Online Testing with Model Programs

Margus Veanes Colin Campbell Wolfram Schulte Nikolai Tillmann
Microsoft Research, Redmond, WA, USA
margus,colin,schulte,nikolait@microsoft.com

ABSTRACT

Online testing is a technique in which test derivation from a model program and test execution are combined into a single algorithm. We describe a practical online testing algorithm that is implemented in the model-based testing tool developed at Microsoft Research called Spec Explorer. Spec Explorer is being used daily by several Microsoft product groups. Model programs in Spec Explorer are written in the high level specification languages AsmL or Spec#. We view model programs as implicit definitions of interface automata. The conformance relation between a model and an implementation under test is formalized in terms of refinement between interface automata. Testing then amounts to a game between the test tool and the implementation under test.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Reliability,Verification

Keywords

Conformance testing, interface automata, runtime verification

1. INTRODUCTION

In this paper we consider testing of *reactive systems*. Reactive systems take inputs as well as provide outputs in form of spontaneous reactions. Testing of reactive systems can very naturally be viewed as a two-player game between the tester and the implementation under test (IUT). Transitions are moves that may originate either from the tester or from the IUT. The tester may use a *strategy* to choose which of the inputs to apply in a given state.

We describe here a new online technique for testing reactive systems. In this approach we join test derivation from a *model program* and test execution into a single algorithm. This combines the benefits of encoding transitions as method invocations of a model program with the benefits of a game-based framework for reactive systems. Test cases become test strategies that are created dynamically as testing proceeds and take advantage of the knowledge gained by exploring part of the model state space.

Formally, we consider model programs as implicit definitions of interface automata and formulate the conformance relation between a model program and a system under test in terms of alternating simulation. This is a new approach for formalizing the testing of reactive systems.

The technique we describe was motivated by problems that we observed while testing large-scale commercial systems, and the resulting algorithm has been implemented in a model-based testing tool developed at Microsoft Research. This tool, called Spec Explorer [1, 7], is in daily use by several product groups inside of Microsoft. The online technique has been used in an industrial setting to test operating system components and Web service infrastructure.

To summarize, we consider the following points as the main contributions of the paper:

- The formalization of model programs as interface automata and a new approach to using interface automata for conformance testing, including the handling of timeouts.
- A new online or on-the-fly algorithm for using model programs for conformance testing of open systems, where the conformance relation being tested is alternating simulation. The algorithm uses state dependent timeouts and state dependent action weights as part of its strategy calculation.
- Evaluation of the effectiveness of the theory and the tool for the test of critical industrial applications.

The rest of the paper is organized into sections as follows. In Section 2 we formalize what it means to specify a reactive system using a model program. This includes a description of how the Spec Explorer tool uses a model program as its input. Then in Section 3, we describe the algorithm for online testing. In Section 4 we walk through a concrete example that runs in the Spec Explorer tool. We evaluate further sample problems and industrial applications of the tool in Section 5. Related work is discussed in Section 6. Finally, we mention some open problems and future work in Section 7.

2. SPECIFYING REACTIVE SYSTEMS USING MODEL PROGRAMS

To describe the behavior of a reactive system, we use the notion of interface automata [11, 10] following the exposition in [10]. Instead of the terms “input” and “output” that are used in [10] we use the terms “controllable” and “observable” here. This choice of terminology is motivated by our problem domain of testing, where certain operations are under the control of a tester, and certain operations are only observable by a tester.

DEFINITION 1. An *interface automaton* M has the following components:

- A set S of *states*.
- A nonempty subset S^{init} of S called the *initial states*.
- Mutually disjoint sets of *controllable actions* A^{c} and *observable actions* A^{o}.
- *Enabling functions* Γ^{c} and Γ^{o} from S to subsets of A^{c} and A^{o}, respectively.
- A *transition function* δ that maps a source state and an action enabled in the source state to a target state.

Remark about notation: In order to identify a component of an interface automaton M, we index that component by M, unless M is clear from the context.

We write A_M for the set $A^{\text{c}}_M \cup A^{\text{o}}_M$ of all actions in M, and we let $\Gamma_M(s)$ denote the set $\Gamma^{\text{c}}_M(s) \cup \Gamma^{\text{o}}_M(s)$ of all *enabled* actions in a state s. An action a *transitions from* s *to* t if $\delta_M(s, a) = t$.

2.1 Model program as interface automaton

A model program P declares a finite set of *action methods* and a set of (*state*) *variables*. A model *state* is a mapping of state variables to concrete values.[1] Model programs in Spec Explorer are written in a high level specification language AsmL [16] or Spec# [4]. An action method m is similar to a method written in a normal programming languages like C#, except that m is in addition associated with a state-based predicate $\textit{Pre}_m[\mathbf{x}]$ called the *precondition* of m, that may depend on the input parameters $\mathbf{x}$ of m. An example of a model program and some of the concepts introduced in this section are illustrated below in Section 4.

The interface automaton M_P defined by a model program P is a complete unwinding or expansion of P as explained next. We omit the suffix P from M_P as it is clear from the context. The set of initial states S^{init}_M of M is the singleton set containing the initial assignment of variables to values as declared in P.

For a sequence of method parameters $\vec{v}$, we write $\vec{v}_{\text{in}}$ for the input parameters, i.e. the arguments, and we write $\vec{v}_{\text{out}}$ for the output parameters, in particular including the return value.

The transition function δ_M maps a source state s and an action $a = \langle m, \vec{v} \rangle$ to a target state t, provided that the following conditions are satisfied:

- $\textit{Pre}_m[\vec{v}_{\text{in}}]$ holds in s,
- the method call $m(\vec{v}_{\text{in}})$ in state s produces the output parameters $\vec{v}_{\text{out}}$, and yields the resulting state t;

In this case the action a is enabled in s. Each action method m, is associated in a state s with a set of enabled actions $\textit{Enabled}_m(s)$. The set of all enabled actions $\Gamma_M(s)$ in a state s is the union of all $\textit{Enabled}_m(s)$ for all action methods m. The set of states S_M is the least set that contains S^{init}_M and is closed under δ_M. The set A_M is the union of all $\Gamma_M(s)$ for s in S_M.

2.1.1 *Reactive behavior*

In order to distinguish behavior that a tester has full control over from behavior that can only be observed about the implementation under test (IUT), the action methods of a model program are disjointly partitioned into *controllable* and *observable* ones. This induces, for each state s, a corresponding partitioning of $\Gamma_M(s)$ into controllable actions $\Gamma^{\text{c}}_M(s)$ enabled in s, and observable actions $\Gamma^{\text{o}}_M(s)$ enabled in s. The action set A_M is partitioned accordingly into A^{c}_M and A^{o}_M. A state s where $\Gamma_M(s)$ is empty is called *terminal*. A nonterminal state s where $\Gamma^{\text{o}}_M(s)$ is empty is called *active*; s is called *passive* otherwise.

2.1.2 *Accepting States*

In Spec Explorer the user associates the model program with an *accepting state condition* that is a Boolean expression based on the model state. The notion of accepting states is motivated by the practical need to identify model states where tests are allowed to terminate. This is particularly important when testing distributed or multi-threaded systems, where IUT does not always have a global reset that can bring it to its initial state. Thus ending of tests is only possible from certain states from which reset is possible. For example, as a result of a controllable action that starts a thread in the IUT, the thread may acquire shared resources that are later released. The successful test should not be finished before the resources have been released.

From the game point of view, the player, i.e. the test tool, may choose to make a move from an accepting state s to a terminal *goal* state identifying the end of the play (or test), irrespective of whether there are any other moves (either for the player or the opponent) possible in s. Notice that an accepting state does not oblige the player to end the test. By restating that in terms of the interface automaton M, there is a controllable *finish* action in A_M and a *goal* state g in S_M, s.t., for all accepting states s, $\delta_M(s, \textit{finish}) = g$. In IUT, the *finish* action must transition from a corresponding state t to a terminal state as well, reflecting the assumption that IUT can reset the system at this point. Thus, ending the test in an accepting state, corresponds to choosing the *finish* action.

2.2 IUT as interface automaton

In the Spec Explorer tool the model program and the IUT are both given by a collection of APIs in form of .NET libraries (or dlls). Typically the IUT is given as a collection of one or more "wrapper" APIs of the actual system under test. The actual system is often multi-threaded if not distributed, and the wrapper is connected to the actual system through a customized test harness that provides a particular high-level view of the behavior of the system matching the abstraction level of the model program. The wrapper provides a serialized view of the observable actions resulting from the execution of the actual system. It is very common that only a particular aspect of the IUT is being tested through the harness. In this sense the IUT is an open system.

The program of the IUT can be seen as a restricted form of a model program. We view the behavior of the IUT in the same way as that of the specification. The interface automaton corresponding to IUT is denoted by M_{IUT}.

The *finish* action in the IUT typically kills the processes or terminates the threads (if any) in the actual system under test.

2.3 Conformance relation

The conformance relation between a model and an implementation is formalized as refinement between two interface automata. In order for the paper to be self contained we define first the notions of alternating simulation and refinement following [10]. The view of the model and the implementation as interface automata is a mathematical abstraction. We discuss below how the conformance relation is realized in the actual implementation.

In the following we use M to stand for the specification interface automaton and N for the implementation interface automaton.

[1] In terms of mathematical logic, states are *first-order structures*.

DEFINITION 2. An *alternating simulation* ρ *from* M *to* N is a relation $\rho \subseteq S_M \times S_N$ such that, for all $(s,t) \in \rho$,

1. $\Gamma^{\mathrm{c}}_M(s) \subseteq \Gamma^{\mathrm{c}}_N(t)$ and $\Gamma^{\mathrm{o}}_M(s) \supseteq \Gamma^{\mathrm{o}}_N(t)$, and
2. $\forall a \in \Gamma^{\mathrm{o}}_M(s) \cup \Gamma^{\mathrm{c}}_N(t)$, $(\delta_M(s,a), \delta_N(t,a)) \in \rho$.

The intuition is as follows. Condition 1 ensures that, on one hand all controllable actions in the model are possible in the implementation, and on the other hand that all possible responses from the implementation are enabled in the model. Condition 2 guarantees that if condition 1 is true in a given pair of source states then it is also true in the resulting target states of any controllable action enabled in the model and any observable action enabled in the implementation.

DEFINITION 3. An interface automaton M *refines* an interface automaton N if

1. $A^{\mathrm{o}}_M \subseteq A^{\mathrm{o}}_N$ and $A^{\mathrm{c}}_M \subseteq A^{\mathrm{c}}_N$, and
2. there is an alternating simulation ρ from M to N, $s \in S^{\mathrm{init}}_M$, and $t \in S^{\mathrm{init}}_N$ such that $(s,t) \in \rho$.

The first condition of refinement is motivated in the following section. Intuition for the second condition can be explained in terms of a *conformance game*. Consider two players: a *controller* and an *observer*. The game starts from an initial state in $S^{\mathrm{init}}_M \times S^{\mathrm{init}}_N$. During one step of the game one of the players makes a move. When the controller makes a move, it chooses an enabled controllable action a in the current model state s and transitions to $(\delta_M(s,a), \delta_N(t,a))$, where the chosen action must be enabled in the current implementation state t or else there is a conformance failure. Symmetrically, when the observer makes a move, it chooses an enabled observable action in the current IUT state t and transitions to the target state $(\delta_M(s,a), \delta_N(t,a))$, where the chosen action must be enabled in the current model state s or else there is a conformance failure. The game continues until the controller decides to end the game by transitioning to the goal state.

2.4 Conformance checking in Spec Explorer

We provide a high level view of the conformance checking engine in Spec Explorer. We motivate the view of IUT as an interface automaton and explain the mechanism used to check acceptance of actions.

Spec Explorer provides a mechanism for the user to bind the actions methods in the model to methods with matching signatures in the IUT. Without loss of generality, we assume here that the signatures are in fact the same. Usually the IUT has more methods available in addition to those that are bound to the action methods in the model, which explains the first condition of the refinement relation. In other words, the model usually addresses one aspect of the IUT and not the complete functionality of IUT.

The user partitions the action methods into observable and controllable ones. In order to track the spontaneous execution of an observable action in the IUT, possibly caused by some internal thread, Spec Explorer instruments the IUT at the binary (MSIL) level. During execution, the instrumented IUT calls back into the conformance engine, notifying it about occurrences of observable method calls. The conformance engine buffers these occurrences, such that they can occur even during the execution of a controllable method in the implementation. A typical scenario is that a controllable action starts a thread in the implementation, during the execution of which several observable actions (callbacks) may happen. Another scenario is that there is only one thread of control; however, observable methods are invoked in course of executing a controllable method.

In the following we describe how a controllable action $a = \langle m, \vec{v}\rangle$ is chosen in the model program P and how its enabledness in the IUT is checked. First, input parameters $\vec{v}_{\text{in}}$ for m are generated such that the precondition of the method call $m(\vec{v}_{\text{in}})$ holds in P. Second, $m(\vec{v}_{\text{in}})$ is executed in the model and the implementation, producing output parameters $\vec{v}_{\text{out}}$ and $\vec{w}$, respectively. Thus a is at this point an enabled action in the model. Third, to determine enabledness of a in the IUT, the expected output parameters $\vec{v}_{\text{out}}$ of the model and the output parameters $\vec{w}$ of the IUT are compared for equality, if $\vec{v}_{\text{out}} \neq \vec{w}$ then a is enabled in the model but not in the IUT, resulting in a conformance failure. For example, if $\vec{v}_{\text{out}}$ is the special return value unit of type *void* but IUT throws an exception when $m(\vec{v}_{\text{in}})$ is invoked, (i.e. $\vec{w}$ is an exception value), then a conformance failure occurs.

An observable action $a = \langle m, \vec{v}\rangle$ happens as a spontaneous reaction from the IUT, which occurrence is buffered in the conformance engine. When the conformance engine is in a state where it consumes the next observable action from that buffer, it proceeds as follows. Let $a = \langle m, \vec{v}\rangle$ be the next action in the buffer. First, the precondition of the method call $m(\vec{v}_{\text{in}})$ is checked in P. If the precondition does not hold, a is not enabled in the model and a precondition conformance failure occurs. Otherwise, $m(\vec{v}_{\text{in}})$ is executed in the model yielding either a conformance failure in form of a model invariant or postcondition failure, or yielding $\vec{w}$. If $\vec{v}_{\text{out}} \neq \vec{w}$, an unexpected return value (or output parameter) conformance failure will be generated. If none of this failure situations occur, a is admitted by the model, which then transitions from its current state s to $\delta_{M_P}(a,s)$.

3. ONLINE TESTING

Online testing (also called on-the-fly testing in the literature) is a technique in which test derivation from a model program and test execution are combined into a single algorithm. By generating test cases at run time, rather than pre-computing a finite transition system and its traversals, this technique is able to:

- Resolve the nondeterminism that typically arises in testing reactive, concurrent and distributed systems. This avoids generating huge pre-computed test cases in order to deal with all possible responses from the system under test.
- Stochastically sample a large state space rather than attempting to exhaustively enumerate it.
- Provide user-guided control over test scenarios by selecting actions during the test run based on a dynamically changing probability distribution.

In Spec Explorer, the online testing algorithm (OLT) uses a (dynamically changing) strategy to select controllable actions. OLT also stores information about the current state of the model, by keeping track of the state transitions due to controllable and observable actions. The behavior of OLT depends on various user configurable parameters. The most important ones are timeouts and action weights. Before explaining the algorithm we introduce the OLT parameters and explain their role in the algorithm.

3.1 Timeouts

In Spec Explorer there is a *timeout function* Δ, given by a model-based expression, that in a given state s evaluates to a value $\Delta(s)$ of type *System.TimeSpan* in the .NET framework. The primary purpose of the timeout function is to configure the amount of time that

OLT should wait for to get a response from the IUT. The timeout value may vary from state to state and may be 0 (which is the default). The definition of the timeout function may reflect network latencies, performance of the actual machine under test, time complexity of the action implementations, test harnessing, etc, that may vary between different test setups. In some situations, the use of the timeout function is reminiscent of checking for quiescence in the sense of ioco theory [20], e.g., when a sufficiently large time span value is associated with an active state. Note however that a timeout is typically enabled in the same state as observable actions and does not correspond to quiescence.

The exact time span values do not affect the conformance relation. To make this point precise, we introduce a timeout extension M^{t} of an interface automaton M as follows. The timeout extension of an interface automaton is used in OLT.

DEFINITION 4. A *timeout extension* M^{t} of an interface automaton M is the following interface automaton. The state vocabulary of M^{t} is the state vocabulary of M extended with a Boolean variable *timeout*. The components of M^{t} are:

- $S_{M^{\mathrm{t}}} = \{s^T, s^F : s \in S_M\}$, where *timeout* is true in s^T and false in s^F.
- $S^{\mathrm{init}}_{M^{\mathrm{t}}} = \{s^F : s \in S^{\mathrm{init}}_M\}$
- $A^{\mathrm{o}}_{M^{\mathrm{t}}} = A^{\mathrm{o}}_M$ and $A^{\mathrm{c}}_{M^{\mathrm{t}}} = A^{\mathrm{c}}_M \cup \{\sigma\}$, where σ is called a *timeout event*.
- Observable actions are only enabled if *timeout* is false:

  $$\Gamma^{\mathrm{c}}_{M^{\mathrm{t}}}(s^F) = \Gamma^{\mathrm{c}}_M(s) \cup \{\sigma\}, \quad \Gamma^{\mathrm{c}}_M(s^T) = \emptyset, \text{ for all } s \in S_M,$$

  and controllable actions are only enabled if *timeout* is true:

  $$\Gamma^{\mathrm{o}}_{M^{\mathrm{t}}}(s^T) = \Gamma^{\mathrm{o}}_M(s), \quad \Gamma^{\mathrm{o}}_{M^{\mathrm{t}}}(s^F) = \emptyset, \quad \text{for all } s \in S_M.$$

- The transition function $\delta_{M^{\mathrm{t}}}$ is defined as follows. For all $s, t \in S_M$ and $a \in \Gamma_M(s)$ such that $\delta_M(s, a) = t$,
  - If a is controllable then $\delta_{M^{\mathrm{t}}}(a, s^T) = t^F$.
  - If a is observable then $\delta_{M^{\mathrm{t}}}(a, s^F) = t^F$.

  The timeout event sets *timeout* to true: $\delta_{M^{\mathrm{t}}}(\sigma, s^F) = s^T$ for all $s \in S_M$.

We say that a state s of M^{t} is an *observation* or *passive* state if *timeout* is false in s, we say that s is a *control* or *active* state otherwise.

3.2 Action weights

Action weights are used to configure the strategy of OLT to choose controllable actions. There are two kinds of weight functions: per-state weight function and decrementing weight function. Each action method in P is associated with a weight function of one kind. Let $\#(m)$ denote the number of times a controllable action method has been chosen during the course of a single test run in OLT.

- A *per-state weight function* is a function ω^{s} that maps a model state s and a controllable action method m to a nonnegative integer.
- A *decrementing weight function* is a function ω^{d} of the OLT algorithm that maps a controllable action method m to the value $\max(\omega^{\mathrm{init}}_m - \#(m), 0)$, where ω^{init}_m is an initial weight assigned to m.

We use $\omega(s, m)$ to denote the value of $\omega^{\mathrm{s}}(s, m)$ if m is associated with a per-state weight function, we use $\omega(s, m)$ to denote the value of $\omega^{\mathrm{d}}(m)$ otherwise.

In a given model state s the action weights are used to make a weighted random selection of an action method as follows. Let $m_1, \ldots, m_k$ be all the controllable action methods enabled in s, i.e. in $\Gamma^{\mathrm{o}}_M(s)$, the probability of an action method m_i being chosen is

$$\mathit{prob}(s, m_i) = \begin{cases} 0, & \text{if } \omega(s, m_i) = 0; \\ \omega(s, m_i) / \sum_{j=1}^{k} \omega(s, m_j), & \text{otherwise.} \end{cases}$$

A per-state weight can be used to guide the direction of the exploration according to the state of the model. These weights can be used to selectively increase or decrease the probability of certain actions based on certain model variables. For example, assume P has a state variable *stack* whose value is a sequence of integers, and a controllable action method *Push*(x) that pushes a new value x on *stack*. One can associate a per-state weight expression *MaxStackSize* $-$ *Size*(*stack*) with *Push* that will make the probability of *Push* less as the size of stack increases and gets closer to the maximum allowed size.

Decrementing weights are used when the user wants to call a particular action method a specific number of times. With each invocation of the method, the associated weight decreases by 1 until it reaches zero, at which point the action will not be called again during the run. A useful analogy here is with a bag of colored marbles, one color per action method – marbles are pulled from the bag until the bag is empty. Using decrementing weights produces a random permutation of actions that takes enabledness of transitions into account.

3.3 Online testing algorithm

We provide here a high level description of the OLT algorithm. We are given a model program P and an implementation under test IUT. The purpose of the OLT algorithm is to generate tests that provide evidence for the refinement from the interface automaton M to the interface automaton $M^{\mathrm{t}}_{\mathrm{IUT}}$, where M is the timeout extension M^{t}_P of M_P.

It is convenient to view OLT as a conservative extension of M where the information about $\#(m)$ is stored in the OLT state, since the controller strategy may depend on this information. This does not affect the conformance relation. In the initial state of OLT, $\#(m)$ is 0 for all controllable action methods m.

A *controller strategy* (or *output strategy*) π maps a state $s \in S_{\mathrm{OLT}}$ to a subset of $\Gamma^{\mathrm{o}}_M(s{\restriction}M)$. A *controller step* is a pair (s, t) of OLT states such that $t{\restriction}M = \delta_M(s{\restriction}M, \langle m, \vec{v}\rangle)$ for some action $\langle m, \vec{v}\rangle$ in $\pi(s)$, and $\#(m)^t = \#(m)^s + 1$. In general, OLT may also keep more information, e.g. limited history of the test runs or, projected state machine coverage data, etc, that may affect the overall controller strategy in successive test runs of OLT. Such extensions are orthogonal to the description of the algorithm below, as they affect only π. An *observer step* is a pair (s, t) of OLT states such that $t{\restriction}M = \delta_M(s{\restriction}M, a)$ for some $a \in \Gamma^{\mathrm{c}}_M(s{\restriction}M)$, and $\#^s = \#^t$.

By a *test run* of OLT we mean a trace $\vec{s} = s_0 s_1 \ldots s_k \in S^+_{\mathrm{OLT}}$ where s_0 is the initial state and, for each i, (s_i, s_{i+1}) is a controller step or an observer step. A *successful* test run is a test run that ends in the goal state.

We are now ready to describe the top-level loop of the OLT algorithm. We write s_{OLT} for the current state of OLT. We say that an action a is *legal* (in the current state) if a is enabled in $s_{\mathrm{OLT}}{\restriction}M$, a is *illegal* otherwise. Initially s_{OLT} is the initial state of OLT. The following steps are repeated subject to additional termination conditions (discussed in the following section):

Step 1 (observe) Assume s_{OLT} is a passive state (*timeout* is false). OLT waits for an observable action until $\Delta(s_{\text{OLT}}\restriction M)$ amount of time elapses. If an observable action a occurs within this time, there are two cases:

1. If a is illegal then the test run *fails*.
2. If a is legal, OLT makes an observable step (s_{OLT}, s) and sets s_{OLT} to s. OLT continues from Step 1.

If no observable action happened, OLT sets *timeout* to true.

Step 2 (control) Assume s_{OLT} is an active state (*timeout* is true). Assume $\pi(s_{\text{OLT}}) \neq \emptyset$. OLT chooses an action $a \in \pi(s_{\text{OLT}})$, such that the probability of the method of a being m is

$$\mathit{prob}(s_{\text{OLT}}\restriction M, m),$$

and invokes a in the IUT. There are two cases:

1. If a is not enabled in IUT, the test run *fails*.
2. If a is enabled in IUT, OLT makes a controllable step (s_{OLT}, s), where s is an observation state, and sets s_{OLT} to s. OLT continues from Step 1.

Step 3 (terminate) Assume $\pi(s_{\text{OLT}}) = \emptyset$. If $s_{\text{OLT}}\restriction M$ is the goal state then the test run *succeeds* else the test run *fails*.

Notice that the timeout event in Step 1 happens immediately if $\Delta(s_{\text{OLT}}\restriction M) = 0$. In terms of M, a timeout event is just an observable action.

The failure verdict in Step 3 is justified by the assumption that a successful run must end in the goal state. Step 3 implicitly adds a new controllable action *fail* to A_{OLT} and, upon failure, a transition $\delta_M(s_{\text{OLT}}, \mathit{fail}) = s_{\text{OLT}}$, such that *fail* is never enabled in M_{IUT}. For example, if a timeout happens in a nonaccepting state and the subsequent state is terminal then the test run fails.

3.4 Termination conditions and cleanup phase

The algorithm uses several termination conditions. Most important of these is the desired length of test runs and the total length of all the test runs. When a limit is reached, but the current state of the algorithm is not an accepting state, then the main loop is executed as above but with the difference that the only controllable actions that are used must be marked as cleanup actions. The intuition behind cleanup actions is that they help drive the system to an accepting state. For example, actions like closing a file or aborting a transition are typical cleanup actions, whereas actions like opening a new file or starting a new task are not.

4. EXAMPLE: CHAT SERVER

We illustrate here how to model and test a simple reactive system, a sample called a *chat system*, using the Spec# specification language [4] and the Spec Explorer tool [1].

4.1 Overview

The chat system lets members enter the chat session. Members that have entered the session may post messages. The purpose of the model is to specify that all messages sent by a given client are received in the same order by all the recipients. We refer to this condition as the *local consistency* criterion of the system. For example, if there are two members in the session, client 1 and client 2, and client 1 sends two messages, first "hi" and then "bye", then client 2 must first receive "hi" and then receive "bye".

We do not describe how the chat session is created. Instead, we assume that there is a single chat session available at all times. The model is given in two parts. The first part describes the variables that encode the state at each point of the system's run. Each run begins in the initial state and proceeds in steps as actions occur. The second part describes the system's actions, such as entering the chat session or posting a message. Each action has preconditions that say in which state the action may occur and a method body that describes how the state changes as a result of the action.

4.2 System State

The state of the system consists of instances of the class `Client` that have been created so far, denoted by `enumof(Client)` in Spec#, 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.

```
class Client {}

MemberState Members = Map{};

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

The system state is identified by the values of `enumof(Client)` and `Members`. In the initial state of the system `enumof(Client)` is an empty set and `Members` is an empty map.

4.3 Actions

There are four action methods for the chat system: the *controllable* action methods `Create`, `Enter`, `Post`, and the *observable* action method `Deliver`. The `Create` action method creates a new instance of the `Client` class, as a result of this action the set of clients created so far, `enumof(Client)` is extended with the new client. Some of the preconditions are related to scenario control and are explained later.

```
Client! Create()
  requires CanCreate; // scenario control
{
  return new Client();
}
```

A client that is not already a member of the chat session may join the session. A client `c` becomes a member of the chat session when the action `Enter(c)` is called. When a client joins the session, the related message queues are initialized appropriately. The precondition $\mathit{Pre}_{\texttt{Enter}}[c]$ of `Enter` is c **`notin`** `Members`.

```
void Enter(Client! c)
  requires CanEnter; // scenario control
  requires c notin Members;
{
  foreach (Client d in Members)
    Members[d] += Map{<c,Seq{}>};
  Members += Map{<c,Map{d in Members; <d,Seq{}>}>};
}
```

A member of the chat session may post a message to all the other members. 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 CanPost; // scenario control
  requires sndr in Members && Members.Size > 1;
{
  foreach (rcvr in Members)
    if (rcvr != sndr) Members[rcvr][sndr] += Seq{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;
}
```

4.4 Scenario control

The Spec Explorer tool allows the user to limit the exploration of the model in various ways. We illustrate some of this functionality on this example.

4.4.1 *Additional preconditions*

In order to impose a certain order of actions we introduce the following derived mode property and define the scenario control related enabling conditions for the controllable action methods using the mode property. The scenario we have in mind is that all clients are created first, then they enter the session, and finally they start posting messages. We also limit the number of clients here to be two.

The additional preconditions that limit the applicability of actions are only applied to controllable actions. The preconditions constrain the different orders in which the controllable actions will be called. For observable actions, any violation of the preconditions is a conformance failure.

```
enum Mode { Creating, Entering, Posting };

Mode CurrentMode { get {
  if (enumof(Client).Size < 2) return Mode.creating;
  if (Members.Size < 2) return Mode.entering;
  return Mode.posting; }
}

bool CanCreate {get{return CurrentMode==Mode.Creating;}}
bool CanEnter  {get{return CurrentMode==Mode.Entering;}}
bool CanPost   {get{return CurrentMode==Mode.Posting; }}
```

4.4.2 *Default parameter domains*

In order to execute the action methods we also need to provide actual parameters for the actions. In this example we do so by restricting the domain of possible messages to fixed strings by providing a default value for the `Message` type through exploration settings of Spec Explorer. In general, parameters to actions are specified by using state dependent parameter generators. A parameter generator of an action method m is evaluated in each state separately and produces in that state a collection of possible input parameter combinations for m.

4.4.3 *State filters*

We may restrict the reachable states of the system with state-based predicates called *filters*. A transition to a new state is ignored if the state does not satisfy the given filters. We make use of two filters in this example: `NumberOfPendingDeliveries <` k for some fixed k, and `NoDuplicates` that are defined below. The first filter prevents the message queues from having more than $k-1$ pending deliveries in total in any given state. The second filter prevents a given message from being posted in states where that message is already in the queue of messages pending delivery.

```
int NumberOfPendingDeliveries { get {
  return Sum{c in Members;
             Sum{d in Members[c]; Members[c][d].Length}};
}}
bool NoDuplicates { get {
  return Forall{c in Members, d in Members[c];
                NoDupsSeq(Members[c][d])};
}}
bool NoDupsSeq(Seq<string> s) {
  return Forall{x in s; Exists1{y in s; x == y}};
}
```

4.4.4 *State groupings*

In Spec Explorer one can use state groupings to avoid multiple equivalent states [13, 8] from being explored. By using the following state grouping we can avoid different orders in which clients enter the session:

```
object IgnoreEnteringOrder { get {
 if (CurrentMode == Mode.posting) return Members;
 else return <enumof(Client),Members.Size>;
}}
```

Notice that in `entering` mode, the number of members increases, but the grouping implies that any two states with the same number of members are indistinguishable. Without using the grouping we would also get the two transitions and the intermediate state where client `c1` has entered the session before client `c0`. With n clients there would be n factorial many orders that are avoided with the grouping. The use of groupings has sometimes an effect similar to partial order reduction.

If we explore the chat model with the given constraints and only allow a single message “hi”, then we explore the state space that is shown Figure 1.

4.5 Execution

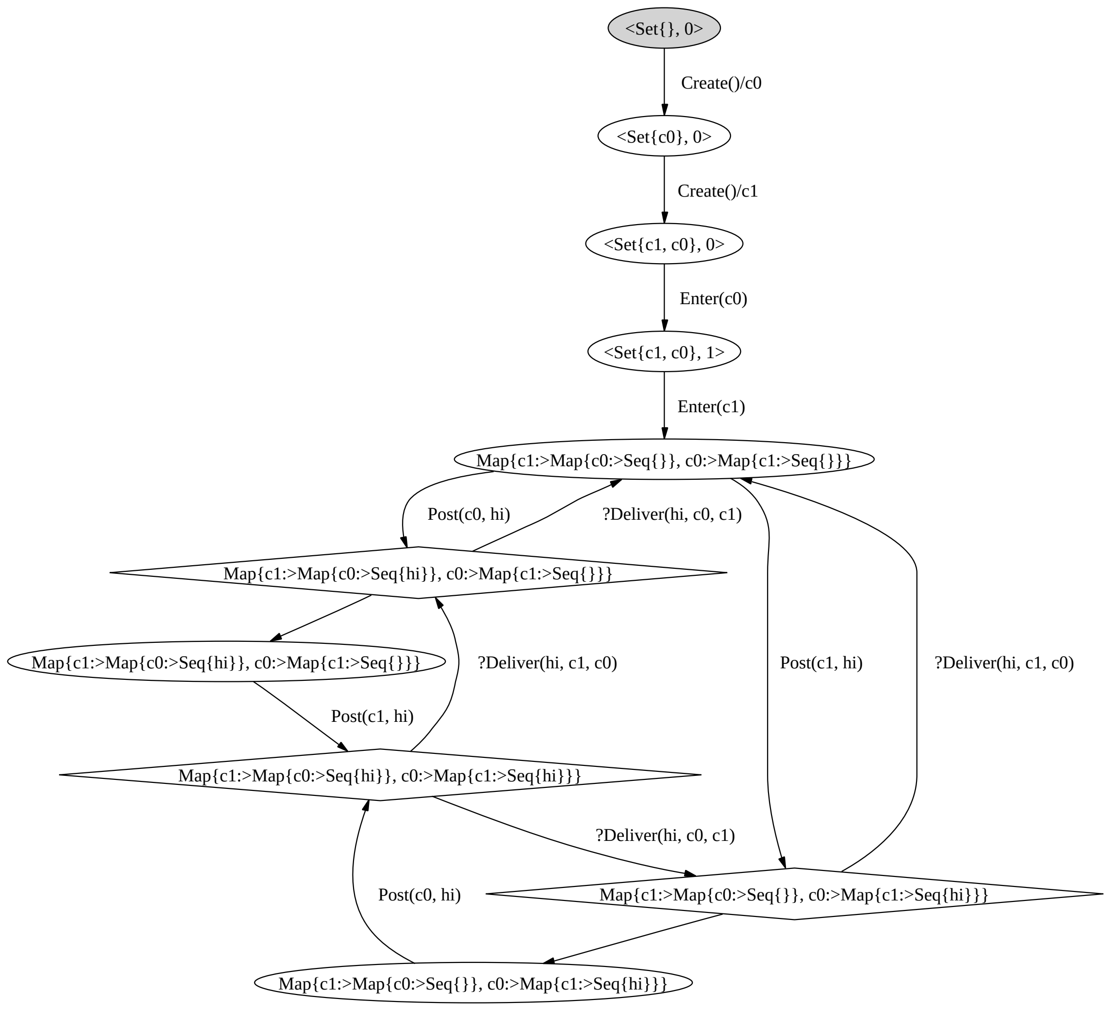
Before we can run the model program against a chat server implementation, here realized using TCP/IP and implemented in .NET, Spec Explorer requires that we complete the test configuration. We do so by providing a reference to the implementation and establish conformance bindings, which are isomorphic mappings between the signature of the model program and the IUT.

Our methodology also requires that objects in the model that are passed as input arguments, must have a one-to-one correspondence with objects in the IUT. This dynamic binding is implicitly established by the `Create` call, which, when run, binds the object created in the model space automatically to the object returned by the implementation.

Running this example in the Spec Explorer tool with the online algorithm showed a number of conformance discrepancies with respect to a TCP/IP-based implementation of this specification written in C#. In particular, the implementation does not respect the local consistency criterion and creates new threads for each message that is posted, without taking into account whether prior messages from the same sender have been delivered to the given receiver. Figure 2 shows a particular run of the model against the implementation using the online algorithm of Spec Explorer where a conformance violation is detected. In this case the `Message` domain was configured to contain two messages “hi” and “bye”.

5. EVALUATION

We evaluate the use of online testing on a number of sample problems. The different case studies are summarized in Table 1. In each case the model size reflects approximately the number of



<Set{}, 0>
Create()/c0
<Set{c0}, 0>
Create()/c1
<Set{c1, c0}, 0>
Enter(c0)
<Set{c1, c0}, 1>
Enter(c1)
Map{c1:>Map{c0:>Seq{}}, c0:>Map{c1:>Seq{}}}
Post(c0, hi)
?Deliver(hi, c0, c1)
Map{c1:>Map{c0:>Seq{hi}}, c0:>Map{c1:>Seq{}}}
Map{c1:>Map{c0:>Seq{hi}}, c0:>Map{c1:>Seq{}}}
?Deliver(hi, c1, c0)
Post(c1, hi)
?Deliver(hi, c1, c0)
Post(c1, hi)
Map{c1:>Map{c0:>Seq{hi}}, c0:>Map{c1:>Seq{hi}}}
?Deliver(hi, c0, c1)
Post(c0, hi)
Map{c1:>Map{c0:>Seq{}}, c0:>Map{c1:>Seq{hi}}}
Map{c1:>Map{c0:>Seq{}}, c0:>Map{c1:>Seq{hi}}}

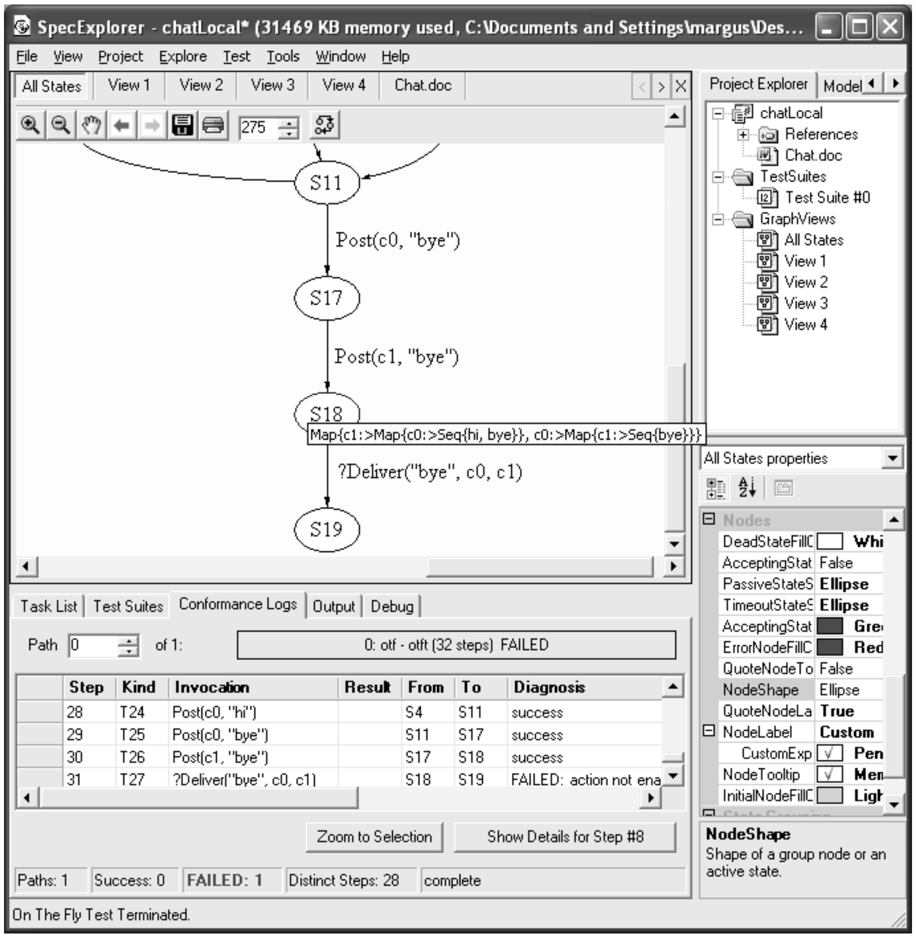
Figure 1: Exploration of the Chat model with two clients, one message "hi", and the filter `NumberOfPendingDeliveries < 2`, generated by Spec Explorer. The label of each node displays the value of the `IgnoreEnteringOrder` property. Active states are shown as ovals and passive states are shown as diamonds. Unlabeled arcs from passive to active state are timeout transitions. Observable actions are prefixed by a question mark.

lines of the model code excluding scenario control. Implementation size reflects approximately the number of implementation code lines that are directly related to the functionality being tested. For example, the full size of the chat server implementation including the test harness is 2000 lines of C# whereas the core functionality of the server that is targeted by the model is just 300 lines of C#. In each case the implementation is either multi-threaded or distributed (in the case of the chat system). The number of threads is the total number of concurrent threads that may be active in the IUT. The number of locks that are use by the implementation is in some cases dependent on the size of the shared resources being protected. For example, the shared bag is implemented by an array of a fixed size and each array position may be individually locked. The number of runs refers to the total number of online test runs starting from the initial state. The number of steps per run is the total number of actions occurring in each test run.

Here is a short evaluation of each of the problems. The first three samples are also available in the sample directory of the Spec Explorer distribution [1]. The last (WP) project is an example of industrial usage of Spec Explorer within Microsoft product teams.

Chat Described in Section 4. Code coverage is not 100% because the functionality that allows clients to exit the session is not modeled. The timeout used is 0, implying that observable actions are not waited for if there is a controllable action enabled in the same state. The Chat Server implementation starts a thread for each delivery that is made, which, in general, violates the local consistency criterion. The bug is discovered with at least two messages being posted by a client. However, the same code coverage is reached already with two clients and a single message. The FSM for this case is illustrated in Figure 1.

Bag A multi-threaded implementation of a bag (multi-set). Several concurrent threads are allowed to add, delete and lookup elements from a shared bag. The example is a variation of a corresponding case study used in [19]. In this case the 85% coverage of the code was already obtained with a single thread and bag with capacity 1. The remaining part of the code is not modeled and consequently not reachable using the model. However, a locking error (missing lock) in the im-



Map{c1:>Map{c0:>Seq{hi, bye}}, c0:>Map{c1:>Seq{bye}}}

Path 0 of 1: 0: otf - otft (32 steps) FAILED

	Step	Kind	Invocation	Result	From	To	Diagnosis
	28	T24	Post(c0, "hi")		S4	S11	success
	29	T25	Post(c0, "bye")		S11	S17	success
	30	T26	Post(c1, "bye")		S17	S18	success
	31	T27	?Deliver("bye", c0, c1)		S18	S19	FAILED: action not ena

Paths: 1 Success: 0 FAILED: 1 Distinct Steps: 28 complete

On The Fly Test Terminated.

Nodes	
DeadStateFillC	Whi
AcceptingStat	False
PassiveStateS	Ellipse
TimeoutStateS	Ellipse
AcceptingStat	Gre
ErrorNodeFillC	Red
QuoteNodeTo	False
NodeShape	Ellipse
QuoteNodeLa	True
NodeLabel	Custom
CustomExp	√ Pen
NodeTooltip	√ Men
InitialNodeFillC	Ligh

NodeShape
Shape of a group node or an active state.

Figure 2: Screenshot of running online testing against the chat server implementation with Spec Explorer. Delivery of message "bye" from client `c0` to client `c1` is observed from state `s18` violating the fifo requirement on message delivery since "hi" from `c0` to `c1` was posted before "bye" but has not been delivered as shown by the tooltip on state `s18` that shows the value of `Members`.

plementation could only be discovered with at least 2 threads and bag with capacity 2. Although the use of 2 threads did not improve code coverage, it gave e.g. full projected state and transition coverage where the projected states were all the possible thread states. In the implementation random small delays were inserted to enforce different schedules of the threads. It was useful for the timeout to be state-based e.g. depending on whether all threads were active or not.

Daisy A model of a distributed file system called Daisy.[2] Roughly 70% of the functionality was modeled, including creation of files and directories, and reading and writing to files. The code coverage measure refers to the whole library including the functionality that was not covered. The model is at a much higher level of abstraction, e.g. nodes and blocks are not modeled. In this case two thirds of the conformance violations that were discovered with multiple users between the model and the implementation were due to modeling or harnessing errors. Same code coverage was reached with a single user. The implementation code was a C# translation of the original case study written in Java. The implementation threads were instrumented with small random delays that helped to produce more interleavings of the different user threads accessing Daisy.

[2]Used as a common case study during the joint CAV/ISSTA 2004 special event on specification, verification, and testing of concurrent software. The event was organized by Shaz Qadeer (`qadeer@microsoft.com`).

WP A system-level object-model (WP1) and a system-level model (WP2) of an interaction protocol between different components of the Windows operating system, used by a Windows test team. WP2 is composed of 7 smaller closely interacting models.

The model-based approach helped to discover 10 times more errors than traditional manual testing, while the effort in developing the model took roughly the same amount of time as developing the manual test cases. 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. Despite the fact that unit testing had already reached 60% code(block) coverage in case of WP2, bugs that were found in the process were shallow. The additional 10% percent was gained by model-based testing. Typically 100% code coverage is not possible due to various reasons, such as features that are cut or intended for future releases and result in dead code from the point of view of the release version under test. More-

Table 1: Online testing sample problems.

Sample problem	Model #lines	IUT #lines	IUT #threads	IUT #locks	IUT block coverage	OLT #runs	OLT #steps per run	OLT timeout
Chat	30	300	#messages	6	90%	10	10	0
Bag	100	200	#clients	bag capacity	85%	10	100	state based 0–500ms
Daisy	200	1500	#clients	#files + #nodes + #blocks	60%	10	200	state based 0–100ms
WP1	200	3500	data dependent	data dependent	100%	100	100	100ms
WP2	2000	20000	data dependent	data dependent	70%	30	100	100ms

over, the model did not cover all of the exceptional behavior of the implementation. However, additional manual tests were only able to increase the code coverage marginally by 1-2%. Using online model-based testing helped to discover deep system-level bugs, for which manual test cases would have been hard to construct. When developing new versions of the code, models need to be adjusted accordingly, but such changes are typically local, whereas manual test cases have to be redesigned and sometimes completely rewritten.

In all the cases above the number on code coverage of the implementation did not reflect any useful measurement on how well the implementation was 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. A demanding task was to correctly instrument the implementation code with commit actions that correspond to observable actions that match the level of abstraction in the model. For example, often conformance violations were discovered due to observable actions being invoked out of order. In order to produce a valid serialization of the observable actions that happen in concurrent threads, the multiplexing technique [9] was used in the test harness of Bag and in the WP projects.

In general, our experience matched that of our users: when our customers discover discrepancies 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. The modeling effort itself had in all cases a major impact during the design phase, a lot of design errors, typically twice as many as the number of bugs discovered afterwards, were discovered early on and avoided during coding.

6. RELATED WORK

Games have been studied extensively during the past years to solve various control and verification problems for open systems. A comprehensive overview on this subject is given in [10], where the game approach is proposed as a general framework for dealing with system refinement and composition. The paper [10] was influential in our work for formulating the testing problem as a refinement between interface automata. The notion of alternating simulation was first introduced in [2].

The basic idea of online/on-the-fly testing is not new. It has been introduced in the context of labeled transition systems using ioco theory [6, 20, 22] and has been implemented in the TorX tool [21]. Ioco theory is a formal testing approach based on labeled transition systems (that are sometimes also called I/O automata). An extension of ioco theory to symbolic transition systems has recently been proposed in [12].

The main difference between alternating simulation and ioco is that the system under test is required to be input-enabled in ioco (inputs are controllable actions), whereas alternating simulation does not require this since enabledness of actions is determined dynamically and is symmetric in both ways. In our context it is often unnatural to assume input completeness of the system under test, e.g. when dealing with objects that have not yet been created – an action on an object can only be enabled when the object actually exists in a given state. Refinement of interface automata also allows the view of testing as a game, and one can separate the concerns of the conformance relation from how you test through different test strategies.

There are other important differences between ioco and our approach. In ioco theory tests can terminate in arbitrary states, and accepting states are not used to terminate tests. In ioco quiescence is used to represent the absence of observable actions in a given state, and quiescence is itself considered as a action. Timeouts actions in Spec Explorer are essentially used to model special observable actions that allow the tool to switch from passive to active mode, and in that sense influence the action selection strategies. Typically a timeout is enabled in a passive state where also other observable actions are enabled (see e.g. Figure 1, where each passive state has two enabled actions, one of which is a timeout), thus timeouts do not, in general, represent absence of other observable actions. State dependency of the timeout function is essential in many applications. In our approach states are full first-order structures from mathematical logic. The update semantics of an action method is given by an abstract state machine (ASM) [15]. The ASM framework provides a solid mathematical foundation to deal with arbitrarily complex states. In particular, we can use state-based expressions to specify action weights, action parameters, and other configurations for OLT. We can also reason about dynamically created object instances, which is essential in testing object-oriented systems. When dealing with objects, interface automata are extended to model automata in [7]. Model automata refinement is alternating simulation where actions are terms that must match modulo object bindings, if a model object is bound to an implementation object then the same model object cannot subsequently (during a later step) be bound to a different implementation object, thus preserving a bijection between objects in the model and objects in the implementation [7]. Support for dynamic object graphs is also present in the Agedis tools [17].

An early version of a model-based online testing algorithm presented here, was implemented in the AsmLT tool [3] (AsmLT is a predecessor of Spec Explorer); in AsmLT accepting states and timeouts are not used. A brief introduction to the Spec Explorer tool is given in [14]. Besides online testing, the main purpose of Spec Explorer is to provide support for model validation and offline test case generation. Test cases are represented in form of finite game strategies [18, 5]. Spec Explorer is being used daily by several Microsoft product groups.

7. OPEN PROBLEMS & FUTURE WORK

There are a number of open problems in testing large, reactive systems. Here is a list of problems that we have encountered, and that are also widely recognized in the testing community.

Achieving and measuring coverage. In the case of external nondeterminism it is difficult to predict the possible behaviors and what part of the state space is being covered in future runs.

Scenario control. What is a convenient language or notation for generating strategies that obtain particular behaviors? This is related to playing games with very large or even infinite state spaces, where at every point in time there is only a limited amount of knowledge available about the history.

Failure analysis. Understanding the cause of a failure after a long test run is related to a similar problem in the context of model-checking.

Failure reproduction. Obtaining short repro cases for failures that have been detected after long runs is an important practical issue. This is complicated by the fact that the reproduction of failures may not always be possible due to external nondeterminism.

Failure avoidance. A tester running online testing in a stress-testing mode against an IUT often wants to continue running the tool even after a failure is found. Of course the same failure should, if possible, be avoided in continued testing.

Some of these problems can be recast as problems of test strategy generation in the game-based sense. For this a unifying formal testing theory based on games and first-order states seems promising. We are currently working on several of these items, in particular scenario control. We are also extending the work started in [5] to online testing using Markov decision theory for optimal strategy generation from finite approximations (called test graphs) of the model program.

Acknowledgment. We thank Wolfgang Grieskamp for many valuable discussions and for his contributions to Spec Explorer. We also thank Pushmeet Kohli who during his iternship in our group contributed to the implementation of the online testing algorithm in Spec Explorer.

8. REFERENCES

[1] Spec Explorer. URL:http://research.microsoft.com/specexplorer, released January 2005.

[2] R. Alur, T. A. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 163–178, 1998.

[3] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In Petrenko and Ulrich, editors, *Formal Approaches to Software Testing, FATES 2003*, volume 2931 of *LNCS*, pages 264–280. Springer, 2003.

[4] M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: An overview. In M. Huisman, editor, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[5] A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. Technical Report MSR-TR-2005-04, Microsoft Research, January 2005.

[6] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 187–193. Springer, 2001.

[7] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-based testing of object-oriented reactive systems with Spec Explorer. Tech. Rep. MSR-TR-2005-59, Microsoft Research, 2005.

[8] 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, Créteil, France*, pages 119–130, 2005.

[9] C. Campbell, M. Veanes, J. Huo, and A. Petrenko. Multiplexing of partially ordered events. In F. Khendek and R. Dssouli, editors, *17th IFIP International Conference on Testing of Communicating Systems, TestCom 2005*, volume 3502 of *LNCS*, pages 97–110. Springer, 2005.

[10] 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.

[11] 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.

[12] L. Franzen, J. Tretmans, and T. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *Proceedings of the Workshop on Formal Approaches to Software Testing (FATES 2004)*, pages 3–17, Linz, Austria, September 2004. To appear in *LNCS*.

[13] 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.

[14] W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 2004. In press, available online.

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

[16] Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 2005. Preliminary version available as Microsoft Research Technical Report MSR-TR-2004-27.

[17] A. Hartman and K. Nagin. Model driven testing - AGEDIS architecture interfaces and tools. In *1st European Conference on Model Driven Software Engineering*, pages 1–11, Nuremberg, Germany, December 2003.

[18] 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.

[19] S. Tasiran and S. Qadeer. Runtime refinement checking of concurrent data structures. *Electronic Notes in Theoretical Computer Science*, 113:163–179, January 2005. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).

[20] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999.

[21] J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.

[22] M. van der Bij, A. Rensink, and J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing: Third International Workshop, FATES 2003*, volume 2931 of *LNCS*, pages 86–100. Springer, 2004.