Specifications in the Development Process: An AsmL Demonstration

Mike Barnett Colin Campbell Wolfgang Grieskamp Yuri Gurevich
Lev Nachmanson Wolfram Schulte Nikolai Tillmann Margus Veanes

Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399
USA
{mbarnett,colin,wrwg,gurevich,levnach,schulte,nikolait,margus}@microsoft.com

ABSTRACT

AsmL is a specification system for software modeling, test generation, test validation, and implementation verification. It comprises a formal specification language, a set of libraries, and a test tool. It is integrated into the .NET Framework and Microsoft development tools. It has multiple source notations that can be used in a literate programming style either in an XML format or embedded within Microsoft Word. Specifications written in the system are executable, a novel feature which allows for semi-automatic test-case generation. In addition, the system can dynamically monitor an implementation to ensure that it conforms to its specification.

1. INTRODUCTION

There has been no lack of specification languages and systems for verifying properties of specifications and also for verifying that an implementation is correct with respect to its specification. What has been lacking (with several notable exceptions) are any results that have affected the "normal" programmer and the "normal" software development process. Our group, The Foundations of Software Engineering [4], is engaged in making specifications a part of the normal software development process at Microsoft. What we have found is that the area in most need of the kind of help specifications can provide is, perhaps surprisingly, testing and not development. Testers are often in the position of needing to understand the overall functionality of a system in order to design proper tests, yet this is often unavailable in any form other than the source code. A usable specification allows testing to begin earlier in the development process, concurrently with the coding effort. Currently, the natural language descriptions that serve as specifications suffer from ambiguity and imprecision. Finally, the most pressing need testers have, a *test oracle* can be provided by a specification.

We first describe the basics of AsmL specifications, then their use for test-case generation. A test-case consists of a sequence of calls to the modeled system, each call must be provided with a set of parameters. Then, when an implementation is available, the test-case can be applied to it and the results checked against its specification.

2. SPECIFICATIONS

AsmL is based upon the theory of Abstract State Machines (ASMs) [7, 8], which is a formal operational semantics for computational systems. A specification written in AsmL is an operational semantics expressed at an arbitrary level of abstraction. We call such an operational semantics a *model program*; we use the terms model and specification interchangably. AsmL incorporates the following features:

Nondeterminism AsmL provides a carefully chosen set of constructs with which one can express nondeterminism. They allow the specification of a range of behaviors within which the implementation must remain. Overspecification can be avoided without sacrificing precision.

Transactions AsmL is inherently parallel: all assignment statements are evaluated in the same state and all of the generated updates are committed in one atomic transaction. Updates that must be made sequentially are organized into *steps*; the updates in one step are visible in following steps and steps can also be organized hierarchically. Limiting the number of steps to prevent unnecessary sequentialization also helps prevent overspecification. The next state of the component is fully specified without making implementation-level decisions on how to effect the changes.

Additionally, AsmL provides a rich set of mathematical data types, such as sets, sequences, and maps (finite functions), along with advanced programming features such as pattern matching and several types of comprehensions.

It is also a full .NET language; AsmL models can interoperate with any other .NET component, e.g., written in

$C^\sharp$, VB, or C++. There are two source notations, a VB-like style in which white space is used to indicate scoping and which looks very similar to pseudo-code, and another style which is a superset of $C^\sharp$. Both notations can be used within a literate programming system; AsmL models are embedded in Word documents where they appear in special style-blocks. AsmL models can be also authored from Visual Studio .NET, where they are represented as XML documents in a particular schema. Bi-directional conversion between XML and Word format is supported. AsmL models can be compiled and directly executed from within Word or Visual Studio. AsmL specifications can be made at the interface level or for individual classes. More details on the use of AsmL specifications can be found in several papers [1, 6].

3. PARAMETER SELECTION

Our method for selecting test-case parameters is based on Korat [3]. The user adds annotations to the model to describe possible values for types and method parameters. From these annotations, the tool derives parameter sets. The annotations consist of values associated with parameters and fields of basic types, as well as a set of predicates that serve as filters on the cross-product of the combinations of parameters and define invariants for complex types. A technique called access driven filtering is used to enumerate the parameter sets in an efficient way that avoids generating redundant combinations.

4. SEQUENCE SELECTION

The tool exhaustively explores the reachable state space of the model, by executing methods with all associated parameters [5]. By necessity, the exploration must be pruned to some finite bound. We utilize a variety of cooperating techniques. One technique is a bound for the branch coverage. Another one can be considered as "state space" coverage, and is based on grouping the states (variable bindings) of the model into equivalence classes and bounding the number of representatives visited during exploration for each equivalence class. Finally, direct filtering of states can also be indicated; any state violating a filter is discarded and not considered as part of the reachable state space.

5. CONFORMANCE CHECKING

The use of AsmL specifications for conformance checking has been described elsewhere [2]. Conceptually, we run the specification and the implementation in parallel and check that the behavior of the latter is a possible behavior of the former. We track objects as they are created, returned from an implementation and then have their instance methods called. In this regard, the specification functions as a test oracle; this increases the efficacy of testing. Arbitrary conformance relations can be specified between the state of the model and the state of the implementation to allow a finer-grain checking than just comparing return values.

The conformance checking works both for test sequences derived from the model and those that are externally supplied. The implementation is instrumented at the IL level (the platform-independent language of the .NET virtual machine); this allows any .NET implementation, irrespective of its source language, to be checked relative to an AsmL specification.

6. CONCLUSION

One crucial decision made in the development of AsmL was to not limit its expressiveness. This has the consequence that general AsmL models are not directly amenable to static verification, such as model checking. However, we are investigating several methods for enforcing restrictions to enable at least some static verification.

We also are continuing to refine the source notations and the test tool itself in close collaboration with several product groups. We strongly believe that the timing is right for formal specifications to become an integral part of industrial software development.

Acknowledgements

This work would not have been possible without the efforts of visiting researchers and interns that have spent time in the Foundations of Software Engineering group at Microsoft Research. We also are indebted to the product groups that have worked with us and provided valuable feedback.

7. REFERENCES

[1] M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, Nov. 2001.

[2] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. *Software Engineering Notes*, 27(4), 2002.

[4] Foundations of Software Engineering, Microsoft Research, 2003. `http://research.microsoft.com/fse`.

[5] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. *Software Engineering Notes*, 27(4):112–122, 2002. From the conference International Symposium on Software Testing and Analysis (ISSTA) 2002.

[6] W. Grieskamp, M. Lepper, W. Schulte, and N. Tillmann. Testable use cases in the abstract state machine language. In *Asia-Pacific Conference on Quality Software (APAQS'01)*, Dec. 2001.

[7] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[8] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.