Specification and Testing at Microsoft:   
Addressing new challenges in the development process

**Yuri Gurevich**

**Foundations of Software Engineering (FSE)**

**Microsoft Research**

**August 2002**

Executive Summary

Microsoft has been amazingly successful in shipping complex software within tight time constraints. But the time-tested development approach faces the challenges of large-scale, trustworthy, distributed computing. We describe some of the problems and argue that software modeling can and should play a greater role in solving the problems. We propose the AsmL language, developed by the FSE group at Microsoft Research, for writing software models:

* Produce designs that are themselves testable.
* Test and enforce the design by means of AsmL-based tools.

In addition to making software more reliable, the new approach makes the work of testers more interesting and satisfying. It involves them early in the development process, allows them to understand the intended functionality of software, and helps to automate some of the more tedious tasks.

Then we discuss who should write models, in particular within the PM-Dev-Test triangle. Currently, the PM writes the spec. But writing software models is in many ways like high-level coding. Not every PM wants to do that and has the necessary skills.

Finally we discuss how model-based development can be rolled out more broadly in Microsoft. We recommend an AsmL tools-and-training group responsible for the rollout. It would be helpful if executive management becomes aware of the results of the AsmL pilot projects and encourages teams to include the new techniques, where appropriate, into their own processes.

# Some Problems

Microsoft's successful development approach is based on small and relatively independent feature teams. The teams check their source code into a source repository, and a centralized build lab assembles the system. The resulting executable image is then distributed back to the teams to test their components[[1]](#footnote-1). As bugs are detected they are triaged in order of severity. When the flow of newly discovered bugs declines, and all known bugs of high severity have been corrected, the system is ready to ship[[2]](#footnote-2).

Daily builds and the readiness to make the hard decisions presented by bug triage have enabled Microsoft to ship feature-rich products that have gone on to dominate their categories. However, to realize the .Net vision, the process needs to be adapted to the new environment of large, distributed, correctness-critical systems with many external dependences. The related problems are well known. Here are some of them.

## The chicken-and-egg problem of stabilizing the build and testing the components

It has traditionally been hard to properly test the components without a stable build. And it is definitely hard to produce a stable build from untested components. As the number of components grows, it becomes increasingly unlikely that untested components can be assembled into a system that functions well enough to perform system tests. (I understand that the experience of Visual Studio .Net is an example of this phenomenon. This large team had difficulty in reliably creating stable builds of their system. Even when the changes passed the check-in tests, they often interacted with other changes to cause instability in the system. In some cases it took weeks to stabilize the build to the point that testing could begin.)

## Too many configurations to test

Testing of all possible combinations of component versions, platforms and settings may be impossible even if the system is self-contained and runs locally. The problem is exacerbated in the case of distributed systems with numerous external dependences. In spite of extensive testing, software may contain undetected severe bugs, in particular security holes. And triage may give us a false sense of security. A bug may seem innocuous because it doesn't block any user scenario in the tested configurations. However, there are many untested configurations.

## Design bugs deserve more attention

When triaging bugs by severity, one tends to prioritize bugs in terms of their effect on the user scenarios at hand and to see design bugs as just another category of bug. It is presumed sometimes that every bug can be fixed, either now or in the future, but this does not necessarily apply to design bugs. In the past, we viewed software as infinitely mutable, in contrast to hardware. But today, the scale of our software systems makes them less mutable. Unlike a code defect whose resolution tends to be localized, a design flaw may require system-wide changes. In order to fix a design bug, one may need to change the whole design. This has unpredictable consequences and may be too expensive.

## Death by a thousand cuts

The ready-to-ship software may include many minor defects that, taken together, lower the user’s satisfaction, even if none of these defects individually is a showstopper.

# Software models can help

Improving specification and specification-based testing is crucial in solving the problems described above. If you test components with respect to a rigorous, well-understood standard of behavior, then assembling them into a larger system is easier. This kind of testing may reduce the set of configurations to test; if the component's behavior is completely defined by the spec, we shouldn't care about the version of the component. If the spec is executable, the design can be tested and debugged even before coding begins. And if you have a detailed and precise knowledge of the observable states and transitions, then the number of bugs within each component can be dramatically reduced. The overall level of system correctness increases accordingly.

**Remark**: While we are confident that our methods scale to large systems, this discussion is restricted to the level of components and systems of limited size. The reason is that we have not yet fully modeled systems of very large scale, certainly not the scale of Windows, Office or Visual Studio.

## There is a semantic hole in today's specs

Currently, specifications are written in English and include several kinds of information:

* A description of customer scenarios that must be covered
* A requirements-oriented description of features
* Constraints upon the implementation (for example, backward compatibility requirements)
* Syntax of the interface (for example, the signatures of API-level methods)
* Operational semantics that prescribes the behavior of a component

Natural language, as a means of communication, is better suited to some of these specification tasks than to others. The area where natural language fails us most is operational semantics. It is hard to keep a natural-language spec precise, and it is hard to maintain a fixed level of abstraction. As a result, we see specs that are vague, incomplete and sometimes even inconsistent when it comes to describing the behavior of a component. This is not necessarily a failing of the program manager; it may be a reflection of the difficulty of using natural language to describe the structure and transitions of a complex system.

## Precise operational specification is possible

The FSE group in Microsoft Research has developed the AsmL language for writing executable specifications. The group is also developing tools for deriving test cases from a spec and tools for runtime verification. AsmL allows the developers to:

**Write an executable spec.** AsmL lets you precisely describe the component's operational semantics and produce designs that are themselves testable. The AsmL model makes explicit all possible states and transitions that may occur at the chosen level of detail. The model itself is executable software.

**Test the spec.** AsmL-based tools let you test the specification — even prior to implementation — using interactive simulation or by means of analysis, either human or machine-assisted. This helps you eliminate designs that allow a system to become wedged.

**Enforce the spec.** AsmL-based tools help you enforce a correspondence between the designer’s understanding of the component’s high-level state (including its evolution) and the implementer’s interpretation of the component. A test harness compares the results of operations with the predictions made by the model and flags any operation of the implementation where the resulting system state contradicts the range of possibilities encoded in the model. This testing technique includes checking pre- and post-conditions but is more powerful: it takes into consideration the history of the run and not just static assertions. It also checks the component’s interaction with other components and not just its internal state changes.

## The harness issue

Simulating a component on the basis of an executable specification requires some integration work. Abstracted models of external systems need to be constructed, and the scenarios need to be coded. This requires an understanding of the external environment but that is nothing new. Whether using AsmL or not, a specifier must understand the environment in which the component will operate and carefully describe the component's interaction with it. Modeling brings into focus the necessary information about the environment and facilitates a clearer understanding of it. And of course, one team’s model may be reused by clients of that team.

# AsmL

## A language for executable specifications

AsmL is a powerful specification language. And AsmL specifications are executable[[3]](#footnote-3)! AsmL lets you specify your design and play with it.

You can freely use mathematical data structures like sets, sequences, maps, sets of sequences, and so on. These are important for producing faithful models that are easy to understand.

AsmL is transaction based. From that point of view, it is somewhat similar to database query languages. Until a transaction is committed, the relevant state does not change. This allows massive parallelism within a transaction. And large steps (the transactions) make analysis easier.

AsmL is object oriented and component oriented. It is a member of the .Net family of languages and thus is interoperable with other .Net languages.

AsmL is integrated with Visual Studio .Net and with MS Word.

See more at <http://asml> .

## Theoretical basis

AsmL is based on the theory of abstract state machines (ASMs); see <http://www.eecs.umich.edu/gasm/>. The ASM thesis asserts that every computer system, at any fixed level of abstraction, can be step-for-step simulated by an appropriate ASM. The ASM community, including FSE, has provided substantial experimental evidence for the thesis. And parts of the thesis have been proved mathematically from first principles.

AsmL isn’t the first ASM-based tool, but it is by far the most advanced.

## Is AsmL technology ready for use?

The numerous AsmL pilot projects have been successful. In every case, we have found instances where specifications were ambiguous and unclear, and in some cases we have found serious issues, including security flaws, in the design. See the FSE website <http://msrweb/foundations/>. Here are three examples of ongoing projects.

* The Indigo group is building AsmL models of WS-Routing and WS-Policy. They are using the models as oracles to ensure that implementations conform to the specifications. They also plan to use the models to generate test cases. Testers are being trained to understand and modify the models and to incorporate them into their existing test harnesses.
* The X# group, a part of the Web Data team, is using AsmL to give the formal semantics of their language. Doing this has exposed design issues that might otherwise not have been discovered until much later in the development process.
* The XAF team, a part of Office, is replacing their current contract specification and verification portions of their test harness with AsmL.

Of course, there are plenty of things to do. The recently released AsmL2 is still being debugged. At the day of this writing (August 13, 2002), the tool for deriving test-cases from AsmL specs has not yet been released.

The AsmL technology is ready for much wider use, though not for immediate adoption by all parts of the company.

## Model-based testing at Microsoft

While our approach based on high-level models is unique, there already exist fruitful complementary approaches, such as the model-based testing work pioneered by Harry Robinson of the Server Management team and Jason Taylor of the Avalon team. The current model-based testing tool, Test Model Toolkit, uses finite-state machines and combinatorics to generate test cases. TMT was demoed at the Windows Test Architect review last December, and MSTE currently offers a course in finite-state modeling to support testers in using TMT. The successor to TMT is the Integrated Testing Environment (ITE) project; see <http://avalon/test/model/ite> .  FSE takes an active part in the ITE project; the appeal of an integrated testing environment is irresistible. AsmL is being integrated into ITE.

# Addressing the organizational issues

Let me concentrate on the PM-Dev-Test triangle. We wanted to make AsmL the communication language in the triangle: The PM writes an AsmL spec and involves the tester to verify it; the Dev implements the precise and complete spec; the tester knows the intended functionality of the software and thus is enabled to test the implementation for compliance with the spec. The current culture is different, and not everybody in the triangle would benefit equally from software modeling. Here is an admittedly simplified picture of the current situation. The PM writes a natural language spec. He isn’t and can’t be held accountable for the precision of operational semantics. The Dev writes the code and decides how to interpret the spec. But he isn't and can't be held accountable for being truly faithful to a spec that may be incomplete and obscure. The tester has a difficult time in discovering the true functionality of the product. It is the tester who stands to benefit most from the introduction of software modeling. And indeed the testers have been most receptive to our ideas.

## The modeling gap

Who should write software models in the PM-Dev-Test triangle?

The job of testers is to test specifications, not to write them. The job of developers is to implement specifications, not create them. It is natural that program managers write software models. After all, they are writing specifications currently. But writing software models is different. It is in many ways like (very high level) coding. Not every program manager wants to do this or has the necessary skills. Sometimes the combination of program manager and the test lead can be appropriate for writing a model; the PMs are concerned with the creative issues, and the testers are concerned with the viability issues.

It may be desirable also to develop a certain specialization in modeling, to have some number of modeling specialists, modelers. This approach would be especially helpful for large teams with many interacting components. Maybe they can have a modeler per architect; or maybe some architects would also be modelers. In any case, if we are ever to get precise specifications into the development process, there needs to be a way to train appropriate people to integrate model-based specification into the process over time.

## Skill set

Regardless of job title--whether program manager or architect or tester--a successful modeler needs the appropriate training. In terms of background and aptitude, we would look for the following qualities:

* The ability to think at different levels of abstraction and stay faithful to each level
* A knowledge of system design
* Familiarity with different programming paradigms (imperative and functional)

The ability to handle different levels of abstraction requires precise thinking and the ability to feel comfortable manipulating abstract symbols. (Programming languages are both a blessing and a curse in this respect. Some programmers take the level of abstraction provided by their programming language to be reality and find it difficult to think in terms of more abstract data.)

Modelers need to understand system design because writing a model isn't simply transcribing English text into AsmL. A modeler must understand how systems fit together and what needs to appear in the model that may not be explicit in the requirement specification.

AsmL models use both the imperative and functional approach. They update variables to keep track of state changes, and, within a single state transaction, they behave as functional models. Modelers should feel comfortable with both techniques.

### Training needs

Formal classes, offered through MSTE, may include:

* Introduction to software modeling, which would teach AsmL as well as show students how to write simple models
* Intermediate software modeling, where students would bring in their own projects and help each other write their models
* A class for testers on test case generation and incorporating models into existing test harnesses

# Recommendations

Suppose that we agree that AsmL models and testing add enough value in the development process so that model-based development should be rolled out more broadly in Microsoft. It is clear that this rollout will take time; some groups may be better positioned than others to take immediate advantage of the new tools. But how should we proceed?

## Form an AsmL tools-and-training group and make this group responsible for the AsmL rollout

It is important that we maintain separate efforts to research new technology and to apply the technology we already have. FSE will continue to focus on pushing the state of the art in the areas of model-based specification and testing. A new application team should focus on training as well as on supporting and promoting the technology we have already developed.

Where will the application team "live"? One possibility would be to place it near other internal development tools such as Product Studio. There is also a question how to create the team. One way to start might be to use as a nucleus the three-person Seattle company, Modeled Computation, that has been successfully working with us in many AsmL pilot projects. The principal, Colin Campbell, has the skills to build the team.

## Involve executive management

I realize how busy executive management is already. But it would be most helpful if executive management becomes aware of the results of the AsmL pilot projects and encourages teams to include the new techniques into their own processes. Eventually, the use of software models could be reflected, where appropriate, in review goals.

1. We use the terms component and feature as synonyms though one may want to distinguish between two kinds of features: components packaged in the .dll or .exe form and aspects of the system that are not independently packaged. [↑](#footnote-ref-1)
2. This process is used for alpha and beta tests as well as for the final release. [↑](#footnote-ref-2)
3. A software model is different from a prototype. A prototype is one particular implementation – a quick one – of a specification. A model is the standard to which all implementations must adhere. Unlike a prototype, a model's code is meant to be scrutinized by many different people to understand the design and to decide if it does what it should. [↑](#footnote-ref-3)