Chapter X

Abstracting UML Behavior Diagrams for Verification

María del Mar Gallardo, University of Málaga, Spain

Jesús Martínez, University of Málaga, Spain

Pedro Merino, University of Málaga, Spain

Ernesto Pimentel, University of Málaga, Spain

Abstract

UML (Unified Modeling Language) and XML (Extensible Markup Language) related technologies have matured, and at present many novel applications of both languages are frequently appearing. This chapter discusses the combined use of both UML and XML in the exciting application domain of software abstraction for verification. In particular, software development environments that use UML notations are now including verification capabilities based on state exploration. This method is effective for many realistic problems, although it is well known that it is affected by the state explosion problem for many complex systems and that some kind of abstraction is needed. This is the point where XML can be used as a powerful technology, due to its features for program transformation. We describe how to use XML-related standards like DOM or XMI in order to extend UML verification tools with automatic abstraction.
Introduction

The increasing demand of reliable software for critical control and communication systems is the reason why many CASE (Computer-Aided Software Engineering) tools include some kind of automatic verification facility. This is the case of academic- or research-oriented tools like SPIN (Holzmann, 2003) and commercial tools like SDL Suite (Telelogic, 2003). All these tools implement variants of model checking (Clarke, Emerson, & Sistla, 1986) as the verification method. Following this approach, the software being verified is described (modeled) with executable domain-specific languages like PROMELA or SDL, respectively. In addition, designers can specify desired or undesired properties using notations like temporal logic (Manna & Pnueli, 1992), automata (Vardi & Wolper, 1986) or Message Sequence Charts (MSC) (ITU-T, 2000). Model checking algorithms perform an exhaustive analysis of all the possible execution paths produced by the model to check whether the properties are satisfied.

Nowadays, UML has become the standard language to model complex software. It is being widely used in earlier stages of system design and, recently, UML-based CASE tools are trying to incorporate verification facilities like model checking. This has motivated the development of new research-oriented tools (VUML [Lilius & Porres, 1999] or HUGU [Schafer, Knapp, & Merz, 2001]), and commercial environments like STATEMATE (I-Logix, 2002) or Visual State (IAR, 2003). As verification requires an executable description of the software (the model), all these tools employ UML behavior diagrams (statecharts) as their input format. These diagrams represent the activation or deactivation of activities, data manipulation and event generation. It is worth noting that there exist different semantics for these diagrams, because original statecharts were introduced by Harel, Pnueli, Schmidt, and Sherman (1987) with a slightly different meaning from the one proposed in recent UML standard documents (Object Management Group, 2002b). Therefore, current verification tools for UML may use different semantics.

Although current verification tools have been effectively used to analyze real systems, they may fail for many complex systems due to the so-called state explosion problem. This problem is even more important in UML commercial environments, because they are oriented to developing final applications, considering richer descriptions of the software in order to allow automatic code generation.

Abstraction techniques are a good option to deal with the state explosion problem in model checking (Clarke, Grumberg, & Long, 1994; Dams, Gerth, & Grumberg, 1997). In this context, abstraction consists of replacing the model to be analyzed with another simpler description that produces a smaller state space and preserves enough information to decide about the satisfaction of the properties.

Abstraction can be implemented working with an internal representation of the model or by transforming the textual description. In previous works, we have defended the transformation approach as an effective choice to implement automatic techniques for abstraction in both academic (Gallardo & Merino, 1999; Gallardo, Martinez, Merino, & Pimentel, 2003) and commercial tools (Gallardo & Merino, 2000; Gallardo, Merino, & Pimentel, 2002). One major benefit of our approach is the possibility of completely reusing the model-checking tool, without modifying its internal code. In particular, we developed
Related Content

A Generic Framework for Defining Domain-Specific Models
[www.igi-global.com/chapter/generic-framework-defining-domain-specific/30535?camid=4v1a](www.igi-global.com/chapter/generic-framework-defining-domain-specific/30535?camid=4v1a)

Modeling and Programming by Commitment Rules in Agent Factory
[www.igi-global.com/chapter/modeling-programming-commitment-rules-agent/35868?camid=4v1a](www.igi-global.com/chapter/modeling-programming-commitment-rules-agent/35868?camid=4v1a)

Business Rules Management for Business Processes: From Modeling to Deployment
[www.igi-global.com/chapter/business-rules-management-business-processes/35874?camid=4v1a](www.igi-global.com/chapter/business-rules-management-business-processes/35874?camid=4v1a)

XML Benchmark
[www.igi-global.com/chapter/xml-benchmark/41500?camid=4v1a](www.igi-global.com/chapter/xml-benchmark/41500?camid=4v1a)