UML Formalization: A Position Paper

Kenneth Baclawski* kenb@ccs.neu.edu
Scott A. DeLoach† sdeloach@afit.af.mil
Mieczyslaw Kokar‡ kokar@coe.neu.edu
Jeffrey Smith§ jeffrey.e.smith@lmco.com

April 24, 2020

We take the position that the accessibility, usability and software engineering methodology advantages of modern CASE tools can be combined with the composability, consistency, verifiability and code generation advantages of formal methods. This can be accomplished by adapting the de facto standard CASE tool interface, the Unified Modeling Language (UML), as an open front-end to a formal methods system, Specware. The Specware system has an underlying mathematical model that supports software refinement, component model composability and specification-driven code generation. The integration of UML and Specware can be realized by formalizing UML semantics and automating the process of translating a UML repository into composable logical theories built into Specware. This integration could minimize the human effort needed to produce formalizations by generating them automatically using tools that are already in wide-spread use.

Although formal methods can provide a foundation for specification and modeling environments that is more complete, consistent and unambiguous than that produced by traditional or object-oriented methods, they are not commonly used for software engineering. As Anthony Hall [7] states,

> Formal methods are controversial. Their advocates claim that they can revolutionize development. Their detractors think they are impossibly difficult. Meanwhile, for most people, formal methods are so unfamiliar that it is difficult to judge the competing claims.

Advocates of formal methods admit that learning to use a formal methods system requires a high degree of skill in mathematical logic. Even if one has such a background, it requires at least six months of effort to become familiar with the specific logical framework and to become skilled in the user interface of even one system. Formalizing UML and constructing

*College of Computer Science, Northeastern University, Boston, Massachusetts 02115
†Air Force Institute of Technology, Wright-Patterson AFB, Ohio 43433
‡Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115
§Sanders, a Lockheed Martin Company, Nashua, New Hampshire

tools that can automate the transformation of UML diagrams to formal models would make it much easier to introduce and to use formal methods in a software engineering project.

The framework for formalizing UML proposed above emphasizes the construction of larger component models from smaller component models, maintaining both local and system constraints and properties. Composability of models is a fundamental mechanism of Specware, and this mechanism can be used as the basis for reuse of software specifications, as well as logical theories. Since UML unifies most modern software engineering development representations, e.g., object, state transition and collaboration diagrams, the same front-end can be used to model diverse system properties such as security, survivability, and real-time embedded software properties.

Although the UML is still under development, extensive documentation exists on its syntax and semantics [1, 2, 9, 10, 4]. Both the syntax and semantics of the UML are only partially formalized, and there is little consistency between different diagrams, so various CASE tools implement the UML in different ways. This situation inhibits tool interoperability. If some formalization of UML were accepted by the OMG, then it would reduce differences between interpretations of the UML semantics by vendors. This, in itself, would improve interoperability a great deal.

Some work has already been done to formalize the UML in [9] and [4], although this work is incomplete. In addition, a great deal of formalization has been done on the OMT language on which the UML is based, in part [3, 5, 6, 11]. Our approach to the formalization of UML is the theory-based object model developed by Scott DeLoach [5] for OMT and introduced in the next section below.

Theory-Based Object Model

In object-oriented systems, the object *class* defines the structure of an object and its response to external stimuli based its current state. In our theory-based object model, we capture the structure of a class with a theory presentation, or algebraic specification, in O-Slang, an object-oriented algebraic specification language. In these class specifications, sorts are used to describe collections of data values. The *class sort* is a distinguished sort that represents the set of all possible objects in the class. In an algebraic sense, this is actually the set of all possible abstract value representations of objects in the class.

Attributes, methods, and operations are defined as functions in O-Slang class specifications. *Attributes* are defined implicitly by functions that return data values while *methods* are functions that modify an object's attribute values. The semantics of functions, as well as invariants among class attribute values, are defined using first order predicate logic *axioms*.

Object instances are fundamental to any object-oriented model, and the theory-based object model captures the main uses of this notion by introducing state sorts and state attributes to model the (internal) state of an object, statecharts to define the transitions between object states, events to specify how objects can communicate with each other, and class sets to capture the notion of a set of objects in a class, such as the extent of a class.

To capture the notion of the internal state of an object, we introduce *state attributes* which are functions from the class sort to a *state sort* that return the current state of an

object. State attributes are distinct from normal attributes. An O-SLANG class specification has at least one *state attribute*. Multiple state attributes allow one to model concurrency and substates. A class specification may also have a set of *states* which are elements in a state sort (defined by nullary functions).

Communication between objects is handled by *events*, which are functions that may invoke methods, generate events for other objects, and directly modify state attributes. Events are distinct from methods to separate control from execution. Each class has a *new* event which triggers the *create* method to create a new object and initialize its attributes.

Operations are functions that do not modify attribute values and are generally used to compute derived attributes. Similar to methods and events, the semantics of operations are also defined using first order predicate logic axioms.

In order to manage a set of objects in a class, a *class set* is also created for each class defined. A class set is a class whose class sort is a set of objects from a previously defined object class. A class set includes *class event* definitions for each event in the original class. This class event is defined so that the reception of a class event by a class set object sends the corresponding event to each object in the class set.

Inheritance

Our theory-based object model uses a strict form of inheritance that allows a subclass object to be freely substituted for a superclass object in any situation as captured in the "substitution property" of Liskov [8]. The substitution property holds if there is a specification morphism from the superclass to the subclass and the subclass class sort is a sub-sort of the superclass class sort. In O-SLANG, this is usually done using the *import* operation, which includes the superclass specification directly into the subclass specification, and a statement that ensures the appropriate subsort relationship between the class sorts.

Multiple inheritance requires a slight modification to the notion of inheritance stated above. The set of superclasses must first be combined via a category theory colimit operation and then used to "inherit from". Importing the colimit specification and specifying that the class sort is a sub-sort of each of the superclass sorts ensures that the subclass inherits from each superclass and satisfies the substitution property.

Aggregation

Aggregation is a relationship between two classes where one class, the *aggregate*, represents an entire assembly and the other class, the *component*, is "part-of" the assembly. Not only do aggregate classes allow the modeling of systems from components, but they also provide a convenient context in which to define constraints and associations between components. Components of an aggregate class are modeled similarly to attributes of a class through the concept of *object-valued attributes*. An object-valued attribute is a class attribute whose sort type is a set of objects – the class-sort of another class. Formally, object-valued attributes are functions that take an object and return an external object or set of objects.

An aggregate class combines a number of classes via the colimit operation to specify a system or subsystem. The colimit operation also unifies sorts and functions defined in separate classes, associations, and events. To capture the entirety of a domain model within a single structure, one can define a domain-level aggregate by taking the colimit of all classes and associations within the domain.

Associations

Associations model the relationships between aggregate components. We define a *link* as a single connection between object instances and an *association* as a set of such links. A link defines what object classes may be related along with any link attributes or link functions. A link is basically a class specification that uses object-valued attributes to reference other objects while associations are represented as a class set of links.

Association *multiplicities* are defined as the number of links in which any given object may participate. These multiplicities are defined as constraints on the links in an association and can be captured axiomatically in the association specification.

Object Communication

In our theory-based object model, each object is aware of only a certain set of *events* that it generates or receives. From an object's perspective, these events are generated and broadcast to the entire system and received from the system. In this scheme, each event is defined in a separate event theory.

An *event theory* consists of a class sort, parameter sorts, and an event signature that are mapped via morphisms to sorts and events in the generating and receiving classes. If an event is being sent to a single object then the event theory class sort is mapped to the class sort of that object class. However, if the event theory class sort is mapped to the class sort of a *class set* then communication may occur with a set of objects of that class. The other sorts in an event theory class are the sorts of event parameters. The final part of an event theory, the event signature, is mapped to a compatible event signature in the receiving class. The colimit of the classes, the event theory is used to unify the event and sorts of two or more classes so that invocation of the event in the generating class corresponds an invocation of the actual event in the receiving class.

Communicating with objects from multiple classes requires the addition of another level of specification which "broadcasts" the communication event to all interested object classes. The class sort of a *broadcast theory* is called a broadcast sort and represents the object with which the sending object communicates. The broadcast theory then defines an object-valued attribute for each receiving class. Multiple receiver classes add a layer of specification; however, multiple sending classes are handled very simply. The only additional construct required is a morphism from each sending class to the event theory mapping the appropriate object-valued attribute in the sending class to the class sort of the event theory and the event signature in the sending class to the event signature in the event theory.

Conclusion

We have proposed a framework for introducing formal methods into the mainstream of software engineering. This framework would combine the advantages of CASE tools with the advantages of formal methods systems while minimizing the human effort involved in learning to use formal methods. Full development of this proposal would make it possible to use a CASE tool for tasks such as rigorous software refinement, component model composability, specification-driven code generation and reuse of software specifications.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *UML Notation Guide, Version 1.1*, September 1997.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *UML Semantics*, September 1997.

[3] R. Bourdeau and B. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

[4] R. Breu, U. Hinkel, et al. Towards a formalism of the Unified Modeling Language. In *ECOOP '97*, pages 344–366, 1997.

[5] S. DeLoach. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications*. PhD thesis, Air Force Institute of Technology, WL AFB, OH, June 1996. Ph.D. Dissertation.

[6] S. DeLoach, P. Baylor, and T. Hartnum. Representing object models as theories. Technical report, Department of Electrical and Computer Engineering, Air Force Institute of Technology, WL AFB, OH, 1997.

[7] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–20, September 1990.

[8] B. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5), May 1988.

[9] G. Övergaard. The semantics of the Unified Modeling Language. In *OOPSLA '96*, San Jose, CA, October 1996.

[10] J. Robbins, N. Medvidovic, D. Redmiles, and D. Rosenblum. Integrating architecture description languages with a standard design method. Technical report, University of CA, Irvine, 1997. White paper based on work sponsored by NSF grants CCR-9924846 and CCR-9701973 and DARPA, RF and USAF.

[11] E. Wang, H. Richter, and B. Cheng. Formalizing and integrating the dynamic model within OMT. In *IEEE International Conference on Software Engineering*, May 1997.