Testing Concurrent Object-Oriented Systems with Spec Explorer

Extended Abstract

Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson,
Wolfram Schulte, Nikolai Tillmann, and Margus Veanes

Microsoft Research, Redmond, WA, USA

Abstract. We describe a practical model-based testing tool developed at Microsoft Research called Spec Explorer. Spec Explorer enables modeling and automatic testing of concurrent object-oriented systems. These systems take inputs as well as provide outputs in form of spontaneous reactions, where inputs and outputs can be arbitrary data types, including objects. Spec Explorer is being used daily by several Microsoft product groups. The here presented techniques are used to test operating system components and Web service infrastructure.

Transition Systems Formalize Reactive and Distributed Systems. Reactive and distributed systems are inherently nondeterministic. No single agent (component, thread, etc.) controls all state transitions, and even external entities like the operating systems scheduler or the network may play a role.

A practical and theoretically sound way to test the evolution of semi-independent state spaces is to use a kind of transition system known as an interface automaton [3]. *Interface automata* make a distinction between input transitions and output transitions. In some states, input is enabled and we can drive the system forward by giving it new things to do; at other times the system and its environment choose what happens next. This is like a game where players take turns. Sometimes it is our turn to make a move; sometimes it is the systems.

To illustrate how this works we will use a network-based chat system as an example. In the chat system there are multiple clients that may post messages. The system delivers pending messages in FIFO order with local consistency. Figure 1 shows a typical scenario of the chat systems' behavior as an interface automaton. The nodes of the graph represent distinct states of the system. The arcs represent actions that change the systems state. Each state in the graph is either input enabled or output enabled. The states drawn with ovals represent *active, input-enabled states* where a client may give the system new work to do. States drawn with diamonds are *passive, output-enabled states* where the system reacts to input or spontaneously makes a move of its own choosing. The Post *action* is said to be *controllable* because it can be invoked by a user to provide system input. The Deliver action is only *observable*; that is, it is an output message. The names of observable actions in the graph are prefixed by the ? symbol. Note that in some passive states there is a race between what the user may do and what the system may do. The Timeout transition, here represented by a transition that carries no label, indicates that no output was seen in the time the user was willing to wait. This causes a transition from an output-enabled state to an input-enabled state.

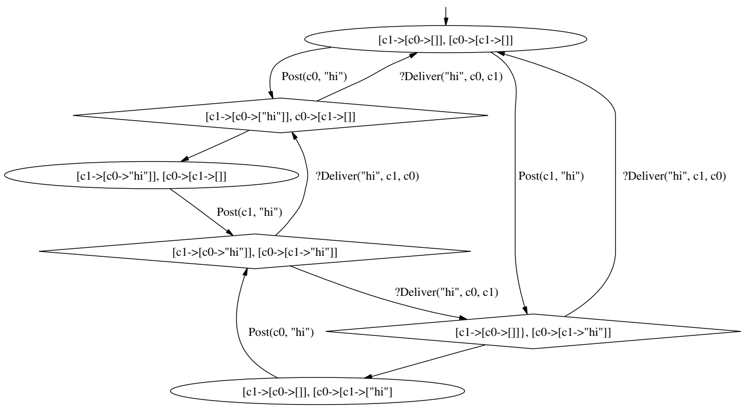
[c1->[c0->[]], [c0->[c1->[]]
Post(c0, "hi")
?Deliver("hi", c0, c1)
[c1->[c0->["hi"]], c0->[c1->[]]
[c1->[c0->"hi"]], [c0->[c1->[]]
?Deliver("hi", c1, c0)
Post(c1, "hi")
?Deliver("hi", c1, c0)
Post(c1, "hi")
[c1->[c0->"hi"]], [c0->[c1->"hi"]]
?Deliver("hi", c0, c1)
Post(c0, "hi")
[c1->[c0->[]]}, [c0->[c1->"hi"]]
[c1->[c0->[]], [c0->[c1->["hi"]

Fig. 1. Exploration of the Chat model with two clients (c0 and c1), a fixed message (“hi”) under the restriction that at most 1 message from each sender must still be delivered to a client

Model Programs Compactly Encode Large Transition Systems. Interface automata describe fixed scenarios. But we are not only interested in modeling and testing a fixed scenario; we want to have a general description for a protocol, like a chat server. This is where model programs can help. Rather than coding our system description directly as a transition system, we use a *model program* to express system behavior as an “abstract state machine”. Tools like Spec Explorer that analyze the states of the machine can produce the transition system needed for testing.

Here is a model program that describes the chat system shown above, written in the Spec# language. The state of the system consists of instances of the class `Client` that have been created so far, and a map `Members` that for each client specifies the messages that have been sent but not yet delivered to that client as sender queues. Each sender queue is identified by the client that sent the messages in the queue. In the initial state of the system there are no clients and and `Members` is an empty map.

```
class Client
type Message = string;
type SendersQueue = Map<Client,Seq<Message>>;
type MemberState = Map<Client,SendersQueue>;

MemberState Members = new Map();
```

We give two actions of the system. Actions are methods with preconditions that say in which state of the system they may occur and for which input parameters. A member of the chat session may post a message for all members except himself to receive. When a sender posts a message, the message is appended at the end of the corresponding sender queue of each of the other members of the session.

```
void Post(Client sndr, Message msg)
  requires sndr in Members && Members.Size > 1;
{ foreach(rcvr in Members)
    if (rcvr != sndr) Members[rcvr][sndr].Add(msg); }
```

A message being delivered from a sender to a receiver is an observable action or a notification callback that occurs whenever the chat system forwards a particular message to a particular client. When a delivery is observed, the corresponding sender queue of the receiver has to be nonempty, and the message must match the first message in that queue or else local consistency is violated. If the preconditions of the delivery are satisfied then the delivered message is simply removed from the corresponding sender queue of the recipient.

```
void Deliver(Message msg, Client sndr, Client rcvr)
  requires rcvr in Members && sndr in Members[rcvr];
  requires Members[rcvr][sndr].Length > 0 &&
           Members[rcvr][sndr].Head == msg;
{ Members[rcvr][sndr] = Members[rcvr][sndr].Tail; }
```

When a client joins the session, the related message queues are initialized appropriately.

To encode a specification of the system's intended behavior in machine-executable form is not the same as writing a second implementation. The model program does less than the implementation. Its purpose is to capture the states of the system that affect the observable behavior of interest.

Exploration can Reveal the Interface Automaton of a Model Program. The interface automaton defined by a model program is a complete unwinding or expansion of the program. An *explicit state model checking algorithm* is used to compute the (possibly infinite) space of all possible sequences of method invocations that 1) do not violate the pre- and postconditions and invariant of the system's contracts and 2) are relevant to a user-specified set of test properties [4].

If the model is infinite state, unwinding doesn't terminate. Spec Explorer thus includes practical features that control how the state space is explored. We mention two of these: *State groupings* allow the exploration to prune away states that are distinct but indistinguishable under a user-provided equivalence relation [1]. Avoiding isomorphic cases that differ in the choice of input but have identical runs results in a body of tests with a better chance of detecting a conformance discrepancy. *State-dependent parameter generation* allows to compute the parameter domains of each action with respect to the current state. This can make exploration more efficient by reducing the search for input parameters to feasible cases.

Interface Automaton Provides the Basis for Model-Based Test Case Generation. Test cases can be automatically generated by *traversing the graph* of the interface automaton. The graph also serves as a *test oracle*: a test fails if observed transitions of the implementation under test do not match transitions in the graph. Additionally, successful test runs must begin in the *initial state* and terminate in an accepting state. *Accepting states* are states that satisfy a user-specified logical condition that says whether the system is in a final, deinitialized state. In this example, the accepting state occurs whenever the message queues are empty.

Differences between the predicted and actual system behavior are called conformance failures. What constitutes a difference is mathematically defined in terms of alternating refinement of interface automata. *Alternating refinement* means that the system under test must accept at least as many inputs as the interface automaton defines (it may accept more inputs) and that, conversely, the test harness must accept at least as many outputs as the system may produce (it may accept more outputs than the system is capable of producing) [2].

Our test graphs are also used to *automatically harness* the implementation for conformance testing. Spec Explorer can instrument a .NET assembly and cause implementation methods corresponding to model actions to be invoked as needed.

Running a test results in a trace log that shows a comparison of expected versus actual behavior. Here is an example:

Step	Invocation	From State	To State	Status
1	Post(c0, ”Hi”)	S0	S1	Succeeded
2	?Timeout	S1	S1’	Succeeded
3	Post(c0, ”Bye”)	S1’	S7	Succeeded
4	?Deliver(”Bye”, c0, c1)	S7	S2	FAILED: observed Deliver(”Bye”, c0, c1), expected Deliver(”Hi”, c0, c1)

This test run observed that the particular chat system implementation being tested did not deliver messages in the order posted, as required by the specification. The server delivered in LIFO order instead of FIFO.

Game Strategies Help Achieve Test Goals. Although any traversal of the graph is a possible trace of the system, we can only choose moves in the active states (i.e., those drawn as ovals in the graph). A state where the system can choose from among more than one move represents nondeterminism from the observers point of view. This means a test case is not a just sequence of actions but a tree of actions and possible system responses. Executing a test is like a so-called *game against nature* where a players opponent chooses moves randomly. Spec Explorer implements game strategies using Markov decision processes as a technique for intelligently choosing input actions that broaden the coverage of nondeterministic tests [5].

On-the-Fly Conformance Checking Scales to Very Large State Spaces. When dealing with model programs that have very large state spaces, we can combine the state exploration and test case generation into an online algorithm called *on-the-fly testing* [6].

When testing in its on-the-fly mode, Spec Explorer’s exploration makes moves based on the observed history of the test run. This allows it to omit exploration of nondeterministic branches that were not taken by the implementation during the test run. It can also be run in a way that attempts to match the distribution of actions exercised during testing to an application profile given as input.

Spec Explorer users rely on both pre-generated, offline tests with complete behavioral coverage over a restricted domain of system inputs and online tests generated on the fly which randomly sample a larger number of system inputs.

Empirical Evidence Shows that Spec Explorer is Effective. Spec Explorer was internally released in summer 2004. Since then approx. 100 testers use it on a daily basis. In fact, most of Microsoft’s forthcoming Web service infrastructure was tested with Spec Explorer and so are components of the Windows operating system.

For instance, recently the Windows test team split a feature set into 4 components and decided to test 2 of them traditionally and 2 with Spec Explorer. The modeling team build (1) a system-level object-model consisting of approx. 200 and another one (2) of approx. 3500 lines of non-blank Spec# code. The multi-threaded implementations under test have 2000 and 20000 lines of non-blank C++ code respectively.

In this particular setting, the model-based approach helped to discover 10 times more errors than traditional test automation. Also the kind of bugs discovered were deep system-level bugs (i.e. bugs that were only found after the system performed many steps), for which manual test cases would have been hard to construct.

The effort in developing the models took roughly the same amount of time as developing the traditional test automation. The biggest impact that the modeling effort had was during the design phase, the process helped to discover and resolve 2 times more design issues than bugs that were found afterwards.

Microsoft developers typically can only check in code, which unit tests achieved already more than 60% feasible branch coverage. It is the testers task to improve this coverage. For (1) and (2) the testers refined the models so that they achieved 100% and 70% feasible branch coverage, respectively. While this improved the statistics, it does not reflect on how well a concurrent implementation is tested. In most cases when bugs were found, at least two or more threads and a shared resource were involved, although the same code coverage could often be achieved with a single thread.

When developing new versions of the code, models need to be adjusted, but such changes are typically local, whereas manual test cases have to be redesigned and sometimes completely rewritten. We have repeatedly observed that this is where model-based testing substantially reduces test case development time.

But caution: When customers discover discrepancies between model and implementation using our tool, typically about half of them originate from the informal requirements specification, the model, or bugs in the test harness, and half are due to coding errors in the implementation under test. But so far every team agreed that the modeling effort was helpful – not only for test, but also, and in particular for design.

References

1. C. Campbell and M. Veanes. State exploration with multiple state groupings. In D. Beauquier, E. Börger, and A. Slissenko, editors, *12th International Workshop on Abstract State Machines, ASM'05, March 8–11, 2005, Laboratory of Algorithms, Complexity and Logic, University Paris 12 – Val de Marne, Créteil, France*, pages 119–130, 2005.
2. L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.
3. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM, 2001.
4. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA'02*, volume 27 of *Software Engineering Notes*, pages 112–122. ACM, 2002.

5. L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA'04*, volume 29 of *Software Engineering Notes*, pages 55–64. ACM, July 2004.
6. M. Veanes, C. Campbell, W. Schulte, and P. Kohli. On-the-fly testing of reactive systems. Technical Report MSR-TR-2005-03, Microsoft Research, January 2005.