Chapter 23

MODELING TECHNIQUES IN DESIGN-BY-REFINEMENT METHODOLOGIES

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

[1]*Cadence Berkeley Labs, Berkeley, CA, 94704.*

[2]*Department of EECS, University of California at Berkeley, Berkeley, CA 94720*

Abstract: Embedded system design methodologies that are based on the effective use of multiple levels of abstraction hold promise for substantial productivity gains. Starting the design process at a high level of abstraction improves control over the design and facilitates verification and synthesis. In particular, if we use a rigorous approach to link the levels of abstraction, we can establish properties of lower levels from analysis at higher levels. This process goes by the name of "design by refinement". To maximize its benefit, design by refinement requires a formal semantic foundation that supports a wide range of levels of abstraction. We introduce such a semantic foundation and describe how it can integrate several models for reactive systems.

Key words: Abstraction, refinement, heterogeneity, semantics.

1. INTRODUCTION

Currently deployed design methodologies for embedded systems are often based on *ad hoc* techniques that lack formal foundations and hence are likely to provide little if any guarantee of satisfying a set of given constraints and specifications without resorting to extensive simulation or tests on prototypes. In the face of growing complexity and tighter time-to-market, cost and safety constraints, this approach will have to yield to more rigorous methods. The objective of the Metropolis project [2] is to provide a *design methodology and the software infrastructure* for embedded systems design, from specification to implementation, using methodologies such as platform-based design, communication-based design and successive refinement. The focus is on formal analysis and synthesis for *heterogeneous* systems, i.e. systems that use different modeling techniques for different parts of designs

that interact with the real world. Metropolis is thus centered on its meta-model of computation, a set of primitives that are used to construct several different models. In this paper we lay the foundations for providing a denotational semantics for the meta-model. In particular we study the semantic domain of several models of computation of interest, and how relationships between these models can be established. To do so, we created a mathematical framework in which to express semantic domains in a form that is close to their natural formulation, i.e. the form that is most convenient for a model. The goal is to have a general framework that encompasses many models of computation, including different models of time and communication paradigms, and yet structured enough to give us results that apply regardless of the particular model in question. At the same time, the framework offers mathematical "tools" to help build new semantic domains from existing ones. Because the framework is based on algebraic structures, the results are independent of any particular design language, and therefore are not just specific to the Metropolis meta-model.

An important factor in the design of heterogeneous systems is the ability to flexibly use different levels of abstraction. Each part of the design undergoes changes in the level of abstraction during the design process and different abstractions are often employed for different parts of a design by way of different models of computation. Abstraction thus comes in different forms that include the model of computation, the scope (or visibility) of internal structure, or the model of the data. Thus, we provide a mathematical framework that allows the user to choose the best abstraction (semantic domain) for the particular task at hand. In this work, we concentrate on semantic domains for concurrent systems and on the relations and functions over those domains. We also emphasize the relationships that can be constructed between different semantic domains. This work is therefore independent of the specific syntaxes and semantic functions employed. Likewise, we concentrate on a formulation that is convenient for reasoning about the properties of the domain. As a result, we do not emphasize finite representations or executable models, which we defer for future work.

2. RELATED WORK

In this section, we give a brief summary of the main approaches. We refer the reader to [6] for a more complete account of related work.

Several formal models have been proposed over the years [9] to capture one or more aspects of computation in embedded systems. Many models of computation can be encoded in the Tagged Signal Model [10]. In contrast, we describe a framework that is less restrictive in terms of what can be used

to represent behaviors, and we concentrate on building relationships between the models that fit in the framework.

Our work shares the basic principles of the Ptolemy project [8] of providing flexible abstractions and an environment that supports a structured approach to heterogeneity. However, while in Ptolemy each model of computation is described operationally in terms of a common executable interface, we base our framework on a denotational representation and de-emphasize executability. Instead, we are more concerned with studying the process of abstraction and refinement in abstract terms. Process Spaces [11] are also 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.

Our notion of conservative approximation is closely related to the Galois connection of an abstract interpretation [7]. In particular, the upper bound of a conservative approximation roughly corresponds to the abstraction function of a Galois connection. However, the lower bound of a conservative approximation appears to have no analog in the theory of abstract interpretations. Thus, conservative approximations allow non-trivial abstraction of both the implementation and the specification, while abstract interpretations only allow non-trivial abstraction of the implementation.

In the Rosetta language [1] domains of agents for different models of computation are described declaratively as a set of assertions in some higher order logic. In contrast, we are not concerned with the definition of a language, and we define the domain directly as a collection of elements of a set. In this sense, the approach taken by Rosetta seems more general. However, the restrictions that we impose on our models allow us to prove additional results that help create and compare the models.

3. OVERVIEW

In the following sections we introduce our framework and concentrate on the basic principles underlying the definitions. We refer the reader to our previous publications [3][4][5][6] for an in depth presentation of specific examples of semantic domains and for a more formal presentation.

3.1 Traces and Trace Structures

The models of computation in use for embedded concurrent systems represent a design by a collection of agents (processes, actors, modules) that interact to perform a function. For any particular input to the system, the agents react with some particular execution, or behavior. In our framework

we maintain a clear distinction between models of agents and models of individual executions. In different models of computation, individual executions can be modeled by very different kinds of mathematical objects. We always call these objects *traces*. A model of an agent, which we call a *trace structure*, consists primarily of a set of traces. This is analogous to verification methods based on language containment, where individual executions are modeled by strings and agents are modeled by sets of strings. However, our notion of trace is quite general and so is not limited to strings.

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

We make a distinction between two different kinds of behaviors: *complete* behaviors and *partial* behaviors. A complete behavior has no endpoint. A partial behavior has an endpoint; it can be a prefix of a complete behavior or of another partial behavior. *Complete traces* and *partial traces* are used to model complete and partial behaviors, respectively.

As an example we may consider traces that are suitable for modeling continuous time, synchronous discrete time and transformational (i.e. non-reactive) systems. In the first case, we might define a trace as a mapping that associates to each signal a function from continuous time (the positive reals) to an appropriate set of values (for example, the reals again). A partial trace in this case is a mapping that associates functions from only a closed time interval $[0, \delta]$, where $\delta > 0$. For an alphabet *A*, the complete traces are defined by $B_C(A) = A \rightarrow (\Re^+ \rightarrow \Re)$, while partial traces are defined by $B_P(A) = A \rightarrow ([0, \delta] \rightarrow \Re)$. We refer to these traces as *metric-time* traces.

An example of a trace suitable for synchronous discrete time systems is a sequence. In this example we assume the signals represent events that occur at distinct instants in time. The corresponding traces are sequences whose elements are subsets of the set of signals (events) available from the alphabet, i.e. $B(A) = (2^A)^\infty$ where the notation ∞ denotes both finite and infinite sequences. Here the partial traces are the finite sequences, while the complete traces are the infinite sequences. We refer to these traces as *synchronous* traces.

Unlike the previous two examples, a transformational system is only concerned with the initial and final state of a computation. If the signals are interpreted as state variables, a corresponding trace may be defined as a pair of mappings associating the state to its initial and final value (from a set of values V), respectively. The case of non-termination is modeled by adding a distinctive value $\perp$ to the set *V*. If we denote with $V_\perp = V \cup \{\perp\}$, partial and

complete traces can be defined as $B(A) = (A \to V) \times (A \to V_\perp)$. We refer to these traces as *pre-post* traces.

Note that a given object can be both a complete trace and a partial trace; what is being represented in a given case is determined from context. For example, a terminating trace above can represent both a complete behavior that terminates or it can represent a partial behavior.

3.2 Trace Algebra and Trace Structure Algebra

In our framework, the first step in defining a model of computation is to construct a trace algebra. The carrier of a trace algebra contains the universe of partial and complete traces for the model of computation. The algebra also includes three operations on traces: *projection*, *renaming* and *concatenation*. These operations are defined to support common tasks used in design, like that of scoping, instantiation and composition of agents. The second step is to construct a trace structure algebra. Here each element of the algebra is a trace structure. Given a trace algebra 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. A trace structure algebra includes four operations on agents: *projection*, *renaming*, *parallel composition* and *sequential composition.*

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

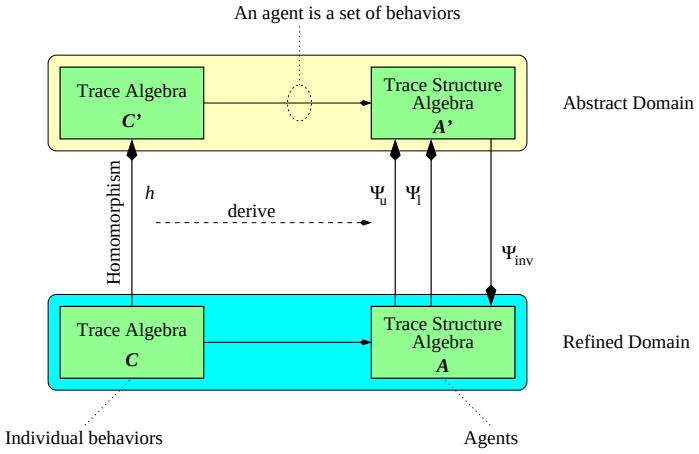

Figure 23-1. Algebras and their relationships

The first operation of trace algebra is called *projection*, and consists in retaining from a trace only the information related to certain signals. The projection operation *proj* takes as argument the set of signals *B* that should be retained. For metric-time traces, projection corresponds to restricting the mapping from the set *A* of signals to a subset *B*. This operation is defined similarly for pre-post traces. Conversely, for synchronous traces projections consists of removing the events not in *B* from each element of the sequence. Projection on trace structures (agents) corresponds to hiding the internal signals, and can therefore be seen as the natural extension to sets of the corresponding operation on individual traces. In other words, the scope of the hidden signals is limited to the agent they belong to.

The second operation is called *renaming*, and consists in changing the names of the visible elements of a trace or an agent. The renaming operation *rename* takes as argument a renaming function *r* that maps the elements of the alphabet *A* into a new alphabet *C*. The function *r* is required to be a bijection in *W* to avoid conflicts of names and potentially a change in the behavior of the agents. In all examples, renaming corresponds to a substitution of signals, and is easily defined in terms of the renaming function *r* or of its inverse. As for projection, renaming of trace structures can be seen as the natural extension to sets of traces of the corresponding operation on individual traces. The effect is that of a renaming of all the signals in the agent: this process corresponds to that of *instantiation* of a master agent into its instances.

Projection and renaming, seen as operators for scoping and instantiation, are common operations that are meaningful to all models of computation. For all trace algebras and all trace structure algebras we require that the operations on traces and trace structures satisfy certain properties. These properties ensure that the operations behave as expected given their intuitive meaning. In addition, we can use these properties as assumptions to prove results that are independent of the particular model of computation in question. These results provide powerful tools that the designer of the model can use to prove general facts about the model and about its relationships with other models.

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

of *T* must be "compatible" with the restrictions imposed by the agents being composed. Thus if *x* is a trace of *T* with alphabet *A*, then its projection $proj(A_1)(x)$ on the alphabet of T_1 must be in P_1, and the projection $proj(A_2)(x)$ on the alphabet of T_2 must be in P_2. The set of traces in *T* must be maximal with respect to that property. Formally:

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

Similarly to projection and renaming, parallel composition of agents must satisfy certain properties. For example we require that it be commutative and associative. A fundamental result of this work is that the properties of the trace algebra are sufficient to ensure that the corresponding trace structure algebra satisfies its required properties.

While parallel composition is at the basis of concurrent models of computation, in other models the emphasis may be on a ``sequential execution'' of the agents. For these models we introduce a third operation on traces called *concatenation.* In the case of synchronous traces, concatenation corresponds to the usual concatenation on sequences. Similarly we can define concatenation for metric-time traces. For pre-post traces, concatenation is defined only when the final state of the first trace matches the initial state of the second trace. The resulting trace has the initial state of the first component and the final state of the second. Note that the information about the intermediate state is lost.

Similarly to the other operations, concatenation must also satisfy certain properties that ensure that its behavior is consistent with its intuitive interpretation. For example we require that it be associative (but not commutative!), and that it behaves consistently when used in combination with projection and renaming. Concatenation induces a corresponding operation on trace structures that we call *sequential composition* by naturally extending it to sets of traces. A more detailed account of sequential composition can be found in [5].

3.3 Refinement and Conservative Approximations

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

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

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

A conservative approximation is actually composed of two mappings. The first mapping is an upper bound of the agent: the abstract agent represents all of the possible behaviors of the agent in the more detailed domain, plus possibly some more. This mapping is usually denoted by Ψ_u. The second is a lower bound: the abstract agent represents only possible behaviors of the more detailed one, but possibly not all. We denote it by Ψ_l.

Conservative approximations are abstractions that maintain a precise relationship between verification results in the two trace structure algebras. In particular, a conservative approximation is defined to preserve results related to trace containment, such that if T_1 and T_2 are trace structures, then $\Psi_u(T_1) \subseteq \Psi_l(T_2)$ implies that $T_1 \subseteq T_2$. When used in combination, the two mappings allow us to relate results in the abstract domain to results in the more detailed domain. The conservative approximation guarantees that this will not lead to a false positive result, although false negatives are possible.

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

A homomorphism *h* is a function between the domains of two trace algebras that commutes with projection, renaming and concatenation. Consider two trace algebras ***C*** and ***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 ***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 ***A*** and ***A'*** are trace structure algebras constructed from ***C*** and ***C'*** respectively, we use the function Ψ_u that maps an agent with traces *P* in ***A*** into the agent with traces *h(P)* in ***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 $h(P) - h(B(A) - P)$. 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 [3]).

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.

3.4 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 ***A*** and ***A'*** be two trace structure algebras, and let Ψ be a conservative approximation between ***A*** and ***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 ***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 heterogeneous systems that use models of computation at multiple levels of abstraction. Assume we want to compose two agents T_1' and T_2' that reside in two different trace structure algebras $\boldsymbol{A_1}$ and $\boldsymbol{A_2}$. To make sense of the composition, we first define a third, more detailed trace algebra that has homomorphisms into the other two. Thus we can construct a third, more detailed, trace structure algebra ***A*** with conservative approximations induced by the homomorphisms. The inverse of these conservative approximations are used to map T_1' and T_2' into their corresponding detailed models T_1 and T_2. The composition then takes place in the detailed trace structure algebra.

4. CONCLUSIONS

The goal of trace algebras is to make it easy to define and to study the relationship between the semantic domains for a wide range of models of computation. All the models of importance "reside" in a unified framework

so that their combination, re-partition and communication may be better understood and optimized. This unified approach will provide a designer a powerful mechanism to actually select the appropriate models of computation for the essential parts of his/her design.

Our representation of agents is denotational, in that no rule is given to derive the output from the input. The algebraic infrastructure allows us to formalize a semantic domain in a way that is close to a natural semantic domain for a model of computation. In addition it introduces additional concepts such as hierarchy, instantiation and scoping in a natural and consistent way. In particular we are concentrating on using the concept of a conservative approximation to study the problem of heterogeneous interaction.

REFERENCES

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

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

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

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

[5] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Using multiple levels of abstraction in embedded software design. In T. A. Henzinger and C. M. Kirsch, editors, 1st International Workshop, EMSOFT 2001, vol. 2211 of LNCS. Springer-Verlag, 2001.

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

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pages 238--252, Los Angeles, California, 1977.

[8] J. Davis II, et al. Heterogeneous concurrent modeling and design in java. Technical Memorandum UCB/ERL M01/12, EECS, Univ. of California, Berkeley, Mar. 2001.

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

[10] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. IEEE Transactions on CAD, 17(12):1217--1229, Dec. 1998.

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