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
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).

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.

Figure 1: Interaction with an ATM

![Diagram of ATM interaction]

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 formalization 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