The Future of Black-Box Testing at Microsoft

Margus Veanes

Foundations of Software Engineering

Microsoft Research

Testing: Current Challenges

- Test is a huge cost of product development
- Test effectiveness and software quality hard to measure

- Incomplete, informal and changing specifications
- Downstream cost of bugs is enormous
- Lack of spec and implementation testing tools

- Integration testing across product groups
- Patching nightmare
- Versions exploding

- Growing need to test distributed and multithreaded applications
- Handling of nondeterminism is becoming more and more important

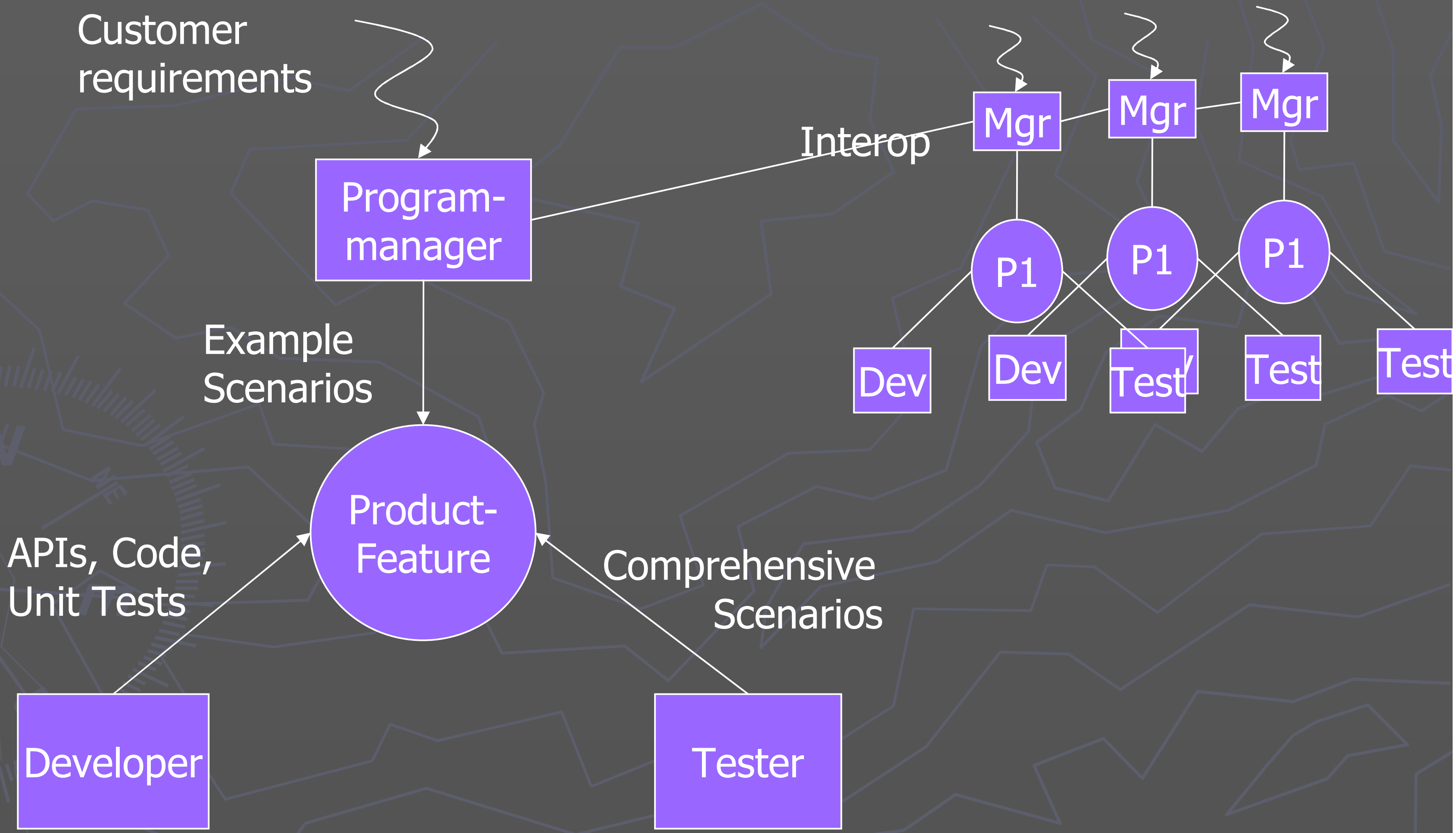
…

Testing: Current Practice

- Black-box testing – behavioral testing
  - Comes from a behavioral specification with no knowledge of implementation details
  - *Example*: scenario test

- White-box testing – structural testing
  - Based on local view of the implementation code
  - *Example*: unit test

Testing within Microsoft: Organizational Structure



Tester: How do I test this API?

Subtasks:

- What is the expected behavior of this API?
  - Build a model
- How are concrete tests created?
  - Traverse the model to create scenarios/test cases
- When does a test succeed or fail?
  - Use the model as an oracle –
    failure may be due to a model-error as well as an implementation error, by default assume the former
- When am I done testing?
  - Use code-coverage as well as model-based behavioral coverage
- What can I conclude when I'm done testing?
  - The model and the implementation agree wrt the test suite

Being in the Shoes of a Tester

Quiz: what is the expected behavior?

- ► Given an empty notepad document, do the following actions in the given order:
  1. Type ‘tere’
  2. Type Ctrl a (select all)
  3. Change font size to 26pt
  4. Type Ctrl z (undo)

- ► What happens after action 4?

A) Font change is undone,

B) Font change and selection are undone, or

C) None of the above!

Major change is pending

- Machine-readable specifications of several kinds will become “part of the build”
  - An extension of the idea of metadata
  - Support for both black-box and white-box views
  - Behavioral testing will move beyond the black box; contracts will move beyond white-box
- Support for behavioral verification will be built into the compiler and runtime environment

New Developments

- Spec# - Extension of C# with:
  - Contracts (pre- and post-conditions)
  - High-level data structures (sets, maps, …)
  - Nondeterministic choice
- Spec Explorer
  - Model-based testing of Spec# models
  - Conformance checking
- Spec# as front-end for static analysis tools (ongoing projects):
  - Static verification of contracts
  - Model checking
  - …

Contracts

- Contracts
  - An “extended type system”
  - “White-box” behavioral constraints using the vocabulary of the implementation
- Contracts with model variables
  - “Black-box” behavioral constraints, using model variables. (May be given for interfaces.)
- Executable contracts, or model programs
  - “Black-box” specifications of possible runs, using model variables.

Model-based testing

- Model: Any description of a system's behavior that precisely defines its possible states and transitions at a high level.
- Testing: Use model to
  - Generate tests by exploring all possible states and parameter combinations
  - Check conformance by comparing actual versus predicted behavior at run time.

Where contracts and models meet

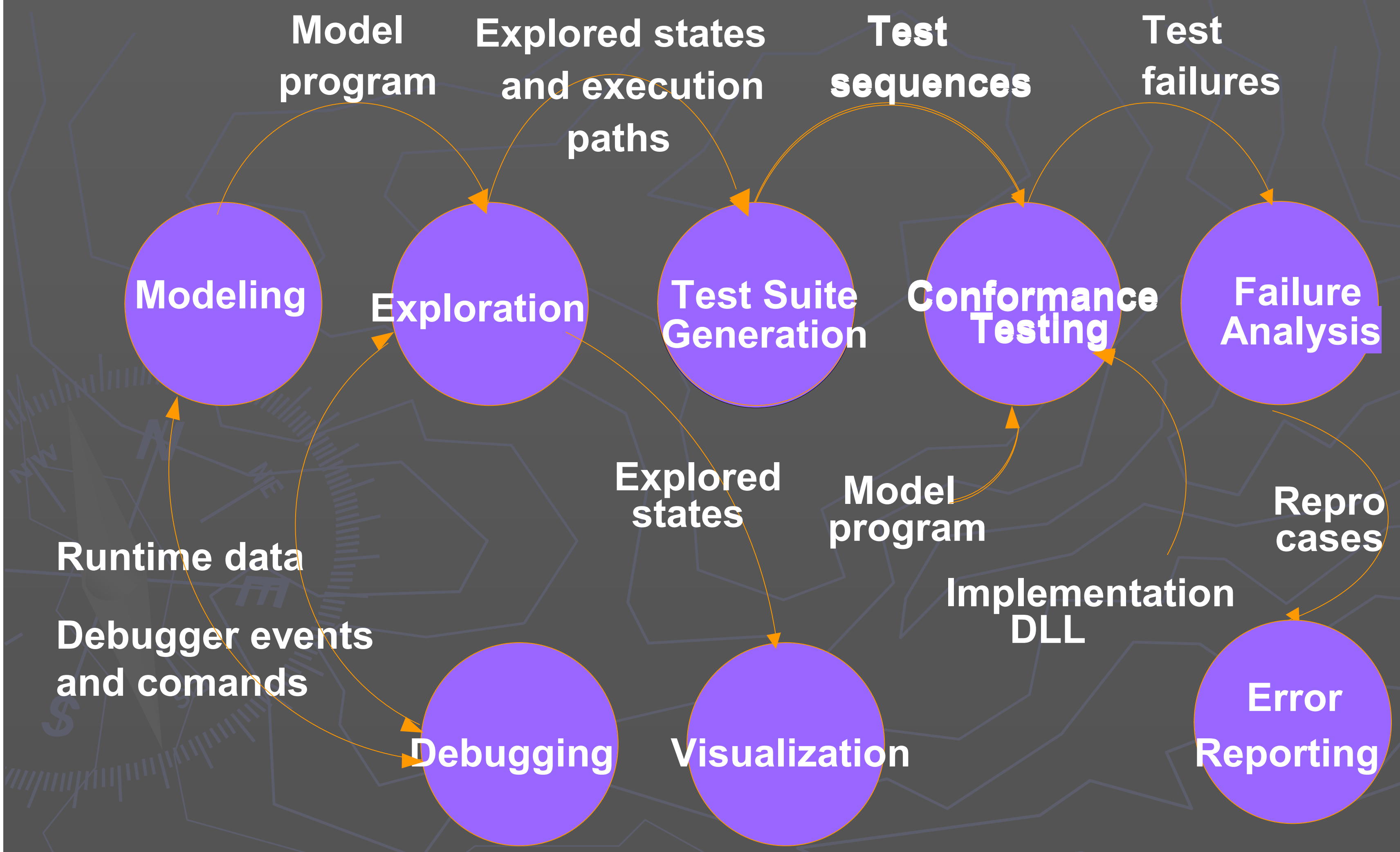
- Move toward more abstract contracts
  - Contracts will include model variables, or abstract state variables not used directly by the implementation
- Move toward model programs
  - Contracts will become complete enough to be executable

Abstraction (via model variables) allows us to speak about the behavior that all implementations must provide.

Benefits of Spec# specifications

- Easy to understand
  - A natural extension of contract-style specifications such as pre- and post-conditions, with a familiar C# syntax.
- Expressive
  - Capable of handling the full range of software artifacts, including method parameters and dynamic objects.
- Precise
  - Well-understood as a formal transition system
- Executable
  - Suitable for many kinds of analysis and what-if testing

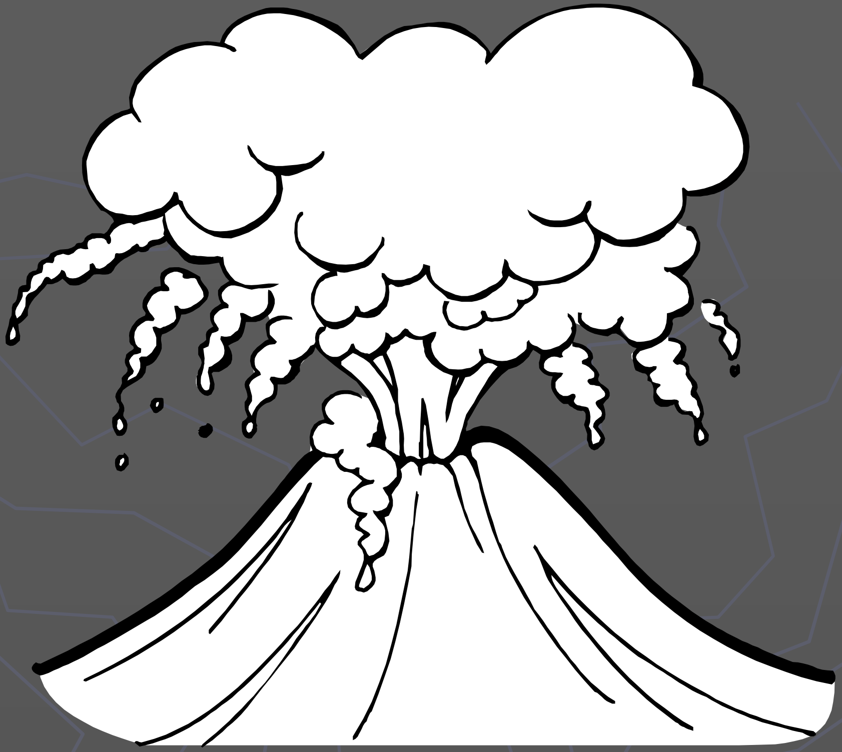
Spec# and Spec Explorer



Exploration using Spec#

- Goal
  - Express all possible runs as a finite-state machine
- Approach
  - At each state, execute any enabled method with any allowed argument values
    - "Enabled" means precondition is true
  - Collect results into a data set

Technical challenge



- **5 components, 100 states/component**
- **Worst case: 10 billion states**

Controlling the state explosion

- Focus on finding the optimal states and transitions instead of doing exhaustive search
- Use techniques like
  - Restricting search to a fixed number of objects
  - Identifying “similar” states and discarding them
  - Maintaining an efficient state representation
  - Using code coverage to guide exploration

Using the results of exploration

- Test sequence generation
  - Tests come from an intelligent traversal
- Conformance testing
  - We check actual versus expected behavior
- Error reporting
  - Tool exports test sequence to reproduce bug

Spec Explorer demo

Experience and Outlook

- Test Projects
  - Web services, Passport, Media player, Indigo Distributed File replication system
  - Models up to 100 pages
  - Growing user base
- Spec Explorer
  - New plug-in architecture supporting remoting
  - Support for model-checking
  - Support for static verification

Thanks!

- Spec# and Spec Explorer public release (summer 2004)

  http://research.microsoft.com/fse