IDEA GROUP PUBLISHING

ITJ2302

701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

Formal Approaches to Systems Analysis Using UML: An Overview

JON WHITTLE, NASA Ames Research Center, USA

Formal methods, whereby a system is described and/or analyzed using precise mathematical techniques, is a well -established and yet, under-used approach for developing software systems. One of the reasons for this is that project deadlines often impose an unsatisfactory development strategy in which code is produced on an ad-hoc basis without proper thought about the requirements and design of the piece of software in mind. The result is a large, often poorly documented and un-modular monolith of code, which does not lend itself to formal analysis. Because of their complexity, formal methods work best when code is well structured, e.g., when they are applied at the modeling level. UML is a modeling language that is easily learned by system developers and, more importantly, an industry standard, which supports communication between the various project stakeholders. The increased popularity of UML provides a real opportunity for formal methods to be used on a daily basis within the software lifecycle. Unfortunately, the lack of preciseness of UML means that many formal techniques cannot be applied directly. If formal methods are to be given the place they deserve within UML, a more precise description of UML must be developed. This article surveys recent attempts to provide such a description, as well as techniques for analyzing UML models formally.

INTRODUCTION

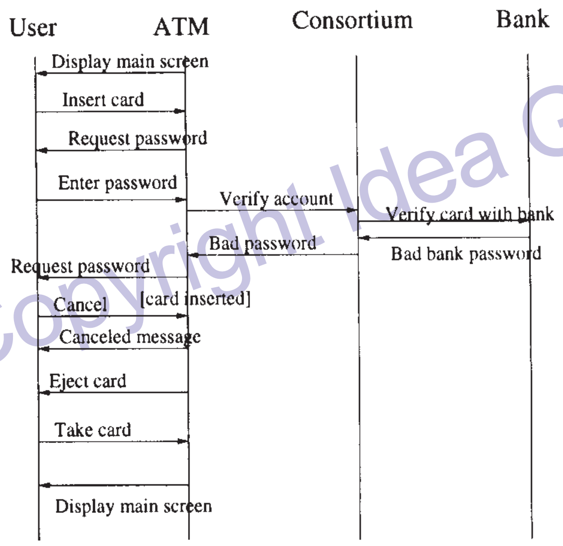
The Unified Modeling Language (UML) (Object Management Group, 1999; Booch, Jacobson, and Rumbaugh, 1998) provides a collection of standard notations for modeling almost any kind of computer artifact. UML supports a highly iterative, distributed software development process, in which each stage of the software lifecycle (e.g., requirements capture/analysis, initial and detailed design) can be specified using a combination of particular UML notations. The fact that UML is an industry standard promotes communication and understanding between different project stakeholders. When used within a commercial tool (e.g., Rhapsody (I-Logix Inc, 1999), Rational Rose (Rational Software Corporation, 1999)) that supports stub code generation from models, UML can alleviate many of the traditional problems with organizing a complex software development project. Although a powerful and flexible approach, there currently exist a number of gaps in the support provided by UML and commercial tools. First and foremost, the consistency checks provided by current tools are limited to very simple syntactic checks, such as consistency of naming across models. A greatly improved process would be obtained if tools were augmented with deeper semantic analyses of UML models. Unfortunately, although many of these techniques already exist, having been developed under the banner of Formal Methods, they cannot be applied directly to UML. UML is, in fact, grossly imprecise. There is as yet no standard formal semantics for any part of UML, and this makes the development of semantic tool support an onerous task.

This article gives an overview of current attempts to provide an additional degree of formality to UML and also of attempts to apply existing Formal Methods analyses to UML models. Space prevents the presentation of too much detail, so the description is at a more introductory level. Our starting point is the UML definition document itself (Object Management Group, 1999) which actually includes a section on UML semantics. Unfortunately, this semantics is by no means formal but essentially provides merely a collection of rules or English text describing a subset of the necessary semantics.

To motivate the need for a formal semantics of UML, consider Figure 1, which gives a simple *sequence diagram* describing a trace in an automated teller machine (ATM). Sequence diagrams, derived in part from their close neighbor Message Sequence Charts (MSCs) (ITU-T, 1996), are a way of visualizing interactions (in the form of message sending and

Manuscript originally submitted March 1, 2000; Revised May 26, 2000; Accepted June 15, 2000 for publication.

Figure 1: Interaction with an ATM



receipt) between different objects in the system. They can be used either in the early stages of development as a way of expressing requirements, or in later stages as a way of visualizing traces (e.g., error traces) of a design or actual code. It is in their use as requirements specifications that their inherent ambiguities are most worrying. Misunderstandings at this stage of development can easily lead to costly errors in code that might not be discovered until final testing.

The simplicity of sequence diagrams makes them suitable for expressing requirements as they can be easily understood by customers, requirements engineers and software developers alike. Unfortunately, the lack of semantic content in sequence diagrams makes them ambiguous and therefore difficult to interpret. For example, is the intended semantics of Figure 1 that the interaction *may* take place within the system, or that it *must* take place? UML provides no standard interpretation. Expressions in square brackets denote conditions, but it is not clear from the Standard whether a condition should only apply to the next message or to all subsequent messages - in the figure, does the [card inserted] condition apply only to the 'Cancel' message, or to all following messages? It is these kinds of ambiguities that could lead to costly software errors. In practice, each stakeholder would probably impose his or her own (possibly ambiguous in itself) interpretation of the sequence diagrams. When these are passed to a design modeler, the designer may introduce yet another interpretation. The result is a loss of traceability between software phases and a high degree of unintended behaviors in the final system.

Further problems with interpretation occur when integrating multiple diagrams. In the case of integrating a collection of sequence diagrams, it is not specified, for example, whether two sequence diagrams can be interleaved or must form completely separate interactions. This problem is experienced more generally when combining diagrams of different types (e.g., sequence diagrams and state diagrams).

UML Semantics

The Standard Semantics

Version 1.3 of the UML Specification (Object Management Group, 1999) contains a section on UML Semantics, which is intended as a reference point for vendors developing tools for UML. The semantics is only semi-formal, however, and ultimately provides merely a set of guidelines for tool developers to follow in order to seek UML compliance. The abstract syntax of each modeling notation (class diagrams, use case diagrams, sequence diagrams, state diagrams, activity diagrams, etc.) is given as a model in a subset of UML consisting of a UML diagram with textual annotations. The static semantics of each notation is given as a set of well-formedness rules expressed in the Object Constraint Language (OCL) (Warmer and Kleppe, 1999) and English. The use of OCL gives a misleading air of formality to this part of the semantics. OCL is a side-effect free, declarative language for expressing constraints, in the form of invariant conditions that must always hold. Constraints can be placed on classes and types in a model, used as pre- and post-conditions on methods, and be used to specify guards. However, as we shall see, OCL does not have complete formal rigor, and hence its use to express well-formedness rules is a little unfortunate. The dynamic semantics in the UML Specification is given as English text.

The official semantics is based on the four-layer architecture of UML, consisting of layers: user objects, model, metamodel and meta-metamodel. The semantics is primarily concerned with the metamodel layer. Table 1 reproduces the four-layer architecture from the UML Specification.

Each layer is further divided into packages (Foundation, Core and Behavioral Elements). The Core package defines the semantics of basic concepts (e.g., classes, inheritance) and the Behavioral Elements package deals with other notations (e.g., sequence diagrams, state diagrams). Each feature of UML is introduced with an English language description followed by its well-formedness rules as OCL constraints and its dynamic semantics as English text. To illustrate the inadequacy of the semantics, consider the specification for sequence diagrams. Sequence diagrams are a particular type of *collaboration* (collaboration diagrams are another type). Collaborations are stated to consist of a number of interactions, where an interaction specifies the communication between instances performing a specific task. Interactions in turn consist of messages sent between instances, where each message has, amongst other things, an activator (a message which invokes the behavior causing the dispatching of the message) and a set of predecessors, or messages which must have been completed before the current message can be executed. The use of predecessors could be utilized for resolving ambiguities—the predecessors give us a partial ordering on the messages, akin to that which appears in a

Table 1: UML Four Layer Architecture

Layer	Description	Example
Meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	*MetaClass, MetaAttribute, MetaOperation*
Metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
Model	An instance of a metamodel. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*
User objects (user data)	An instance of a model. Defines a specific information domain.	*<Acme_SW_Share_98789>, 654.66, sell_limit_order, <Stock_Quote_Svr_32123>*

sequence diagram. The well-formedness rules in fact give additional details about predecessors, such as "A message cannot be the predecessor of itself ", expressed in OCL as:

not self.allPredecessors-> includes (self)

and "The predecessors must have the same activator as the Message":

self.allPredecessors-> forAll (p | p.activator = self.activator)

However, the semantics still says nothing about how to combine multiple sequence diagrams, or about the existential or universal nature of a sequence diagram. As such, the Standard gives a framework in which to express semantic constraints, but is incomplete.

Formalizing the Semantics

There is a large body of research on formalizing object-oriented systems and graphical notations associated with them. Most of this work has concentrated on the static aspects of the system (e.g., class diagrams). A number of different formalisms exist for the basic concepts of class diagrams, e.g., (Bourdeau and Cheng, 1995; Overgaard, 1998; Wieringa, 1998). A full explanation of these is beyond the scope of this paper. Less work has been done on modeling the semantics of the dynamic aspects of systems (i.e., sequence diagrams, state diagrams etc.) but some research is appearing (e.g., Jackson, 1999; Overgaard, 1999; Araujo, 1998).

The Precise UML Group

A group of researchers, affiliated with the UML Standardization effort, have recognized the impreciseness of the official UML semantics and have formed the Precise UML Group (pUML) (pUML, 2000) to investigate ways of developing UML as a formal modeling language. The approach of the pUML is to define a core semantics for essential concepts of the language and then to define the rest of the semantics, in a denotational manner, by mapping into this core semantics. For example, the meaning of a sequence diagram might be understandable in terms of a subset of the behavioral semantics of the core. This approach is consistent with the standard semantics of the metamodel, and indeed, the intention is to build on the current informal semantics, rather than integrating many varied and disparate semantics. Whilst this is a noble effort, it is inevitably a huge undertaking, and so thus far, only small portions of UML have been worked on (e.g., see (Evans, France, and Lano, 1999) for a discussion of semantics for generalization and packages).

To give a small flavor of the work of the pUML, consider part of the task of making generalization precise, taken from (Evans, France, and Lano, 1999) and (Bruel and France, 1998). In the UML document, the more generic term for a class is a classifier. The metamodel fragment for relating a classifier to its instances is given in Figure 2. This is given along with English language and OCL annotations. The pUML addition to the semantics essentially adds further OCL constraints corresponding to the English text.

Page 2-36 of Object Management Group (1999) states that a set of generalizations is disjoint in the sense that an instance of the parent classifier may be an instance of no more than one of the children. This is formalized by pUML with the following OCL expression:

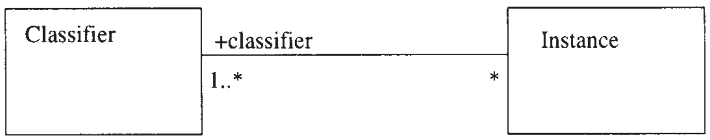
```
context c : Class inv
    c.specialization.child ->
        forall(i : Classifier | forall j : Classifier |
        i <> j implies i.instance ->
            intersection(j.instance) -> isEmpty)
```

which says that for any two children, i and j, of a class, the set of instances of i and j are disjoint.

Other Relevant Semantic Efforts

The pUML work has so far concentrated on basic concepts like objects and inheritance which unfortunately do not deal with how sequence diagrams, state diagrams etc., are

Figure 2: Metamodel fragment for Classifier and Instance relationship.



meant to be interpreted. Formal semantics already exist for variants of some of the UML notations. State diagrams are a typical example. State diagrams are derived from Harel's statecharts (Harel, 1987) and a large number of semantics have been developed for various versions of statecharts (see von der Beek, 1994) for a comparison of many of these. None of these can be directly applied to UML state diagrams. Sequence diagrams are another good example. They are derived from (amongst other things) the Message Sequent Chart (MSC) notation (ITU-T, 1996a) for which a formal standard semantics based on process algebras already exists (ITU-T, 1996b). As Damm and Harel (1999) points out, however, this semantics still leaves open a number of fundamental questions. In an attempt to overcome this, Harel has produced a formal semantics for his own version of MSCs (Damm and Harel, 1999) . He defines Live Sequence Charts (LSCs) as a conservative extension of MSCs, the main addition being the ability to specify a *temperature* (hot or cold) on both the vertical timelines in the chart and on the messages. Hot timelines indicate interactions that *must* take place, whereas cold timelines indicate possible interactions. Similarly, hot messages must be sent (or be received) whereas cold ones *might* be sent/received. The motivation behind this, is that developers actually use MSCs differently depending on the stage of development. Early in the software lifecycle, the designer is more interested in possible behaviors whereas, later on, the emphasis is usually on specifying more precisely exactly what behaviors will occur and when. Another notation similar to UML sequence diagrams is Timed Sequence Diagrams, a formal semantics of which, based on trace theory, is presented in Facchi (1995) .

It remains an open question if it is possible to integrate existing semantics such as these with the metamodel semantics pursued by the pUML group.

Object Constraint Language

The Object Constraint Language (OCL) is a side-effect free, declarative, constraint language included in UML both to allow users to express additional constraints that could not be expressed in the existing notations, but also to facilitate specification in a formal, yet comprehensible way. It was designed to bring the advantages of formality and yet be readable enough for use by software developers. The specification of OCL itself, however, is largely given by an English language description which is often ambiguous or inconsistent. As an example of this, consider the following OCL expression (taken from Richters and Gogolla, 1998):

```
context rs : RentalStation inv
    rs.employee->iterate(p: Person;
        names : String = "" | names.concat(p.lastname))
```

This query is intended to build a list of names of employees of a car rental station (Object Management Group (1999) prescribes no statement about the order of evaluation if the structure being iterated over is a set or a bag, hence evaluations may yield different results caused by different iteration sequences.

Richters and Gogolla (1998) presents a set-based semantics of OCL (not currently accepted as a UML standard) and overcomes the problem above by defining a deterministic evaluation semantics. For iteration over sequences, Richters and Gogolla take the elements in the order of the sequence (as in the UML Standard). For sets or bags, iteration must satisfy a precondition, which states that the operation which combines expressions at each stage of the iteration (in this case, concat) is associative and commutative. If this precondition fails, then the result is dependent on the order of execution and so the set or bag should be converted to a sequence first.

Vaziri and Jackson (1999) point out a number of shortcomings of OCL. It has been shown (Mandel and Cengarle, 1999) that OCL is, in terms of its expressiveness, not equivalent to the relational calculus (an example is given of an operation that cannot be encoded in the relational calculus but can be expressed in OCL).

Tool Support

Despite the current lack of a precise semantics, there have been a number of attempts to provide sophisticated tool support for UML. In this paper, we concentrate on those approaches that are already well established in the Formal Methods community.

The Different Guises of Model Checking

The term *model checking* has been used in a variety of different contexts with different meanings. This ranges from its use as meaning the checking of simple syntactic constraints (e.g., in the Rhapsody tool, which offers this capability) to its use in the Formal Methods community (McMillan, 1993; Holzmann, 1997) to describe a technique for exhaustively searching through all possible execution paths of a system, usually with the intention of detecting a system state that does not satisfy an invariant property, expressed in temporal logic.

Model checking, in the latter sense, is now a well-established technique for validating hardware systems, and as such, has been incorporated into commercial CASE tools. However, the use of model checking in the software world is more problematic. Software systems generally tend to be more complex than hardware, containing more intricate data structures that may involve hierarchy, making them structured differently from hardware systems. In addition, software is often specified with infinite state descriptions, whereas model checking can only be applied to finite state systems (as is the case for hardware).

There are two main types of model checking. In both cases, the idea is to enumerate each possible state that a system can be in at any time, and then to check for each of these states that a temporal logic property holds. If the property does not hold, a counterexample can be generated and displayed to the user (this is done using MSCs in the Spin model checker (Holzmann, 1997)). Explicit model checking represents each state explicitly and exploits symmetries for increased performance. Symbolic model checking, on the other hand, uses an implicit representation in which a data structure is used to represent the transition relation as a boolean formula. In this way, *sets* of states can be examined together - those which satisfy particular boolean formulas. The key to the success of the method is the efficient representation of states as boolean formulas, and the efficient representation of these boolean formulas as Binary Decision Diagrams (BDDs).

Despite recent successes in model checking software, e.g., Havelund, Lowry, and Penix (1997) , all approaches fall foul of the so-called state space explosion. The sheer complexity of software systems leads to a state space that is too large (e.g. $> 2^{1000}$ states) to be explored exhaustively. This means that only very small systems can be analyzed automatically. There exist techniques for dealing with this problem, most notably the use of abstractions (Lowry and Subramaniam, 1998), which reduce the size of the model by replacing the model with a simpler but equivalent one. The drawback here is that coming up with useful abstractions is a highly creative, and hence non-automatable task. In addition, the equivalence of the abstracted and original system may need to be proved formally which generally requires an intimate knowledge of theorem-proving techniques. Another promising technique is to modify the model checking algorithms, developed for hardware, to take into account the structure of the software system. This idea has the potential to alleviate the state space explosion problem, but research in this area is very much in its infancy. Structure can be exploited either by compositional model checking (de Roever, Langmaack, and Pnueli, 1998) in which a complex system is broken down into modules, which are checked individually, or by hierarchical model checking (Alur and Yannakakis, 1998) in which the hierarchy in a system is used to direct the search.

Despite these drawbacks, it still seems that incorporating existing model checking techniques into UML would be worthwhile. Indeed, the highly structured nature of most UML models (yielded by the object-oriented style and also hierarchy in state diagrams) may mean that model checking is likely to be more successful when applied to UML than to other languages.

So far, there have been surprisingly few attempts to apply model checking to UML and all of these have concentrated on checking state diagrams. UML State diagrams are a variant of Harel's statecharts (Harel, 1987) which have always been an attractive target for model checking software because they are more abstract than code. Indeed, there has been a reasonably large body of work on model checking statecharts (Day, 1993; Mikk, Lakhnech, Siegel, and Holzmann, 1998), including case studies on large, real-world systems such as the TCAS II system (Traffic Alert and Collision Avoidance System II) (Chan, Anderson, Beame, and Notkin, 1998) , a complex airborne collision avoidance system used on most commercial aircraft in the United States. One issue that has always plagued researchers attempting to analyze statecharts is the choice of a semantics. Harel did produce an official semantics for statecharts, as incorporated in iLogix's CASE tool STATEMATE (I-Logix Inc, 1996), but this is not a compositional semantics and so does not support compositional checking of large specifications.

The semantics of UML state diagrams differ significantly from Harel's statecharts. This stems from the fact that STATEMATE statecharts are function-oriented, whereas UML state diagrams are object-oriented. As a result, both Paltor and Lilius (1999) and Harel and Grey (1997) give UML state diagrams a semantics which includes an event queue for each object and a run-to-completion algorithm which dispatches and executes events on an object's event queue until the top state generates a completion event and the state machine exits. This is in contrast to the STATEMATE semantics in which events reach their destinations instantly.

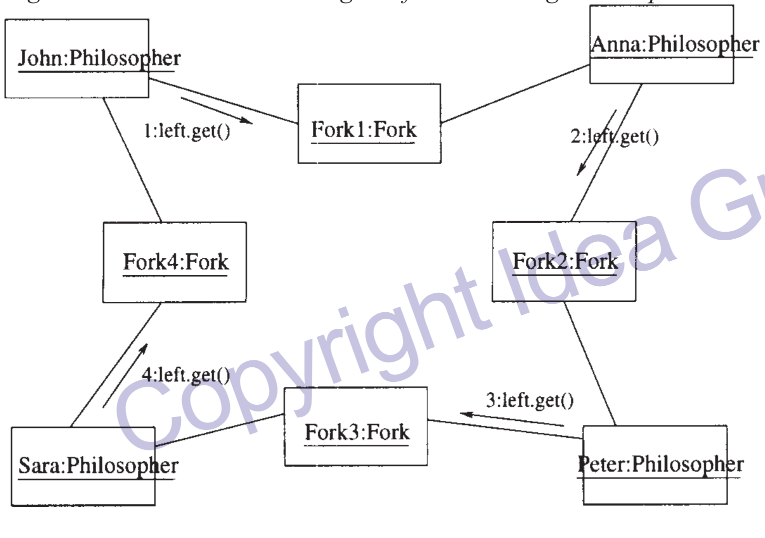
vUML applies model checking based on the semantics given in Paltor and Lilius (1999). The key idea of the vUML tool is that UML state diagrams and an associated collaboration diagram are translated into the PROMELA language which is the input language for the widely recognized SPIN model checker (Holzmann, 1997). A collaboration diagram (see Figure 3) is equivalent to a sequence diagram, but emphasizes the relationship between objects in an interaction rather than timing constraints between messages. The collaboration diagram is required to specify the instances of classes present in a particular execution. vUML is also capable of displaying counterexamples from the SPIN model checker as UML sequence diagrams. Note that SPIN itself generates counterexample traces as MSCs, but vUML then gives a sequence diagram in terms of the original UML objects and messages, not the internal PROMELA syntax used in the model checking. As yet, vUML has only been tested on a small number of examples, including the well-known Dining Philosophers problem and the verification of a Production Cell. Although the vUML semantics is not compositional, the authors claim that large models can be broken down into parts and model checked, by using a feature that non-deterministically generates events not specified in the current model.

JACK (Gnesi, Latella, and Massink, 1999) is a tool suite for various formal methods techniques that includes a model checker for UML state diagrams based on branching time temporal logic. The main difference between JACK and vUML is that hierarchy within state diagrams is treated as a first class citizen in JACK. This means that when translating into the model checker's input language, JACK preserves the hierarchical structure of the state diagrams, whereas vUML "flattens out" the diagrams. In principle, this preservation of structure is a necessary step if hierarchical model checking algorithms are going to be used for UML in the future. The structure preservation is done by first translating UML state diagrams into *hierarchical automata* (HA) (Mikk, Lakhnech, and Siegel, 1997) which are essentially statecharts but without inter-level transitions, i.e. transitions with source and target at different levels in the hierarchy. Inter-level transitions are akin to GOTO statements in imperative programming and as such, make formal analysis near impossible because they destroy modularity. The trick in translating to HAs is that each inter-level transition is "lifted" to the same level by annotating it with additional information regarding which sub-state of its target state it will enter upon firing. The statechart model checker MOCES (Mikk, Lakhnech, Siegel, and Holzmann, 1998) uses the same technique for model checking function-oriented statecharts.

Model Finding

Another technique, which is often also referred to as model checking, but which is more properly termed model *finding* is the enumeration of all possible assignments until a model is found (i.e., an assignment in which the given formula is true). A UML specification can be translated into a formula, and model finding can then be used either for simulation, in which models of the specification are presented to the user, or for detecting errors, in which case the model finder is applied to the negation of the formula and if a model is found, it is a counterexample. In this way, simulation is useful in the early stages of development --suggested models may be surprising and suggest necessary modifications. Model checking is useful in the later stages and can be used to find model counterexamples. Note that model finding does not carry out a temporal analysis as is the case for model checking in the previous section, nor does it compute reachable states. However, model finding can be an extremely useful technique for ironing out bugs in software specifications.

Figure 3: A Collaboration Diagram for the Dining Philosophers



Perhaps the most successful application of model finding so far is the Alcoa system which is based not on UML but on an alternative object modeling language called Alloy (Jackson, 1999). Alloy is not as extensive as UML but essentially consists of a language for expressing object diagrams very similar to class diagrams, and a relational language similar to OCL, but designed to overcome some of the drawbacks of OCL mentioned earlier. In particular, Alloy is intended to be similar in spirit to OCL and the relational specification language Z (Davies and Woodcock, 1996) but more amenable to automatic analysis.

Alcoa is a model finder for Alloy based on an earlier incarnation Nitpick (Jackson, 1996), but using different underlying technology. Whereas Nitpick looked for all possible assignments and attempted to reduce the number of possibilities by looking for symmetries, Alcoa translates an Alloy model into a huge Boolean formula. This Boolean formula can then be checked for satisfaction using standard SAT solvers. This translation is only possible because the user specifies a *scope* over which model finding is carried out. This scope states the maximum number of instances of each particular class that is allowed and reduces any Alloy model to a finite one. Relations can then be translated into a matrix of Boolean formulas. For each relational variable, a matrix of boolean variables is created. Each term is then translated by composing the translations of its sub-terms. As an example, if terms p and q are translated into matrices $[p]$ and $[q]$ respectively, then $[p]_{ij}$ is a boolean formula that is interpreted as being true when p maps the ith element of its domain type to the jth element of its range type. Similarly for $[q]_{ij}$. The translation of the term $p \cap q$ is given by

$$[p \cap q]_{ij} = [p]_{ij} \wedge [q]_{ij}$$

Note that all matrices are finite because the scope imposes finite limits on the number of instances.

Alcoa turns out to be reasonably successful at ironing out simple errors in the early stages of development. The author of the tool states that a handful of relations with a scope of size 3 can be handled easily. Any larger scope and the tool begins to fade. This limit of size 3 scope appears at first sight to be extremely limiting. However, the developers of Alcoa claim that most problems can be discovered using a very small scope, and that most errors present in specifications with a large number of instances are also present if the number of instances is restricted. Alcoa is still very much a research prototype and so further experimentation is needed to see if indeed this claim holds. Alcoa is in many senses similar to the FINDER model finding tool (Slaney, 1995), although it has been developed specifically with object models in mind.

Other Approaches

Model checking and model finding techniques have found greater acceptance both in general and within UML than have approaches based on theorem proving. Theorem provers are very general purpose machines for carrying out mathematical analysis. In principle, they can take a UML description as input and automatically or interactively be used to prove properties about the UML model. In practice, however, theorem proving is an extremely difficult task to automate and although there has been some success (Richardson, Smaill, and Green, 1998), most theorem proving is highly interactive, making it unsuitable for the average software engineer. Theorem proving can be successful in a very restricted domain. The Amphion system (Lowry, Philpot, Pressburger, and Underwood, 1994) uses theorem proving in a restricted setting (that of space trajectories) to automatically synthesize Fortran functions. These functions are extracted from a proof that a specification holds and can be synthesized using very little or no knowledge about theorem proving. It may be possible to identify restricted subtasks in analyzing UML models that theorem proving could be applied to in a similarly fruitful way. One possibility is to synthesize UML state diagrams from a collection of sequence diagrams annotated with OCL constraints. Certain OCL expressions correspond to a particular implementation as a statechart — for example, an OCL expression stating that the order of dispatch of a set of messages is irrelevant would be implemented as a statechart differently than an OCL expression enforcing a particular execution order of messages. Theorem proving could be used to refine a collection of OCL expressions into those that correspond to a particular statechart implementation. In this way, theorem proving would yield an intelligent synthesis of statecharts from sequence diagrams. First steps in this direction have already been taken (Whittle and Schumann, 2000).

Rewriting

A technique that is in many ways similar to theorem proving is rewriting, in which a rewriting engine applies rewrite rules to rewrite a term into an equivalent one. Very efficient rewriting engines now exist, e.g., the Maude system (Clavel et al., 1999) , and there are signs that such engines are being applied to UML development. Aleman and Alvarez (2000) presents a formalization of UML class diagrams in the OBJ3 specification language, a precursor to Maude. The aim of this work is to formalize the UML metamodel which would hence point out inconsistencies and ambiguities in the current semantics document. By using OBJ3's rewriting engine, UML models can also be checked to satisfy the well-formedness rules in the semantics document. The idea is to provide a set of rewrite rules that rewrite a formalization of a UML class diagram. If there are errors in the class diagram (in the sense that the well-formedness rules are violated), this will be shown by rewriting to a term involving a particular kind of exception.

Although the technique has so far only been used to check well-formedness rules, the framework allows class diagrams to be rewritten into equivalent, perhaps simpler ones. If a formalization of other UML notations is incorporated, it could enable a nice way of expressing transformations between different UML diagrams (such as between sequence diagrams and statecharts mentioned above).

Transformations on UML Diagrams

Some work on providing deductive transformations has been done by the pUML group members. The motivation here has been to support *transformational development* - the refinement of abstract models into more concrete models, using design steps which are known to be correct with respect to the semantics. This development methodology has for a long time been investigated within the context of developing correct implementations of safety-critical systems. The approach, presented in Lano (1998), is based on a semantics given in terms of theories in a Real-time Action Logic (RAL). An RAL theory has the form:

theory *name*
types *local type symbols*
attributes *time-varying data representing instance or class variables*
actions *actions which may affect the data such as operations, statechart transitions and methods*
axioms *logical properties and constraints between the theory elements*

A UML class C can then be represented as a theory:

theory Γ_C
types C
attributes $C : \Im(C)$
  self $: C \to C$
  att$_1 : C \to T_1$
  ...

actions **create**$_C$ $(c : C)\ \{C\}$
  kill$_C$ $(c : C)\ \{C\}$
  op$_1$ $(c : C, c : X_1) : Y_1$
  ...

axioms
$\forall\, c : C \quad \mathbf{self}(c) = c \ \wedge \ [\,\mathbf{create}_C(c)](c \in C)$
$[\,\mathbf{kill}_C(c)](c \notin C)$

$\Im$ is the "set of finite sets of", X is the set of existing instances of X. For an action symbol, α, and predicate P , $[\alpha]P$ is a predicate meaning "every execution of α establishes P on termination".

Other parts of the UML (e.g., associations) can be formalized similarly. The introduction of the RAL representation of UML means that transformations can be defined on UML and proved to be correct. Lano (1998) identifies three kinds of transformations:

- *Enhancement transformations* which extend a model with new model elements.
- *Reductive transformations* which reduce a UML model in the full language to a model in a sub-language of UML.
- *Refinement transformations* which support rewriting models in ways which lead from analysis to design and implementation.

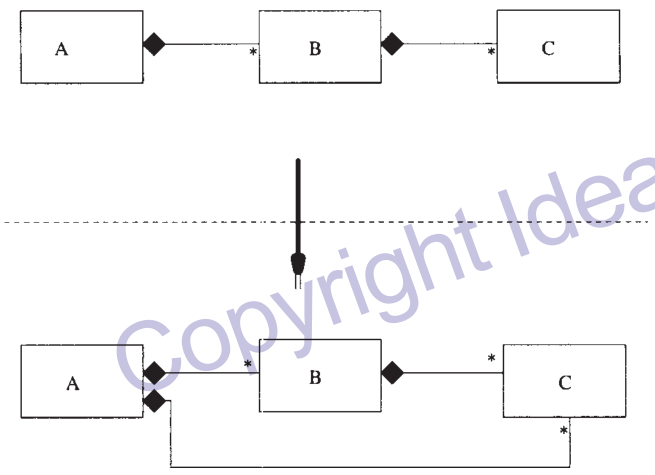
As an example, Figure 4 shows a refinement based on composition of compositions. The refinement adds an extra composition, in the case that *A*, *B* and *C* have no common objects. The approach to assuring correctness is that each UML component should satisfy certain properties, e.g., composition should be:

- one-many or one-one (a whole can have many parts, but a part cannot belong to different wholes at the same time);
- deletion propagating - deleting the whole deletes its parts;
- transitive;
- irreflexive.

For a particular transformation, such as that given in Figure 4, it can be proved using the RAL theories that the transformation preserves these properties.

This notion of correctness for transformations is very similar to the one used for assuring correctness of refactorings of C++ programs in Opdyke (1992). The transformations make only very minor changes to the model so that the proofs of correctness are usually trivial. In addition, the notion of correctness seems to be based on guaranteeing a set of predefined constraints, and hence cannot guarantee full behavioral correctness. UML does at least provide a framework for expressing such refactoring transformations. The properties to be checked can often be extracted from the UML specification. This is not the case for C++ refactorings, and, as a result, the refactorings in Opdyke are often overly constrained (so that their proofs remain trivial) to a point where they cannot be used in many real situations.

Figure 4: Composition of Compositions.



An interesting twist on this topic is presented in Evans (1998) in which the author proposes that properties of UML models (specifically class diagrams) be proven using diagrammatic transformations. The idea is that to check an invariant property of a model, the software developer expresses the property as *another class diagram*. Diagrammatic transformation rules are then used to transform the class diagram model into the class diagram property. This proves the property. All transformations are proved correct in a similar way to that in the previous paragraph, but using a set-based semantics expressed in the Z notation. The advantage of the approach is that no knowledge of complex logical, formal languages is required as all theorem proving is done diagrammatically. On the other hand, the class of properties that can be proved is limited because the properties must be expressible as a class diagram. Moreover, if the class diagram contains additional OCL constraints, the soundness of the transformation rules may no longer hold.

A final system worth mentioning is UMLAUT (Ho, Jezequel, Le Guennec, and Pennaneac'h, 1999) , a framework for implementing transformations on UML models, where transformations are based on pre-defined atomic transformations.

Conclusion

This paper has presented an overview of some of the attempts to formalize the semantics of UML and to apply techniques from Formal Methods to analyze UML models. Surprisingly, despite the plethora of analysis techniques in other formalisms such as statecharts (Harel, 1987) and Petri Nets (The World of Petri Nets, 2000), there has been relatively little transfer of these ideas to UML. The author believes that this is due to the largely informal nature of the UML semantics. As a result, more effort should be directed towards making the semantics precise. In addition, it is still worthwhile to investigate the application of formal techniques, even if the semantics is not fully decided. By taking a pragmatic approach, tools could support the well-understood part of the semantics and be designed in such a way that variations in the rest of the semantics could be quickly integrated. UML provides a great opportunity for practitioners in Formal Methods. As well as offering a huge potential user base, the UML can be seen as a test bed for different techniques. For instance, UML could provide a collection of standard examples to be used as case studies for the various techniques. The author believes that more effort should be invested in the application of Formal Methods to UML.

References

Aleman, J.L.F., & Alvarez, A.T. (2000). *Formal modeling and executability of the UML class diagram: a holistic approach.* University of Murcia, Spain: Software Engineering Research Group.

Alur, R., & Yannakakis, M. (1998). Model checking of hierarchical state machines. *6th ACM Symposium on the Foundations of Software Engineering*, (p.175-188).

Araujo, J. (1998). Formalizing sequence diagrams. In L. Andrade, A. Moreira, A. Deshpande & S. Kent (Eds.), *Proceedings of the OOPSLA98 Workshop on Formalizing UML. Why? How?*.

Booch, G., Jacobson, I., & Rumbaugh, J. (1998). *The Unified Modeling Language User Guide.* : Addison Wesley.

Bourdeau, R., & Cheng, B. (1995). A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering, 21*(10), 799-821.

Bruel, J.M. & France, R.B. (1998). Transforming UML models to formal specifications. In *Proceedings of UML98 - Beyond the Notation*. : Springer-Verlag. LNCS 1618.

Chan, W., Anderson, R.J., Beame,P., & Notkin, D. (1998). Improving efficiency of symbolic model checking for state-based system requirements. *Proceedings of ISTA98: ACM SIGSOFT Symposium on Software Testing and Analysis*, (p.102-112).

Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., and Quesada, J. (1999). *Maude: Specification and programming in rewriting logic*. Computer Science Lab: SRI International.

Damm, W. and Harel, D. (1999). LSCs: Breathing life into Message Sequence Charts. *3rd International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS99)*, (p. 293-312).

Davies, J., & Woodcock, J. (1996). *Using Z: Specification, refinement and proof*. : Prentice-Hall.

Day, N. (1993). *A model checker for statecharts* (Tech. rep. 93-35). Dept of Computer Science: University of British Columbia.

de Roever, W.P., Langmaack, H., & Pnueli, A. (1998). *Compositionality: the significant difference (COMPOS97)*. : Springer Verlag. LNCS 1536.

Evans, A.S. (1998). Reasoning with UML class diagrams. *Workshop on Industrial Strength Formal Methods (WIFT98), IEEE Press*.

Evans, A.S., France, R.B., Lano, K.C. & Rumpe, B. (1999). Metamodeling semantics of UML. In H. Kilov (Ed.), *Behavioral Specifications for Businesses and Systems* : Kluwer.

Facchi, C. (1995). *Formal semantics of timed sequence diagrams* (TUM-I9540). Munich: Technische Universitat Munchen.

Gnesi, S., Latella, D., & Massink, M. (1999). Model checking UML statechart diagrams using JACK. *4th IEEE International Symposium on High Assurance Systems Engineering*.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming, 8*, 231-274.

Harel, D., & Grey, E. (1997). Executable object modeling with statecharts. *IEEE Computer, 30*(7), 31-42.

Havelund, K., Lowry, M., & Penix, J. (1997). Analysis of a space craft controller using SPIN. *4th International SPIN Workshop*.

Ho, W.M., Jezequel, J., Le Guennec, A., & Pennaneac'h, F. (1999). UMLAUT: An extendible UML transformation framework. In *Proceedings of Automated Software Engineering Conference (ASE99)* (pp. 275-278). Cocoa Beach, FL: IEEE Press.

Holzmann, G.J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering, 23*(5), 270-295.

I-Logix Inc. (1996). *STATEMATE*. Burlington, MA.

I-Logix Inc. (1999). *Rhapsody*. Andover, MA.

ITU-T (1996a). *Message Sequence Chart (MSC), Formal description techniques* (Z.120).

ITU-T (1996b). *Message Sequence Chart semantics (MSC), Formal description techniques* (Z.120 Annex B).

Jackson, D. (1996). Nitpick: A checkable specification language. *Workshop on Formal Methods in Software Practice*.

Jackson, D. (1999). *Alloy: a lightweight object modelling notation*. MIT: http://sdg.lcs.mit.edu/~dnj/abstracts.html

Lano, K. (1998). Definining semantics for rigorous development in UML. In L. Andrade, A. Moreira, A. Deshpande, & S. Kent (Eds.), *Proceedings of OOPSLA98 Workshop on Formalizing UML. Why? How?*.

Lowry, M., & Subramaniam, M. (1998). Abstraction for analytic verification of concurrent software systems. *Symposium on Abstraction, Reformulation and Approximation.*

Lowry, M., Philpot, A., Pressburger, T., & Underwood, I. (1994). Amphion: Automatic programming for scientific subroutine libraries. In *Proc 8th International Symposium on Methodologies for Intelligent Systems*. Charlotte, North Carolina.

Mandel, L., & Cengarle, M.V. (1999). On the expressive power of OCL. In *FM99 - Formal methods. World congress on Formal methods in the Development of Computing Systems* (pp. 854-874). Toulouse: Springer. LNCS 1708.

McMillan, K. (1993). *Symbolic Model Checking*. Boston: Kluwer Academic.

Mikk, E., Lakhnech, Y., & Siegel, M. (1997). Hierarchical automata as a model for statecharts. *ASIAN 1997,* 181-196.

Mikk, E., Lakhnech, Y., Siegel, M., & Holzmann, G.J. (1998). Implementing statecharts in Promela/SPIN. *Proceedings of WIFT98.*

Object Management Group. (June 1999). *Unified Modeling Language Specification, version 1.3.* Available from Rational Software Corporation. Cupertino, CA.

Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks.* University of Illinois at Urbana-Champaign: PhD thesis.

Overgaard, G. (1998). A formal approach to relationships in the Unified Modeling Language. In M. Broy, D. Coleman, T.S.E. Maibaum & B. Rumpe (Eds.), *Proceedings PSMT98 Workshop on Precise Semantics for Modeling Techniques*. Munich: Technical Report TU Munich.

Overgaard, G. (1999). A formal approach to collaborations in the Unified Modeling Language. In R. France & B. Rumpe (Eds.), *UML99 - The Unified Modeling Language. Beyond the Standard. Second International Conference.* Fort Collins, CO: Springer.

Paltor, I., & Lilius, J. (1999). Formalising UML state machines for model checking. In R.B. France & B. Rumpe (Eds.), *UML99 - The Unified Modeling Language. Beyond the Standard. Second International Conference.* Fort Collins, CO: Springer. LNCS 1723.

Rational Software Corporation (1999). *Rational Rose* Cupertino, CA.

Richardson, J.D.C., Smaill, A., & Green, I.M. (1998). System description: Proof planning in higher-order logic with LambdaCLAM. In C. Kirchner & H. Kirchner (Eds.), *Proceedings of CADE-15* : Springer Verlag. LNCS 1421.

Richters, M. & Gogolla, M. (1998). On formalizing the UML object constraint language OCL. In T. Wang Ling, S. Ram, and M. Li Lee (Ed.), *Proceedings of 17th International Conference on Conceptual Modeling (ER98)* (pp. 449-464). Berlin: Springer. LNCS 1507.

Slaney, J. (1995). FINDER: Finite domain enumerator (system description). In A. Bundy (Ed.), *12th International Conference on Automated Deduction (CADE-12)* : Springer.

The Precise UML Group. (2000). Retrieved from the World Wide Web: http://www.cs.york.ac.uk/puml.

The World of Petri Nets. (2000). Retrieved from the World Wide Web: http://www.daimi.au.dk/PetriNets.

Vaziri, M. and Jackson, D. (1999). *Some shortcomings of OCL, the object constraint language of UML. Response to Object Management Group's RFI on UML 2.0* : OMG.

von der Beek, M. (1994). A comparison of statechart variants. In V. Langmaack & W. de Roever (Eds.), *Formal Techniques in Real-Time and Fault-Tolerant Systems*. New York: Springer Verlag.

Warmer, J.B., & Kleppe, A.G. (1999). *The Object Constraint Language: Precise Modeling with UML.* : Addison-Wesley.

Whittle, J., & Schumann, J. (2000). Generating statechart designs from scenarios. In *Proceedings of International Conference on Software Engineering (ICSE2000)*. Limerick, Ireland.

Wieringa, R. & Broersen, J. (1998). A minimal transition system semantics for lightweight class and behavior diagrams. In M. Broy, D.Coleman, T.S.E. Maibaum & B. Rumpe (Eds.), *Proceedings PSMT98 Workshop on Precise Semantics for Modeling Techniques*. Munich: Technical Report TU Munich.

Jon Whittle *received his B.A. degree in Mathematics from the University of Oxford in 1995. He obtained a Masters degree in artificial intelligence in 1996 and a PhD in formal methods in 1999, from the University of Edinburgh, Scotland. He now works in the Automated Software Engineering group at NASA Ames Research Center, Moffett Field, CA. His research interests lie in making lightweight Formal Methods practical and usable. His PhD concentrated on developing an editor for functional programs that incrementally carries out sophisticated forms of static analysis (e.g., termination analysis). Nowadays, he is involved in the UML formalization effort and has written papers on applying lightweight analysis to UML models (e.g., detecting conflicts between sequence diagrams). He is also involved in deductive program synthesis, in the context of automatically generating code for avionics applications using theorem proving.*