A General Framework for Formalizing UML with Formal Languages

William E. McUmber and Betty H.C. Cheng
Department of Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, MI 48824, USA
(517) 355-8344
{mcumber,chengb}@cse.msu.edu

Abstract

Informal and graphical modeling techniques enable developers to construct abstract representations of systems. Object-oriented modeling techniques further facilitate the development process. The Unified Modeling Language *(UML), an object-oriented modeling approach, could be broad enough in scope to represent a variety of domains and gain widespread use. Currently, UML comprises several different notations with no formal semantics attached to the individual diagrams. Therefore, it is not possible to apply rigorous automated analysis or to execute a UML model in order to test its behavior, short of writing code and performing exhaustive testing. We introduce a general framework for formalizing a subset of UML diagrams in terms of different formal languages based on a homomorphic mapping between metamodels describing UML and the formal language. This framework enables the construction of a consistent set of rules for transforming UML models into specifications in the formal language. The resulting specifications derived from UML diagrams enable either execution through simulation or analysis through model checking, using existing tools. This paper describes the use of this framework for formalizing UML to model and analyze embedded systems. A prototype system for generating the formal specifications and results from an industrial case study are also described.*

Keywords: Object-oriented modeling, formal specifications, model checking

1. Introduction

Object-oriented modeling techniques, typically semi-formal and graphical, enable developers to construct abstractions that are domain- and application-specific. As a result, object-orientation has been increasingly used for large-scale systems development as a means to manage complexity and facilitate reuse. The *Unified Modeling Language* (UML) [18], an object-oriented modeling approach, could be broad enough in scope to represent a variety of domains and gain widespread use. Currently, UML comprises several different notations with no formal semantics attached to the individual diagrams. Therefore, it is not possible to apply rigorous automated analysis or to execute a UML model in order to test its behavior, short of writing code and performing exhaustive testing.

We introduce a general framework for formalizing a subset of UML diagrams in terms of different formal languages based on a mapping between metamodels describing UML and a formal language. This framework enables the construction of a consistent set of rules for transforming UML models into specifications in the formal language. The resulting specifications derived from UML diagrams enable either execution through simulation or analysis through model checking, using existing tools.

A given application domain determines what kind of formal semantics should be attached to the UML diagrams (since it is possible for a given set of UML diagrams to have more than one interpretation). Therefore, the intent of our formalization framework is to produce *a* semantics for a set of UML diagrams for modeling systems in a given domain, not *the* definitive semantics for UML for modeling all types of systems. For example, a UML state diagram

0-7695-1050-7/01 $10.00

used to model the behavior of a management information system (MIS), such as a financial investment system, will require different formal semantics than a state diagram used to model a flight controller. In order to increase the likelihood of industrial use of our formalization framework, we chose to model embedded systems.

Embedded systems are typically 10 to 100 times more common than their desktop counterparts [5], residing in systems ranging from engines, to toasters, to autopilots. The software for embedded systems is, in general, more difficult to write and debug because it usually involves time-dependent sections in difficult to instrument situations. Furthermore, embedded systems usually must achieve a higher level of robustness and reliability because they control real-world physical processes or devices upon which we depend, frequently, in a critical way. Consequently, methods for modeling and developing embedded systems and rigorously verifying behavior before committing to code, are increasingly important.

Currently, much of the embedded systems industry use *ad hoc* development approaches [6]. Frequently, there are few, if any, intermediate steps between high-level, prose descriptions of requirements and code written in the target implementation language, such as C. We contend that object-oriented methods are one remedy to the development of embedded systems software. The gains are in part due to the often close correlation between real components in the physical system and software objects, although other factors such as data hiding, encapsulation, *etc*, also contribute.

Our formalization of UML follows the well-established method of mapping a semi-formal language to a formal language. Two significant problems with this formalization approach, however, are completeness and consistency of the rules that map one language to another. While there is no "incorrect" semantics,[1] inconsistent mapping rules can introduce unexpected behavioral consequences.

Consistency and completeness of the rules are addressed by basing the diagram formalizations on mappings between *metamodels* of the modeling notations. A metamodel is a model of the notation itself, depicted in the class-structure notation, where "classes" represent syntactic components of the modeling language. A *homomorphic mapping* is established between the metamodels of the semi-formal (source), UML and the formal (target) languages. We require each mapping rule to conform to the homomorphic mapping between metamodels. The resulting specifications derived from UML diagrams enable either execution through simulation or analysis through model checking, using existing tools. The mapping process from UML to a target language has been automated in a tool called *Hydra*. Using this framework and Hydra, we have successfully performed an industrial case study involving the design of an embedded system from the automotive domain. In order to leverage existing specification languages and tools that are applicable to the embedded systems domain, we have formalized UML in terms of VHDL [13] and Promela for use with the SPIN model checker [11]. (This paper introduces the Promela formalization, and the VHDL formalization is described in [13].)

The remainder of this paper is organized as follows. In Section 2 we briefly review UML and metamodels, and review Promela/SPIN, one of two target specification languages that we have investigated. A supporting piece for this framework is the formalization of the class diagram and the notation used to depict the metamodels, which is presented in Section 3. Section 4 discusses the practicalities of the metamodel mapping between UML and existing specification languages. Section 5 describes how the homomorphic mapping and formalization rules work together to provide semantics, and Section 6 discusses the use and applications of the overall framework. Section 7 reviews related work. Finally, we give concluding remarks and discuss current and future investigations in Section 8.

2. Background

This section overviews the source semi-formal language and a target formal language that we have explored. One overarching goal of this research, motivated by technology transfer objectives, is to enable developers to continue to use widely accepted development techniques, both in terms of modeling languages as well as the target specification language. The framework that we have developed permits UML diagrams to be formalized in terms of a variety of formal languages. While UML offers several different notations, our preliminary investigations indicate that for modeling requirements and high-level design, the class and state diagrams are usually sufficient for modeling embedded systems, although sequence diagrams can often also be useful.

2.1. UML

The Unified Modeling Language (UML) [18] is described as a "general-purpose visual modeling language that is designed to specify, visualize, construct, and document the artifacts of a software system". UML is based on a series of diagrams that depict the class structure, dynamic properties, and event sequencing for an object-oriented (OO) software system. UML is an extension and melding of several modeling languages, most notably the Object Modeling Technique [17] (OMT) and StateCharts [9]. UML class diagrams and dynamic model diagrams use notation similar to

[1] There is also no absolute "correct" semantics. "Correct" and "incorrect" are in terms of what a designer reasonably expects to happen for a given syntactic construct for a particular problem domain.

OMT. Unlike OMT, UML has no functional model to depict dataflow.

UML contains several distinct types of diagrams: Class Diagrams, Use Case Diagrams, Dynamic Models (state diagrams), Interaction Diagrams (sequence and collaboration diagrams), Activity Diagrams, and Deployment Diagrams. In this paper we deal primarily with the class diagram and the state diagram: The *class diagram* depicts the classes and the interrelations between the classes. Figure 1 shows how the class diagram notation is used for depicting a metamodel (the description of the metamodel is given in the next section). Classes are drawn as rectangles with relationships between classes drawn as annotated lines between the rectangles. There are four types of relationships between classes. *Association* is a binary relationship between two classes. Multiplicities on associations are written as a number at each end, with the number applying to instances at that end of the line. An optional instance is denoted by "0..1", "*" indicates many, and "1..*" denotes one or many. Three additional relationships are *subtype, aggregation*, and *composition*. Subtyping is denoted by a small hollow triangle on the superclass end of the association. The derived class end of the subtype relationship is not marked. Filled and empty diamonds are placed on the aggregate end of composition and aggregation relationships, while the classes that constitute the parts of the aggregation are not marked. Both aggregation and composition indicate collections where the part plays a role in the behavior of the whole, but composition is the stronger relationship because the existence of the part depends on the existence of the whole. In aggregation, this dependency does not hold.

The *dynamic model* is based on Statechart [9] conventions and describes dynamic behavior of objects through the use of Statechart-like notation. States are drawn as rounded rectangles with transitions between states drawn as arrows between the boxes, which indicate the direction of the transition. Transitions are labeled with a transition event, which has its own syntax. Composite states may contain further sets of state diagrams. Concurrency of composite state machines is indicated by separating the composite states with dotted lines.

The syntax of each type of UML diagram is described by a metamodel. An instance of a metamodel is a UML diagram just as an instance of a UML class diagram is a set of objects in a system. In order to explicitly address the integration of the class diagrams and the dynamic model, we composed, or integrated, their respective metamodels, as shown in Figure 1, where a dashed line delimits the class metamodel. The dynamic model portion of the metamodel describes the interrelationship between states, transitions, events and the other parts of the UML dynamic model. Classes decorated with italics are abstract, and therefore are not realized by actual components in a UML dynamic

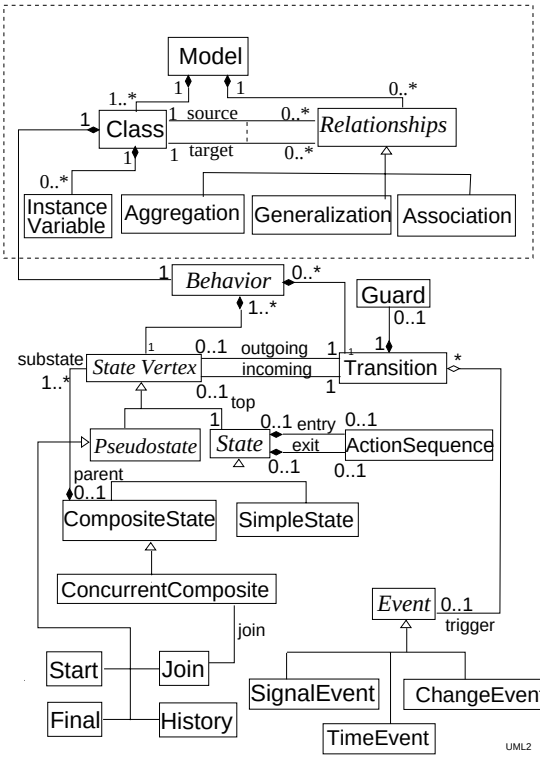

Figure 1. Metamodel of a UML dynamic model.

model diagram. The triangle pointing to class *State* in Figure 1 signifies a specialization association between "*State*" and "CompositeState", or "SimpleState". Since "*State*" is abstract, components of type "*State*" will not appear in a dynamic model but components of types "CompositeState" or "SimpleState" will appear wherever a type "*State*" is required.

Based on the metamodel in Figure 1, a "*Behavior*" is constrained to be an aggregation of "*State Vertex*" that are realized as "CompositeState" and/or "SimpleStates". A "CompositeState" is further specified to be an aggregation of "*State Vertex*", each of which is either a "*State*" or a "*PseudoState*", forming a recursive containment relationship. "*StateVertex*" has two required associations with transitions, one in an outgoing role and another in an incoming role.

2.2. Promela/SPIN

We have formalized UML in terms of Promela, the system description language for SPIN [11]. A specification written in Promela can execute in a simulation environment, or model checking techniques can be applied to verify a

range of properties. We are particularly interested in exploring how simulation and model checking can be used in tandem to analyze and validate specifications for embedded systems.

The syntax of Promela is loosely based on the C language. Promela programs consist of *processes*, *channels*, and *variables*. Processes are global objects running asynchronously and can be created dynamically. Channels and variables may either be local or global. The language was influenced significantly by the Dijkstra "guarded command language" [4] and CSP [10]. There are, however, important differences. Dijkstra's language has no primitives for process interaction. CSP was based exclusively on synchronous communication, constructed in Promela as an unbuffered channel, but Promela also permits buffered channels allowing the construction of message queues. Also in CSP, the type of statement that can appear in guards is restricted, while Promela has no restrictions.

3. Metamodel Formalization

Unless the class model itself is formalized, the mappings developed between metamodels will not rest upon a rigorous basis. This section presents a brief description of the formalization of the class model, which enables the formal description of homomorphic mappings between metamodels.

The class model, upon which the metamodel is built, has been formalized by Bourdeau and Cheng [1]. They showed that a class model[2] can be formalized using algebraic specifications and specific algebras related to OMT instance diagrams. In their formalization, classes are represented as types, and associations between classes are formalized as relations with various properties. We draw upon their work by also representing classes as types and associations as relations, but we do not use algebras.

Specifically, a class model is formalized as a set of typed entities where the class of the entity corresponds to the entity type. Class membership can therefore be expressed with a typing predicate. We write T_{class} for a class predicate, and therefore for a component x, in class y, $T_y(x)$ is true.

In UML, types and classes are not exactly the same [19], although they behave almost identically. The differences lie in realization, where an object may be of several types but of only one class. For the discussion of metamodels we associate a type with each class.

Class relationships are formalized as binary predicates between members of each class. For example, in Figure 1 a model component x of type "State Vertex" and a component y of type "Transition" are related by the predicates $incoming(x, y)$ and $outgoing(y, x)$.

Class specialization, also called *subclassing*, is constrained by requiring each object of a subclass to be a member of the superclass's type. Formally, for subclass C and superclass P,

$$\forall x \, (T_C(x) \Longrightarrow T_P(x)). \tag{1}$$

Expression (1) also expresses the relation between virtual classes, containing no objects themselves, and subclasses. Additional details of the class formalization are not included due to space constraints [14].

4. Semantics For Semi-Formal Models

Supplying precise semantics to a semi-formal UML model is achieved by describing a set of mapping rules from UML to the target formal language. In order to be complete and consistent, each rule is constrained by a homomorphic mapping between metamodels of the source and target languages.

4.1. Homomorphic Mappings on Metamodels

Homomorphic mappings have the important property that they preserve structural relationships between entities in two different systems. This property enables compositional, semantic-preserving mappings from one system to another.

In algebras, a homomorphic mapping maps one algebra to another with the property of preserving operations [8]. For algebra A with binary operation $\oplus$, algebra B with operation $\otimes$, and a homomorphic mapping h that maps elements of A to B, we have

$$a, b \in A \quad h(a \oplus b) = h(a) \otimes h(b).$$

We define a homomorphic mapping from the source metamodel to the target metamodel that preserves *relationships*. The homomorphism maps one metamodel class to another and one predicate relation to another. For example, suppose the source language metamodel contains classes (types) A and B that are mapped by homomorphism h to classes (types) A' and B' in the target language metamodel. Also assume there exists a relation R between A and B that is mapped by h to R'. Then we require h to satisfy:

$$\begin{aligned} &\forall x, y \; (T_A(x) \wedge T_B(y) \wedge R(x, y) \Longrightarrow \\ &R'(h(x), h(y)) \wedge T_{A'}(h(x)) \wedge T_{B'}(h(y))) \end{aligned} \tag{2}$$

Expression (2) requires classes A and B that are associated by R to map to classes A' and B', respectively. Expres-

[2] Bourdeau and Cheng's work pertained to OMT class models but the differences between OMT and UML class models are minor and mostly related to multiplicity notations. Most symbols are common between UML and OMT.

sion (2) further requires the existence of an association R' in the target metamodel connected to R by h between classes A' and B'. We note in passing that h is not necessarily injective, and with actual metamodels, is very rarely so. In the above example, there may be other relations between A' and B' other than R', but minimally, R' must exist.

4.2. Handling Structural Differences

It is rarely the case that the source metamodel and target metamodel are structurally identical. In general, we are given a metamodel of the source language, but since the goal is to produce a mapping to supply semantics to the source language, we have some flexibility when constructing the target metamodel to choose and arrange the parts of the target language that are semantically relevant. In order to make the mapping straightforward, when possible, a target class is constructed for each source class in the source metamodel. Since the function is homomorphic, it need not be surjective, therefore, there may be more classes in the target than in the source. For example, we develop the Promela metamodel shown in Figure 2, and it contains subtypes of "ActionSequence" that are not in the UML metamodel in Figure 1. Similarly, there may be more associations in the target than in the source. The intent of the mapping is to produce a correspondence between the source language and the target language, and not to be a generator (from a grammar standpoint) of the target language.

4.3. Target Metamodel Templates

The target languages that we have typically selected are at least executable in simulation and therefore often have statements that describe more details of execution than the constructs in the source UML metamodel. For example, when using the target language Promela, a UML metamodel class "SimpleState" maps to Promela metamodel class "State Block" represented as a series of Promela statements presented as a *template*.[3] Each template represents a parameterized series of target language statements. Templates contain *metaterms*, denoted by angle brackets, as placeholders for statements that are either extracted directly from the UML diagram or are generated by other rules. Rule Promela 1 in Figure 4 (explained below), shows an example of a template for "State Block". In the template for Rule Promela 1, `<entry actions>` is a placeholder for Promela statements mapped from actions contained on an "entry" declaration for that state in a UML dynamic model. Using detailed templates permits the automatic generation of target language specifications that are executable. More examples of templates are presented in the following section.

[3]By *template*, we do not mean *template class* as found in C++, but rather a specific pattern of target language statements.

5. Formalization Rules For UML

With the formalization of the homomorphism between metamodels established, we can describe the construction of rules mapping UML to a target specification language. A UML formalization consists of a 4-tuple (S, T, h, R). S and T are the metamodels of the source and target languages, respectively, and h is the homomorphism between S and T as described earlier. The formalization rules, R, provide the specific mappings of the semi-formal source to a target formal specification language. Each rule specifies what target language construct(s) shall be derived for a given source language construct. As described earlier, this is generally accomplished with a template showing how target language statements are arranged for a given class or classes in the target metamodel. The constraints for the application of rules are also explicitly enumerated and describe how the metaterms in the templates are formed. Rules mapping one or more source components to target components must be consistent with the homomorphism by preserving relationships in the target language metamodel. Each rule must have a direct correspondence with the homomorphic mapping, h, between metamodels. Since we claim the homomorphism preserves structure and provides consistency, the rules constructed for the mapping between source and target languages must also preserve structure and be consistent.

In order to illustrate the interaction between metamodels, a homomorphism, and a set of mapping rules, we examine portions of two rules from the formalization of UML with Promela. Space does not permit entire rules (templates are shown) or the inclusion of the entire mapping rule set, but this is not needed to show how the homomorphism constrains the rules.

The metamodel for Promela is shown in Figure 2. Unlike UML, the Promela metamodel does not pre-exist. Since we have some freedom in the structure of the target language metamodel, the target metamodel can be structured to make the homomorphism as simple as possible without loss of generality.

Figure 3 gives an abbreviated portion of the homomorphic mapping from the UML metamodel shown in Figure 1 to the metamodel given in Figure 2.

In Figure 2, a UML "SimpleState" is a state that contains no states, while a "CompositeState" can contain "SimpleStates" and other "CompositeStates". Figure 4 shows Rule Promela 1 for mapping "SimpleState" (Figure 1) to "StateBlock" (Figure 2). Rule Promela 1 calls for a "SimpleState" to be formalized as a *State Block* constructed from the template shown in Figure 4. In this template, `state-name` is the name of the state, `object-name` is the name

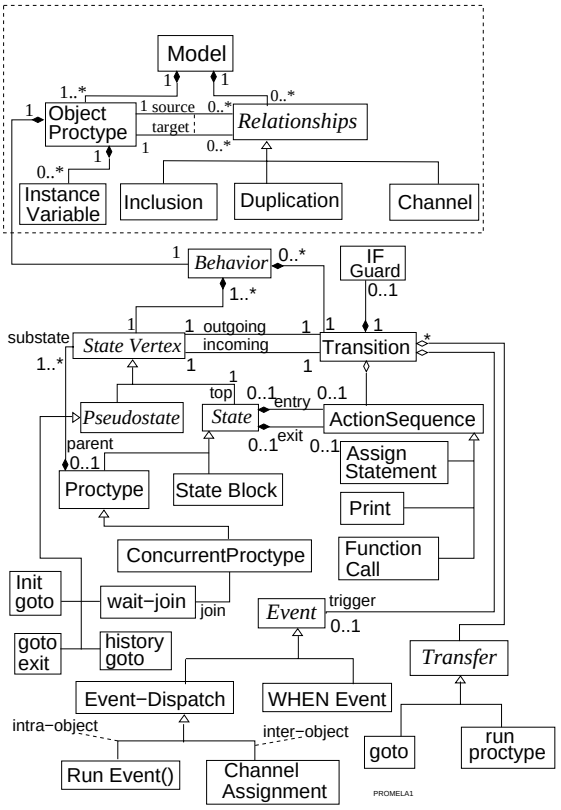

Figure 2. The metamodel for SPIN/Promela metamodels.

of the enclosing object, and `composite-state-name` is the name of the enclosing composite state (or object if this is the highest level). The metaterms `<transition event expression j>` are channel receive expressions for `<event-name j>` described in other rules. Metaterm `<guard list j>` is the placeholder for the construction of transition guards also described by other rules. `<action list j>` is a list of Promela statements for the action on the transition, and `<send list j>` is a list of channel send operations for messages sent by the transition. The second line of the template is present only if a history state is present in the enclosing composite state.

Figure 3 shows that the homomorphism maps UML metamodel class "SimpleState" to "State Block". Similarly, "*State*" is mapped to "*State*" and "ActionSequence" is mapped to "ActionSequence". Since the relationship is homomorphic over relationships, the relationships `entry` and `exit` between "*State*" and "ActionSequence" in the UML metamodel (Figure 1) must be preserved between "*State*" and "ActionSequence" in the Promela metamodel

UML Metamodel	⇒ *Promela Metamodel*
Model	⇒ Model
Class	⇒ Object-Proctype
Relationships	⇒ Relationships
State Vertex	⇒ State Vertex
Transition	⇒ Transition
Pseudostate	⇒ Pseudostate
State	⇒ State
CompositeState	⇒ Proctype
SimpleState	⇒ State Block
ActionSequence	⇒ ActionSequence

Figure 3. A few of the homomorphic mappings of components from the UML metamodel to the Promela metamodel.

(Figure 2). As a consequence, in order for Rule Promela 1 to conform with the Promela metamodel (and the homomorphism), the "State Block" template must have relationships `entry` and `exit` with "ActionSequence" (see Figure 2) by virtue of its inheritance from the virtual class "*State*". Lines 3 and 15 in Rule Promela 1, respectively, show `<entry actions>` and `<exit actions>` in satisfaction of the metamodel requirements and conformance with the homomorphism.

Compare Rule Promela 1 with the template in Rule Promela 2, the rule for Composite States in Figure 5. Rule Promela 2 calls for a composite state named `composite-state-name` to be formalized as a `proctype` (a *proctype* defines a Promela process) with a formal parameter of type `mtype`. A `proctype` representing a composite state is activated in the parent composite state or object with a `run(first-state-name)` statement that passes the name of the state to begin, or value `none`, when the transition is to the boundary of the composite state. The template shown in Figure 5 is used to construct "Proctype". The metaterm `<initial state sequence>` is either a history state construct or an initial state construct, as required. Metaterm `<state sequences>` is defined in the rules as an aggregate of bodies of simple states and transfers to composite states.

In Figure 2, Promela metamodel class "Proctype" inherits from "*State*" just as "State Block" does. Therefore, "Proctype" must also contain a sequence of statements for `entry` and `exit` actions that must be reflected in the rule template. "Proctype" also has a recursive containment relationship through classes "*State*" and "*State Vertex*" that must also be preserved in the template. Conformance with the metamodel is achieved in Rule Promela 2 by specifying that `<state sequences>` contain "State Blocks" or calls to other `proctypes`.

Rule Promela 1

```
1  state-name:
2     H_composite-state-name = st_statename;
3     <entry actions>
4     evt!<event-name 1>,_pid;
5     evt!<event-name 2>,_pid;
6        .
7        .
8        .
9     evt!<event-name n>,_pid;
10    if
11    ::  <transition event expression 1>
12          -> <guard list 1>
13       <action list 1>
14       <send list 1>
15       <exit actions 1>
16       goto nextstate
17    ::  <transition event expression 2>
18          -> <guard list 2>
19       <action list 2>
20       <send list 2>
21       <exit actions 2>
22       goto nextstate
23       .
24       .
25       .
26    ::  <transition event expression n>
27          -> <guard list n>
28       <action list n>
29       <send list n>
30       <exit actions n>
31       goto nextstate
32    fi;
```

End of Rule Promela 1.

Figure 4. The UML to Promela rule mapping simple states to equivalent Promela specifications.

Rule Promela 2

```
1 proctype composite-state-name(mtype state)
2 {atomic{
3 <initial state sequence>
4 <entry actions>
5 <state sequences>
6 exit:      <exit actions>
7            skip
8 }}
```

End of Rule Promela 2.

Figure 5. The UML to Promela rule mapping composite states to equivalent Promela specifications.

If Rule Promela 2 were formulated without the constraint of the Promela metamodel that is itself constrained by the homomorphism between the UML metamodel and the Promela metamodel, then it would be easy to overlook the requirement of entry and exit actions on composite states. Since "Proctype" and "State Block" are common descendants of "*State*" in the Promela metamodel (as required by the homomorphism), both classes must have similar constructs, such as entry actions, exit actions, and the incoming and outgoing relationships with transitions. Because of the homomorphism, the structural relationship of the elements in the UML model match the structure of the generated Promela specification.

For completeness, every UML metamodel class must map, via the homomorphism, to some class in the Promela metamodel. Since relationships between metamodel classes must be preserved, no UML metamodel class is omitted from the mapping rules.

As a concrete illustration of specifications produced by this framework using Hydra[4], Figure 6 shows a simple UML model consisting of two classes. The dynamic model for class "ClassOne" is shown in the same diagram, as indicated by the dashed line. Rule Promela 1, working in conjunction with other rules, produces the Promela specification shown in Figure 7. Rule Promela 1 produces lines 20 to 35, and lines 36 to 44, representing simple states A and B, respectively, in Figure 6. The label `A_G` on line 24 is generated as a consequence of a guarded transition by another rule. Lines 1 through 11 are generated by a rule that establishes global declarations based on the class model. Lines 12 and 49 show the generation of `proctypes` as class containers for classes "ClassOne" and "ClassTwo", respectively, as dictated by rules that map class structure.

6. Tool Support and Case Study

Since the mapping from UML is concrete and specific to executable sets of formal language statements, it is possible to construct an automated tool that will transform a UML diagram to a specification in the formal language. The specification can then be used with tools specific to the formal language. We have constructed such a prototype tool called Hydra that translates an instance of a UML diagram into

[4]As mentioned in the introduction, *Hydra* is our tool using the described framework for producing Promela (or other target language specifications) directly from UML diagrams.

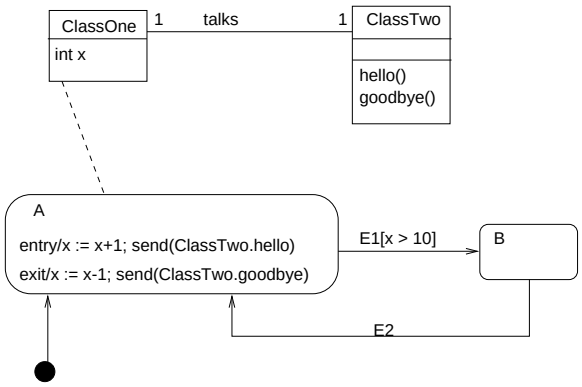

Figure 6. A simple UML model consisting of two classes. The dynamic model for Class "ClassOne" is shown in the lower portion of the diagram.

executable specifications using this framework. Currently, Hydra can generate either VHDL or Promela specifications. The resulting specifications can be run in simulation to determine the behavior of the dynamic model described by the UML model or, in the case of Promela, used with SPIN model checking analysis tools.

6.1. Industrial Automotive Application

In an industrial case study, we used Promela as the target language for the design of a "Smart Cruise Control" for automobiles. Smart Cruise control augments standard cruise control with a small radar unit to monitor the location of vehicles ahead of the car. When necessary, the speed of the car is slowed to maintain a safe trailing distance from the vehicle ahead. When the leading vehicle is no longer present, the Cruise Control resumes the initial speed set by the driver. The system includes numerous checks and warnings that can be issued to the driver when unsafe conditions are detected.

The formalization of UML enables us to perform structural as well as behavioral analysis of the diagrams. Structural analysis includes inter- and intramodel consistency checks using utilities within Hydra. Example errors include use of an instance variable without it being declared, use of a signal/message without it being declared, or expecting a signal/message that no object sends. The first two errors are inconsistencies between a class and its corresponding state diagram, while the third is an inconsistency among the state diagrams contained in the entire model. We were able, using Hydra, to move directly from UML class and behavior diagrams to model checking and simulation. SPIN's model checking and simulation capabilities were extremely useful during the behavior analysis of the Smart Cruise Design.

```
1  chan evq=[10] of {mtype,int};
2  chan evt=[10] of {mtype,int};
3  chan wait=[10] of {int,mtype};
4  mtype={goodbye, hello, E1, E2};
5  typedef ClassOne_T {
6                 int x;
7                 }
8  ClassOne_T ClassOne_V;
9  chan ClassTwo_q=[5] of {mtype};
10 chan t=[1] of {mtype};
11 mtype={free};
12 proctype ClassOne()
13 {
14 mtype m;
15 int dummy;
16                ClassOne_V.x = 1;
17 /* Init state */
18                goto A;
19 /* State A */
20 A:             printf("in state ClassOne.A");
21                atomic{ClassOne_V.x=ClassOne_V.x+1;
22 ClassTwo_q!hello;; }
23
24 A_G:
25                evq!E1,_pid;
26                atomic {if ::  !t?[free] ->
27 t!free ::  else skip fi;}
28                if
29                ::  evt??E1,eval(_pid) -> t?free; if
30                   ::  ClassOne_V.x>10 ->
31                      ClassOne_V.x=ClassOne_V.x-1;
32 ClassTwo_q!goodbye;; goto B
33                   ::  else -> goto A_G
34                   fi
35                fi;
36 /* State B */
37 B:             printf("in state ClassOne.B");
38                evq!E2,_pid;
39                atomic {if ::  !t?[free] ->
40 t!free ::  else skip fi;}
41                if
42                ::  evt??E2,eval(_pid) ->
43 t?free; goto A
44                fi;
45 exit:      skip
46 }
47
48
49 proctype ClassTwo()
50 {
51 mtype m;
52 int dummy;
53 exit:      skip
54 }
```

Figure 7. Hydra-generated Promela specifications produced from the UML class and dynamic model shown in Figure 6.

Simulation is used to validate the behavior of the system and enables us to focus the model checking efforts on specific parts of the system exhibiting erroneous behavior. Using SPIN *never claims* we verified a number of critical temporal properties of the system, including safety and freedom from deadlock. For example, model checking discovered a missing transition in a series of states entered only when the cruise system had to be shut down by an exception event. The missing transition is only needed when a specific set of events occur in a certain order. Finding the missing transition through simulation would require either luck or many simulation runs. Furthermore, being able to visually correlate the behavior diagram and the model checking results greatly eased the discovery and repair of this problem. Due to space constraints, details of this case study are not included [3].

7. Related Work

This section briefly discusses other projects that have investigated formalization of UML diagrams using formal target languages.

Other work has approached this problem from two major directions. In the first, UML is mapped to another language such as Z, but there is no overarching consistency-generating framework for the rule generation. The pUML Project [2, 20] is an effort to formalize the meaning of UML diagrams using the Z language as its formalization vehicle. In the second approach, UML models are first mapped to an intermediate form, then mapped to a formal target language. For example, Latella, *et al.* [12] have formalized UML state diagrams through a mapping to extended automata, but not in the context of the class diagram. Our homomorphism between metamodels provides a direct mapping from UML to the chosen target language.

There have been other attempts to add formality to UML. For example, when using UML, it is also possible to annotate the diagrams with the Object Constraint Language (OCL) [15] that uses simple logic for specifying invariant properties of systems comprising sets and relationships between sets. OCL is intended to be a light-weight formal specification language, for annotation purposes (e.g., invariants, guards, pre-/postconditions for methods), rather than to be used as the basis for extensive (automated) behavior analysis, such as simulation and model checking. In addition, there is no formal mechanism to integrate the information from the different UML models or provide a means to rigorously trace the requirements down to implementation. Robbins, *et al.* have also extended UML notation to enable architecture-based modeling for languages such as C2 and Wright [16].

We leverage previous work by Wang and Cheng [22, 21, 23] into the formalization of OMT [17], which in many ways is the predecessor to UML. Their formalization targeted LOTOS specifications only. In contrast, we have developed a general framework for formalizing UML that may have several target languages.

The idea of a homomorphism between metamodels is closely related to *Institution morphisms* [7], however, our homomorphism is most likely not an institution morphism since UML is not a formal system.[5] Also, the idea of a relational homomorphism does not seem to be a required feature of an institution morphism.

8. Conclusions

We have presented a framework for formalizing the semantics of a key set of UML diagrams using a mapping from UML metamodels to formal language metamodels. The approach has the advantage that no intermediate mappings are required. Furthermore, the mapping leads to a set of rules for constructing specifications in the formal language. We have constructed Hydra, a prototype tool to demonstrate that the rules are sufficiently defined that formal language specifications can be generated automatically from UML diagrams. Although we do generate executable specifications, code generation is not the emphasis of this work. The intent of the generated specifications is to attach formal semantics to UML diagrams.

Our initial target formal languages are VHDL and Promela/SPIN, and we have applied the framework to the analysis of embedded systems. Although a concrete semantics is required for any specific project, it is possible to alter UML semantics by modifying the mapping rules and the homomorphism.

While the mappings of UML to VHDL and Promela provide very similar semantics, there are differences that may affect analysis in a specific project. VHDL contains the ability to perform precise timing simulations, while Promela has virtually no timing capability. On the other hand, the SPIN model checker can be used with Promela models[6] to explore extended behavior, which is not currently possible directly with VHDL models. Fairness constraints were much easier to achieve in VHDL than Promela. In fact, fairness was very hard to achieve in Promela models. From a mapping standpoint, it was easier to map inter-state messaging across the UML hierarchical state machine structure to VHDL while Promela required considerable extra work. On the other hand, class structure was more straightforward to achieve in Promela and relatively difficult in VHDL.

[5] It has not been formally proven whether or not a set of UML diagrams is an institution.

[6] In fact, Hydra can pre-process LTL temporal claims written in the notation used in the UML model, incorporating the constructs required for model checking into the Promela specification.

We expect to include other diagrams in the homomorphic mapping, and consequently into the mapping rules. In particular we expect that UML *use cases* and *sequence diagrams* will be the primary source of drivers to test the models in both simulation and for model checking, and can automatically provide further constraints to verify a design.

Acknowledgements

The authors are grateful for R.E.K. Stirewalt's detailed comments on earlier drafts of this paper. L.A. Campbell and R. Stephenson have provided invaluable assistance on this project through their contributions on the Hydra system; A. Torre was instrumental to the project in providing the project specification and domain expertise for the smart cruise case study. Finally, this work has been supported in part by National Science Foundation grants EIA-0000433, CCR-9901017, CDA-9700732, CDA-9617310, CCR-9633391, and DARPA grant No. F30602-96-1-0298, managed by Air Force's Rome Laboratories, and Eaton Corporation.

References

[1] R. H. Bourdeau and B. H. C. Cheng. A formal semantics of object models. *IEEE Trans. on Software Engineering*, 21(10):799–821, October 1995.

[2] J.-M. Bruel and R. B. France. Transforming UML models to formal specifications. In *UML'98 - Beyond the notation*, LNCS. Springer, 1998.

[3] L. A. Campbell, W. E. McUmber, and B. H. Cheng. Enabling automated analysis of embedded systems designs via formalization of UML. Technical Report MSU-CSE-00-22, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, October 2000. submitted to IEEE Transactions on Computers.

[4] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *CACM*, 18(8):453–457, 1975.

[5] B. P. Douglass. *Real–Time UML*. Addison–Wesley, 1998.

[6] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*, chapter 1. P T R Prentice Hall, 1994.

[7] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *CACM*, 39(1):95–146, January 1992.

[8] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1994.

[9] D. Harel. StateCharts: A Visual Formalism for Complex Systems. *Sci. Comput. Programming*, 8, 1987.

[10] C. Hoare. Communicating sequential processes. *Communications of the ACM*, (8):666–677, August 1978.

[11] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[12] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*. Kluwer, 1999.

[13] W. E. McUmber and B. H. Cheng. UML-based analysis of embedded systems using a mapping to VHDL. In *Proc. of IEEE High Assurance Software Engineering (HASE99)*, Washington, DC, November 1999.

[14] W. E. McUmber and B. H. Cheng. Using metamodels and homomorphisms to integrate informal and formal techniques. Technical Report MSU-CPS-99-10, Department of Computer Science, Michigan State University, East Lansing, Michigan, February 1999.

[15] Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951. *Object Constraint Language Specification*, 1.1 edition, September 1997.

[16] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. *Proceedings of the International Conference on Software Engineering (ICSE98)*, April 1998.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[19] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, chapter 13, pages 484–485. Addison-Wesley, 1999.

[20] M. Shroff and R. B. France. Towards a Formalization of UML Class Structures in Z. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 646–651. IEEE Comput. Soc, Los Alamitos, CA, USA, August 1997.

[21] E. Y. Wang and B. H. C. Cheng. A rigorous object-oriented design process. In *Proc. of International Conference on Software Process*, Naperville, Illinois, June 1998.

[22] E. Y. Wang and B. H. C. Cheng. Formalizing the functional model within object-oriented design. *International Journal of Software Engineering and Knowledge Engineering*, 10(1):5–30, 2000.

[23] E. Y. Wang, H. A. Richter, and B. H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *Proc. of IEEE International Conference on Software Engineering (ICSE97)*, Boston, MA, May 1997.