Using Multiple Levels of Abstractions in Embedded Software Design

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

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

Abstract. The methodologies that are in use today for software development rely on representations and techniques appropriate for the applications (compilers, business applications, CAD, etc.) that have been traditionally implemented on programmable processors. Embedded software is different: by virtue of being embedded in a surrounding system, the software must be able to continuously react to stimula in the desired way. Verifying the correctness of the system requires that the model of the software be transformed to include (refine) or exclude (abstract) information to retain only what is relevant to the task at hand. In this paper, we outline a framework that we inted to use for studying the problems of abstraction and refinement in the context of embedded software for hybrid systems.

1 Introduction

Embedded Software (ESW) design is one, albeit critical, aspect of the more general problem of Embedded System Design (ESD or just ES). ESD is about the implementation of a set of functionalities satisfying a number of constraints ranging from performance to cost, emissions, power consumption and weight. The choice of implementation architecture implies which functionality will be implemented as a hardware component or as software running on a programmable component. In recent years, the functionalities to be implemented in ES have grown in number and complexity so much so that the development time is increasingly difficult to predict and control. The complexity increase coupled with the constantly evolving specifications has forced designers to look at implementations that are intrinsically flexible, i.e., that can be changed rapidly. Since hardware-manufacturing cycles do take time and are expensive, the interest in software-based implementation has risen to previously unseen levels. The increase in computational power of processors and the corresponding decrease in size and cost have allowed moving more and more functionality to software. However, this move corresponds to increasing problems in verifying design correctness, a critical aspect of ESD since several application domains, such as transportation and environment monitoring, are characterized by safety considerations that are certainly not necessary for the traditional PC-like software applications. In addition to this aspect, little attention has been traditionally

paid to hard constraints on reaction speed, memory footprint and power consumption of software. This is of course crucial for ES. These considerations point to the fact that ESW is really an implementation choice for a functionality that can be indifferently implemented as a hardware component, and that we cannot abstract away hard characteristics of software as we have done in the traditional software domain. No wonder then that we are witnessing a crisis in the ES domain for ESW design. This crisis is not likely to be resolved going about business as usual but we need to focus at the root of the problems.

Our vision for ESW is to change radically the way in which ESW is developed today by: 1) linking ESW upwards in the abstraction layers to system functionality; 2) linking ESW to the programmable platforms that support it thus providing the much needed means to verify whether the constraints posed on ES are met.

ESW today is written using low level programming languages such as C or even Assembler to cope with the tight constraints on performance and cost typical of most embedded systems. The tools available for creating and debugging software are no different than the ones used for standard software: compilers, assemblers and cross-compilers. If any difference can be found, it is that most tools for ESW are rather primitive when compared to analogous tools for richer platforms. On the other hand, ESW needs hardware support for debugging and performance evaluation that in general is not a big issue for traditional software. In most embedded software, operating systems were application dependent and developed in house. Once more, performance and memory requirements forced this approach.

When embedded software was simple, there was hardly need for a more sophisticated approach. However, with the increased complexity of ES application, this rather primitive approach has become the bottleneck and most system companies have decided to enhance their software design methodology to increase productivity and product quality. However, we do not believe that the real reasons for such a sorry state are well understood. We have seen a flurry of activities towards the adoption of object-oriented approaches and other syntactically driven methods that have certainly value in cleaning the structure and the documentation of embedded software but have barely scratched the surface in terms of quality assurance and time-to-market. Along this line, we also saw a growing interest towards standardization of Real-Time Operating Systems either de facto, see for example the market penetration of WindRiver in this domain, or through standard bodies such as the OSEK committee established by the German automotive industry. Quite small still is the market for development tools even though we do believe this is indeed the place where much productivity gain can be had. There is an interesting development in some application areas, where the need to capture system specifications at higher levels of abstraction is forcing designers to use tools that emphasize mathematical descriptions, making software code an output of the tool rather than an input. The leader in this market is certainly the Matlab tool set developed by MathWorks. In this case, designers develop their concept in this friendly environment where they can assemble their designs quickly and simulate their behaviour. While this approach

is definitely along the right direction, we have to raise the flag and say that the mathematical models supported by Matlab and more pertinently by Simulink, the associated simulator, are somewhat limited and do not cover the full spectrum of embedded system design. The lack of data flow support is critical. The lack of integration between the FSM capture tool (State Flow) and Simulink is also a problem. As we will detail later, this area is of key interest for our vision. It is at this level that we are going to have the best results in terms of functional correctness and error free refinement to implementation. The understanding of the mathematical properties of the embedded system functionality must be a major emphasis of our approach.

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

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

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

2. **Solid theoretical foundations that provide the necessary infrastructure for a new generation of tools**: The tools used in Metropolis will be interoperable and will work at different levels of abstraction, they will verify, simulate, and map designs from one level of abstraction to the next, help choose implementations that meet constraints and optimize the criteria listed above. The theoretical framework is necessary to make our claims of correctness and efficiency true. Metropolis will deal with both embedded software and hardware designs since it will intercept the design specification at a higher level of abstraction. The design specifications will have precise semantics. The semantics is essential to be able to: (i) reason about designs, (ii) identify and correct functional errors, (iv) initiate synthesis processes.

Several formal models have been proposed over the years (see e.g. [6]) to capture one or more aspects of computation as needed in embedded system design. We have been able to compare the most important models of computations using a unifying theoretical framework introduced recently by Lee and Sangiovanni-Vincentelli [9]. However, this denotational framework has only helped us to identify the sources of difficulties in combining different models of computation that are certainly needed when complex systems are being designed. In this case, the partition of the functionality of the design into different models of computation

is somewhat arbitrary as well as arbitrary are the communication mechanisms used to connect the "ports" of the different models. We believe that it is possible to optimize across model-of-computation boundaries to improve performance and reduce errors in the design at an early stage in the process.

There are many different views on how to accomplish this. There are two essential approaches: one is to develop encapsulation techniques for each pair of models that allow different models of computation to interact in a meaningful way, i.e., data produced by one object are presented to the other in a consistent way so that the object "understands" [4, 5]. The other is to develop an encompassing framework where all the models of importance "reside" so that their combination, re-partition and communication happens in the same generic framework and as such may be better understood and optimized. While we realize that today heterogeneous models of computation are a necessity, we believe that the second approach is possible and will provide designers 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 their design.

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

This framework is a work in progress. Our earlier work [2, 3] does not provide a sufficiently general notion of sequential composition, which is essential for modeling embedded software. The three models of computation described in this paper (which all include sequential composition) are examples of the kinds of models that we want to have fit into the framework. After the framework has been thoroughly tested on a large number of different models of computation, we plan to publish a complete description of the framework.

2 Overview

This section presents the basic framework we use to construct semantic domains, which is based on trace algebras and trace structure algebras. The concepts briefly described here will be illustrated by examples later in the paper. This overview is intended to highlight the relationships between the concepts that will be formally defined later.

More details of these algebras can be found in our earlier work [2, 3]. Note, however, that our definitions of these algebras do not include sequential composition. For this reason, and other more technical reasons, the models of computation used in this paper do not fit into our earlier framework.

We maintain a clear distinction between models of processes (a.k.a. agents) and models of individual executions (a.k.a. behaviors). In different models of computation, individual executions can be modeled by very different kinds of mathematical objects. We always call these objects *traces*. A model of a process, 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 processes 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. A given object can be both a complete trace and a partial trace; what is being represented in a given case is determined from context. For example, a finite string can represent a complete behavior with a finite number of actions, or it can represent a partial behavior.

In our framework, the first step in defining a model of computation is to construct a trace algebra. 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. Concatenation can be used to define the notion of a prefix of a trace. We say that a trace x is a prefix of a trace z if there exists a trace y such that z is equal to x concatenated with y.

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 *conservative approximation* is a kind of mapping from one trace structure algebra to another. It can be used to do abstraction, and it maintains a precise relationship between verification results in the two trace structure algebras. The two trace structure algebras do not have to be based on the same trace alge-

bra. Thus, conservative approximations are a bridge between different models of computation. Conservative approximations have inverses, which can be used to embed an abstract model of computation into a more detailed one. Conservative approximations can be constructed from homomorphisms between trace algebras.

3 Trace algebras for embedded software

In this section we will present the definition of three trace algebras at progressively higher levels of abstraction. The first trace algebra, called *metric time*, is intended to model exactly the evolutions (the flows and the jumps) of a hybrid system as a function of global real time. With the second trace algebra we abstract away the metric while maintaining the total order of occurence of events. This model is used to define the untimed semantics of embedded software. Finally, the third trace algebra further abstracts away the information on the event occurences 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. The next section will then present the definition of the homomorphisms that we use to approximate a more detailed trace algebra with the more abstract ones.

3.1 Metric Time

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

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

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

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

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

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

The set of partial traces for a signature γ is $B_P(\gamma)$. Each element of $B_P(\gamma)$ is as a triple $x = (\gamma, \delta, f)$. The non-negative real number δ is the *duration* (in time) of the partial trace. The function f has domain A. For $v \in V_{\mathcal{R}}$, $f(v)$ is a

function in $[0,\delta] \to \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers and the closed interval $[0,\delta]$ is the set of real numbers between 0 and δ, inclusive. This function must be piece-wise continuous and right-hand limits must exist at all points. Analogously, for $v \in V_{\mathcal{N}}$, $f(v)$ is a piece-wise constant function in $[0,\delta] \to \mathcal{N}$, where $\mathcal{N}$ is the set of integers. For $a \in M_I \cup M_O$, $f(a)$ is a function in $[0,\delta] \to \{0,1\}$, where $f(a)(t) = 1$ iff action a occurs at time t in the trace.

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

To complete the definition of this trace algebra, we must define the operations of projection, renaming and concatenation on traces. The projection operation $proj(B)(x)$ is defined iff $M_I \subseteq B \subseteq A$. The trace that results is the same as x except that the domain of f is restricted to B. The renaming operation $x' = rename(r)(x)$ 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 $x_3 = x_1 \cdot x_2$, wherer x_1 is a partial trace and x_2 is either a partial or a complete trace, is more complicated. If x_2 is a partial trace, then x_3 is defined iff $\gamma_1 = \gamma_2$ and for all $a \in A$,

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

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

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

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

3.2 Non-metric Time

In the definition of this trace algebra we are concerned with the order in which events occur in the system, but not in their absolute distance or position. This is useful if we want to describe the semantics of a programming language for hybrid systems that abstracts from a particular real time implementation.

Although we want to remove real time, we want to retain the global ordering on events induced by time. In particular, in order to simplify the abstraction from metric time to non-metric time described below, we would like to support the case of an uncountable number of events[1]. Sequences are clearly inadequate given our requirements. Instead we use a more general notion of a partially

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

ordered multiset to represent the trace. We repeat the definition found in [12], and due to Gischer, which begins with the definition of a labeled partial order.

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

1. *a* vertex set V*, typically modeling* events*;*
2. *an* alphabet Σ *(for symbol set), typically modeling* actions *such as the arrival of integer 3 at port Q, the transition of pin 13 of IC-7 to 4.5 volts, or the disappearance of the 14.3 MHz component of a signal;*
3. *a* partial order $\leq$ *on* V*, with* $e \leq f$ *typically being interpreted as event* e *necessarily preceding event* f *in time; and*
4. *a* labeling funciton $\mu : V \to \Sigma$ *assigning symbols to vertices, each labeled event representing an* occurence *of the action labeling it, with the same action possibly having multiple occurence, 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 abstractness equivalent to that enjoyed by strings (regarded as finite linearly ordered labeled sets up to isomorphism), ordinals (regarded as well-ordered sets up to isomorphism), and cardinals (regarded as sets up to isomorphism).

This representation is suitable for the above mentioned infinte 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*.

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

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

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 $v \in V_{\mathcal{R}}$, $\sigma(v)$ is a real number (the value of variable v resulting from the action σ); for all $v \in V_{\mathcal{N}}$, $\sigma(v)$ 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 $V_{\mathcal{R}}$ and $V_{\mathcal{N}}$. For each partial trace, there must exist a unique maximal element $max(V)$ (which may be indentical to $min(V)$).

Notice that, as defined above, the set of partial traces and the set of complete traces are not disjoint. It is convenient, in fact, to extend the definitions so that traces are labeled with a bit that distinguishes partial traces from complete traces, although we omit the details.

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 $\gamma = (V_{\mathcal{R}}, V_{\mathcal{N}})$ and the alphabet of γ is $A = V_{\mathcal{R}} \cup V_{\mathcal{N}}$. A non-degenerate state s is a function with domain A such that for all $v \in V_{\mathcal{R}}$, $s(v)$ is a real number (the value of variable v in state s); and for all $v \in V_{\mathcal{N}}$, $s(v)$ 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 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 Trace Structure Algebras

The basic relationship between trace algebras, trace structures and trace structure algebras was described earlier (see section 2). This section provides a few more details.

A trace algebra provides a set of signatures and a set of traces for each signature. A trace structure 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 signatures. The set P represents the set of possible behaviors of an agent.

A trace structure algebra contains a set of trace structures over a given trace algebra. Operations of projection, renaming, parallel composition and serial composition on trace structures are defined using the operations of the trace algebra, as follows.

Project and renaming are the simplest operations to define. When they are defined depends on the signature of the trace structure in the same way that definedness for the corresponding trace algebra operations depends on the signatures 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.

Sequential composition is defined in terms of concatentation in an analogous way. The only difference from projection and renaming is that sequential composition requires two traces structures as arguments, and concatenation requires two traces as arguments.

Parallel composition of two trace structures is defined only when all the traces in the structures are complete traces. Let trace structure T'' be the parallel composition of T and T'. Then the components of T'' are as follows (M_I and M_O are omitted in pre-post traces):

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

4 Homomorphisms

The three trace algebras defined above cover a wide range of levels of abstraction. The first step in formalizint the relationships between those levels is to define homomorphims between the trace algebras. As mentioned in section 2, trace algebra homomorphisms induce corresponding conservative approximations between trace structure algebras.

4.1 From metric to non-metric time

A homomorphism from metric time trace algebra 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

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

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 $\mathcal{R}$ or $\mathcal{N}$. 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 $v \in V_{\mathcal{R}} \cup V_{\mathcal{N}}$ the function $f(v)$ is stable at t; and for all $a \in M_I \cup M_O$, $f(a)(t) = 0$.*

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

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

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

4.2 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 trace structure be changed by removing M_I and M_O.

Let $y = h(x)$. The initial state of y is formed by restricting $\mu(min(V))$ (the initial state of x) to $V_{\mathcal{R}} \cup V_{\mathcal{N}}$. If x is a complete trace, then the final state of y is $\perp_\omega$. If x is a complete trace, and there exists $a \in M_I \cup M_O$ and time t such that $f(a)(t) = 1$, the final state of y is $\perp_*$. Otherwise, the final state of y is formed by restricting $\mu(max(V))$.

5 Conservative Approximations

Trace algebras and trace structure algebras are convenient tools for constructing models of agents. We are interested in relating different models that describe systems at different levels of abstraction. Let $\mathcal{A}$ and $\mathcal{A}'$ be two trace structure algebras. A *conservative approximation* is a pair of functions (Ψ_l, Ψ_u) that map the trace structures in $\mathcal{A}$ into the trace structures in $\mathcal{A}'$. Intuitively, the trace structure $\Psi_u(T)$ in $\mathcal{A}'$ is an upper bound of the behaviors contained in T (i.e. it contains all abstracted behaviors of T plus, possibly, some more). Similarly, the trace structure $\Psi_l(T)$ in $\mathcal{A}'$ represents a lower bound of T (it contains only abstract behaviors of T, but possibly not all of them). As a result,

$$\Psi_u(T_1) \subseteq \Psi_l(T_2) \text{ implies } T_1 \subseteq T_2.$$

Thus, a verification problem that involves checking for refinement of a specification can be done in $\mathcal{A}'$, where it is presumably more efficient than in $\mathcal{A}$. The conservative approximation guarantees that this will not lead to a false positive result, although false negatives are possible.

5.1 Homomorphisms and Conservative Approximations

A conservative approximation can be derived from a homomorphism between two 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 over $\mathcal{C}$ and $\mathcal{C}'$ respectively, we use the function Ψ_u that maps an agent P in $\mathcal{A}$ into the agent $h(P)$ 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 [2]).

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 Ψ 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.

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.

5.2 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 Ψ 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 Ψ 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 Ψ_{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 embedded software at multiple levels of abstraction.

6 Embedded Software

This section outlines our approach for using multiple levels of abstraction to analyze embedded software. Our motivating example is a small segment of code used for engine cutoff control [1]. This example is particularly interesting to us because the solution proposed in [1] includes the use of a hybrid model to describe the torque generation mechanism.

6.1 Cutoff Control

The behaviors of an automobile engine are divided into regions of operation, each characterized by appropriate control actions to achieve a desired result. The cutoff region is entered when the driver releases the accelerator pedal, thereby requesting that no torque be generated by the engine. In order to minimize power train oscillations that result from suddenly reducing torque, a closed loop control damps the oscillations using carefully timed injections of fuel. The control

problem is therefore hybrid, consisting of a discrete (the fuel injection) and a continuous (the power train behavior) systems tightly linked. The approach taken in [1] is to first relax the problem to the continuous domain, solve the problem at this level, and finally abstract the solution to the discrete domain.

Figure 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 [1] 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).

```
void control_algorithm( void ) {
  // state definition
  struct state { double x1; double x2; double omega_c; } current_state;
  // Init the past three injections (assume injection before cutoff)
  double u1, u2, u3 = 1.0;
  // Predictions
  double sigma_m, sigma_0;

  loop forever {
    await( action_request );
    read_current_state( current_state );
    compute_sigmas( sigma_m, sigma_0, current_state, u1, u2, u3 );
    // update past injections
    u1 = u2;
    u2 = u3;
    // compute next injection signal
    if ( sigma_m < sigma_0 ) {
      action_injection( );
      u3 = 1.0;
    } else {
      action_no_injection( );
      u3 = 0.0;
    }
  }
}
```

Fig. 1. The control algorithm

6.2 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 trace structures are quite similar to conventional semantics. As described earlier, we can also embed pre-post trace structures 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.

This subsection describes the semantics of several basic programming language constructs in terms of pre-post trace structures. The following two subsections describe how these semantics can be integrated into more detailed models.

The semantics of each statement is given by a trace structure. To simplify the semantics, we assume that inter-process communication is done through shared actions rather than shared variables. A pre-post trace structure has a signature γ of the form $(V_{\mathcal{R}}, V_{\mathcal{N}})$. 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 trace structure for the statement in the block includes in its signature the local variables. The trace structure for the block is formed by projecting away the local variables from the trace structure of the statement.

The sequential composition of two statements is defined as the concatenation of the corresponding trace structures: the definition of concatenation ensures that the two statements agree on the intermediate state. The traces in the trace structure 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 a trace structure 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 trace structure 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 trace structure for the definition of `proc`. The parameter passing semantics that results is *value-result* (i.e. no aliasing or references) with the restriction that no parameter can be used for both a value and result. More realistic (and more complicated) parameter passing semantics can also be modeled.

To define the semantics of `if-then-else` and `while` loops we define a function $init(x, c)$ 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, $init(x, c)$ 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

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

Notice that this definition can be used for any trace algebra where $init(x, c)$ 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:

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

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

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

6.3 Using Non-Metric Time Traces

Using an inverse conservative approximation, as described earlier, the pre-post trace semantics described in the previous subsection can be embedded into non-metric time trace structures. However, this is not adequate for two of the constructs used in figure 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.

6.4 Using Metric Time Traces

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

7 Conclusions and Comparisons with Other Approaches

It was not our goal to construct a single unifying semantic domain, or even a parameterized class of unifying semantic domains. Instead, we wish to construct a formal framework that simplifies the construction and comparison of different semantic domains, including semantic domains that can be used to unify specific, restricted classes of other semantic domains.

There is a tradeoff between two goals: making the framework general, and providing structure to simplify constructing semantic domains and understanding their properties. While our framework is quite general, we have formalized several assumptions that must be satisfied by our semantic domains. 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 [10, 11] 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 conservative approximations, we

remove the need to reprove these properties when a new conservative approximation is constructed.

Similarly, our notion of conservative approximation can be described in terms of abstract interpretations. However, abstraction interpretations are such a general concept that they do not provide much insight into abstraction and refinment relationships between different semantic domains.

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 varies in the way they represent the sequence. One example is the Masaccio model ([7, 8]) proposed by Henzinger et alii. 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. The underlying semantic model is based on sequences. The behavior of each component is characterized by a set of finite executions, each of them composed of an entry location and a sequence of steps that can be either jumps or flows. An exit location is optional. The equations associated with the transitions in the state diagram define the legal jumps and flows that can be taken during the sequence of steps.

The operation of composition in Masaccio comes in two flavors: parallel and serial. The parallel composition is defined on the semantic domain as the conjunction of the behaviors: each execution of the composition must also match an execution of the individual components. Conversely, serial composition is defined as the disjunction of the behaviors: each execution of the composition need only match the execution of one of the components. Despite its name, this operation doesn't serialize the behaviors of the two components. Instead, a further operation of *location hiding* is required to string together the possible executions of a disjunction.

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

In our paper we presented three models for semantic domains. Masaccio compares to our more detailed model. In our approach we have decided to model the flows and the jumps using a single function of a real variable: flows are the continuous segments of the functions, while jumps are the points of discontinuity. This combined view of jumps and flows is possible in our framework because we are not constrained by a representation based on differential equations, and hence

we do not require the function to be differentiable. Another difference is that different components are allowed to concurrently execute a jump and a flow, as long as the conditions imposed by the operation of parallel composition are satisfied.

Because in Masaccio the operations of composition are defined on the semantic domain and not on the representation it is easy to do a comparison with our framekwork. Parallel composition is virtually identical (both approaches use a projection operation). On the other hand we define serial composition in quite different terms: we introduce a notion of concatenation that is difficult to map to the sequence of steps that include serial composition and location hiding. In fact, it appears that the serial composition so obtained might contain side-effects that are intuitively not intended in a proper sequential composition of behaviors (because of the projection during the serial composition, a behavior of the compound component might include executions that were not originally present in the components themselves). We believe this could simply be an artifact of the representation based on state transitions that requires the identification of the common points where the control can be transferred.

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

In our framework we must distinguish between two notions of refinement. The first is a notion of refinement within a semantic domain: in our framework this notion is based on pure trace containment. We believe this notion of refinement is sufficient to model the case of sequential systems as well: it is enough to require that the specification include all possible continuations of a common prefix.

The second notion of refinement that is present in our framework has to do with changes in the semantic domain. This notion is embodied in the concept of conservative approximation that relates models at one level of abstraction to models at a different level of abstraction. There is no counterpart of this notion in the Masaccio model.

References

1. A. Balluchi, M. D. Benedetto, C. Pinello, C. Rossi, and A. Sangiovanni-Vincentelli. Cut-off in engine control: a hybrid system approach. In *IEEE Conf. on Decision and Control*, 1997.
2. J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, Aug. 1992. Technical Report CMU-CS-92-179.
3. J. 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.

4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 17(12):1217–1229, Dec. 1998.
10. R. Negulescu. *Process Spaces and the Formal Verification of Asynchronous Circuits.* PhD thesis, University of Waterloo, Canada, 1998.
11. R. Negulescu. Process spaces. In C. Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.
12. V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, Feb. 1986.