the hierarchy of models used in the design flow. We then described how to construct models of computation suitable for the design of embedded systems, and how conservative approximations can be derived for these models starting from simple functions (homomorphisms) over their set of behaviors. In addition, the inverse of a conservative approximation has been shown to identify components that can be used indifferently in several models, thus enabling reuse across domains of computation. The resulting theory can be used as the basis of frameworks that support heterogeneous modeling.

Our current work focuses on extending techniques that make it easier to construct conservative approximations between agent models. The axialities of homomorphisms on behaviors described in this paper is one such example. However, homomorphisms are usually defined to preserve the alphabet of behaviors, so that the induced conservative approximations, too, must preserve the alphabet of agents. More interesting conservative approximations can be constructed by letting the homomorphism change the alphabet of a behavior, for example by hiding certain signals, like clocks and activation signals, that have no meaning in a more abstract model. This is also appropriate for converting a detailed protocol specification into a more abstract, transaction-based, specification. Arbitrary changes of the alphabet are also possible. In this case, however, the homomorphism must not only be applied to the behaviors, but also to the operators, in order to correctly translate expressions. In this case the homomorphism becomes similar to a functor between categories, where a category has behaviors as objects and the operators as morphisms.

A model that uses behaviors as its underlying structure may impose restrictions on the kind of agents that can be constructed. For example, only receptive (or progressive, or input enabled) agents might be allowed. The axialities of a homomorphism, however, may not necessarily yield agents that satisfy such conditions. A promising avenue of future research consists therefore in identifying the agent that most faithfully approximates the missing abstraction, while satisfying the constraints imposed by the model, and while still functioning as the bound of a conservative approximation. This would constitute a generalization of the technique proposed by Loiseaux et al. [13] on property-preserving abstractions in the context of transition systems.

References

1. Balarin, F., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A., Watanabe, Y., Yang, G.: Concurrent execution semantics and sequential simulation algorithms for the metropolis meta-model. In: Proceedings of the Tenth International Symposium on Hardware/Software Codesign, Estes Park, CO (May 2002)
2. Balluchi, A., Benedetto, M.D., Pinello, C., Rossi, C., Sangiovanni-Vincentelli, A.: Cut-off in engine control: a hybrid system approach. In: IEEE Conf. on Decision and Control (1997)
3. Burch, J.R.: Trace Algebra for Automatic Verification of Real-Time Concurrent Systems. PhD thesis, School of Computer Science, Carnegie Mellon University (Aug 1992)
4. Carloni, L.P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: Languages and Tools for Hybrid Systems Design. Foundations and Trends in Electronic Design Automation, vol. 1. Now Publishers (2006)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking, 2nd edn. The MIT Press, Cambridge (1999)

6. Cousot, P., Cousot, R.: 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, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)
7. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
8. Erné, M., Koslowski, J., Melton, A., Strecker, G.E.: A primer on galois connections. In: The Design of an Extendible Graph Editor. Ann. New Yosk Acad. Sci, vol. 704, pp. 103–125 (1993)
9. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1995)
10. Kurshan, R.P., McMillan, K.L.: Analysis of digital circuits through symbolic reduction. IEEE Trans. Comput.-Aided Design Integrated Circuits 10(11), 1356–1371 (1991)
11. Lee, E.A.: Overview of the Ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley (July 2003)
12. Lee, E.A., Xiong, Y.: System-level types for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, Springer, Heidelberg (2001)
13. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design 6, 1–35 (1995)
14. Moriconi, M., Qian, X., Riemenschneider, R.A.: Correct architecture refinement. IEEE Transactions on Software Engineering 21(4), 356–372 (1995)
15. Negulescu, R.: Process Spaces and the Formal Verification of Asynchronous Circuits. PhD thesis, University of Waterloo, Canada (1998)
16. Pasareanu, C., Pelánek, R., Visser, W.: Concrete model checking with abstract matching and refinement. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
17. Passerone, R.: Semantic Foundations for Heterogeneous Systems. PhD thesis, Department of EECS, University of California at Berkeley (May 2004)
18. Passerone, R., Burch, J.R., Sangiovanni-Vincentelli, A.L.: Refinement preserving approximations for the design and verification of heterogeneous systems. Formal Methods in System Design 31(1), 1–33 (2007)
19. Pratt, V.R.: Modelling concurrency with partial orders. International Journal of Parallel Programming 15(1), 33–71 (1986)
20. Sassone, V., Nielsen, M., Winskel, G.: Models for concurrency: Towards a classification. Theoretical Computer Science 170, 297–348 (1996)