On-The-Fly Testing of Reactive Systems[⋆]

Margus Veanes[1], Colin Campbell[1], Wolfram Schulte[1], and Pushmeet Kohli[2⋆⋆]

[1] Microsoft Research, Redmond, WA, USA
margus,colin,schulte@microsoft.com
[2] Oxford Brookes University, Oxford, UK
pushmeet.kohli@brookes.ac.uk

Abstract. On-the-fly testing is a technique in which test derivation from a model program and test execution are combined into a single algorithm. It can also be called online testing using a model program, to distinguish it from offline test generation as a separate process. We describe a practical on-the-fly 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 a high level specification language 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, and testing amounts to a game between the test tool and the implementation under test.

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 a new on-the-fly technique for testing reactive systems. In our 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 with the benefits of online state exploration as part of a game strategy.

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 refinement. We discovered that several

[⋆] Microsoft Research Technical Report MSR-TR-2005-05, January 2005. This work would not have been possible without the contributions by Wolfgang Grieskamp and Nikolai Tillmann.

[⋆⋆] The author contributed to this work during his summer 2004 internship at Microsoft Research.

communities have been working more or less independently on closely related problems in this area and attempt to bring them together.

Finally, we address in a practical way the scalability issues of testing the large state spaces that arise from dynamic object graphs and output nondeterminism of the system under test.

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], is in daily use by several product groups inside of Microsoft. Our on-the-fly technique has been used in an industrial setting to test operating system components and Web service infrastructure.

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 show the algorithm for on-the-fly testing. We then present in Section 4 a concrete example that runs in the Spec Explorer tool. Finally, we discuss related work.

2 Specifying reactive systems using model programs

To describe the behavior of a reactive system, we use the notion of interface automata [8, 7] following the exposition in [7].

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 *input actions* A^{i} and *output actions* A^{o}.
- *Enabling functions* Γ^{i} and Γ^{o} from S to subsets of A^{i} 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 $A^{\mathrm{i}}_M \cup A^{\mathrm{o}}_M$ of all actions in M, and we let $\Gamma_M(s)$ denote the set $\Gamma^{\mathrm{i}}_M(s) \cup \Gamma^{\mathrm{o}}_M(s)$ of all *enabled* actions in a state s. We say that an action *a transitions from s to t* if $\delta_M(s, a) = t$.

2.1 Model program as interface automaton

Model programs in Spec Explorer are written in a high level specification language AsmL [13] or Spec# [4]. A model program P is an implicit definition of a transition system by using a finite collection of *guarded update rules*. A guarded update rule in P is defined as a parameterized method using Spec# (or AsmL), similar to the way methods are written in normal programming languages like C#. The execution of a single step of a guarded update rule is an ASM step [12], for the ASM semantics of the core of AsmL see [10]. A guarded update rule defined by a method is called an *action method*.

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. Typically the methods of a model program P create objects and use unbounded data structures, like integers, strings, sets, maps, etc., in which case M is infinite. The program declares a finite vocabulary of model *variables*. A model *state* is a mapping of those variables to concrete values.[3] The set of initial states S_M^{init} of M is the singleton set containing the initial assignment of variables to values as declared in P.

Each action method m, is associated in a state s with a set $\mathit{Enabled}_m(s)$ of enabled actions $\langle m, \boldsymbol{v}\rangle$, where $\boldsymbol{v}$ is a sequence of concrete values matching the signature of m. (In Spec Explorer, the enabling conditions of action methods are defined by using state based parameter domain expressions and method preconditions.) The set of all enabled actions $\mathit{\Gamma}_M(s)$ in a state s is the union of all $\mathit{Enabled}_m(s)$ for all action methods m.

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

The transition function δ_M maps a source state s and an action $a = \langle m, \boldsymbol{v}\rangle$ to a target state t, given that the method call $m(\boldsymbol{v}_{\mathrm{in}})$ in P produces the output parameters $\boldsymbol{v}_{\mathrm{out}}$, produces updates on s that yield the sequel state t, and that a is enabled in s. The set of states S_M is the least set that contains S_M^{init} and is closed under δ_M. The set A_M of actions is the union of all $\mathit{\Gamma}_M(s)$ for s in S_M.

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 programs of a model program are disjointly partitioned into *controllable* and *observable* ones. This induces, for each state s, a corresponding partitioning of $\mathit{\Gamma}_M(s)$ into controllable (output) actions $\mathit{\Gamma}_M^{\mathrm{o}}(s)$ enabled in s, and observable (input) actions $\mathit{\Gamma}_M^{\mathrm{i}}(s)$ enabled in s. The action set A_M is partitioned accordingly into A_M^{i} and A_M^{o}. A state s where $\mathit{\Gamma}_M^{\mathrm{i}}(s)$ is empty is called *stable*; s is called *unstable* otherwise. A state s where $\mathit{\Gamma}_M(s)$ is empty is called *terminal*.

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

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

acquire shared resources that are later released. The 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. Typically accepting states are stable. 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 *reset* action in A_M and a *goal* state g in S_M, such that, for all accepting states s, $\delta_M(s, \mathit{reset}) = g$. In IUT, the *reset* 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 *reset* 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 managed .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 multithreaded 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. 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 sence the IUT is an open system.

The program of the IUT, typically written in C# (that is a proper subset of Spec#), is 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 *reset* 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 an 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 [7]. 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.

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

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 $A^{\mathrm{i}}_M \subseteq A^{\mathrm{i}}_N$, 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 $(s,t) \in \rho$.

We say that an IUT *conforms* to a model P if M_P refines M_{IUT}. 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 in 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 that is not necessarily complete.

The user partitions the action methods into observable and controllable ones. Method level binding of observable action is provided automatically through instrumentation of the IUT at the binary (MSIL) level. During execution, IUT makes callbacks to the conformance engine of the tool through such bindings. 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.

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

An observable action $a = \langle m, \boldsymbol{v} \rangle$ happens as a spontaneous reaction from the IUT. To determine enabledness of a in the model the following steps are taken. First, the precondition of the method call $m(\boldsymbol{v}_{\mathrm{in}})$ is checked in P. If the precondition does not hold, a is not enabled in the model and a precondition failure occurs. Second, $m(\boldsymbol{v}_{\mathrm{in}})$ is executed in the model yielding either a conformance failure in form of a model invariant or postcondition failure (i.e. a is not enabled in the model), or the invocation returns $\boldsymbol{w}$ as the expected output parameters. If $\boldsymbol{v}_{\mathrm{out}} \neq \boldsymbol{w}$ then a is not enabled in the model, resulting in an unexpected return value (or output parameter) conformance failure. If a is enabled in the model then the model transitions from its current state s to $\delta_{M_P}(a, s)$.

3 On-The-Fly testing with Spec Explorer

On-the-fly testing is a technique in which test derivation from a model program and test execution are combined into a single algorithm. It can also be called *online* testing using a model program, to distinguish it from *offline* test generation as a separate process. By generating test cases at run time, rather than precomputing 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 on-the-fly testing algorithm (OTF) uses a (dynamically changing) strategy to select controllable actions. OTF 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 OTF depends on various user configurable settings. The most important ones are timeouts and action weights. Before explaining the algorithm we introduce the OTF settings 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 OTF 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 depend on 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 [16], e.g., when a sufficiently large time span value is associated with a stable state.

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 OTF.

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 $A^{\mathrm{i}}_{M^{\mathrm{t}}} = A^{\mathrm{i}}_M \cup \{\sigma\}$, where σ is called a *timeout event.*
- Observable actions are only enbled if *timeout* is false:

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

  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: $\delta_{M^{\mathrm{t}}}(\sigma, s^F) = s^T$.

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

3.2 Action weights

Action weights are used to configure the strategy of OTF 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 OTF.

– A *per-state weight function* is a function ω^{s} that maps a state s in M_S and a controllable action method m to a non-negative integer.
– A *decrementing weight function* is a function ω^{d} of the OTF algorithm that maps a controllable action method m to the value $\max(\omega_m^{\mathrm{init}} - \#(m), 0)$, where ω_m^{init} 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 the all the controllable action methods enabled in s, i.e. in $\Gamma_M^{\mathrm{o}}(s)$, 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 On-The-Fly testing algorithm

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

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

A *controller strategy* (or *output strategy*) π maps a state $s \in S_{\mathrm{OTF}}$ to a subset of $\Gamma^{\mathrm{o}}_M(s{\restriction}M)$. A *controller step* is a pair (s,t) of OTF states such that $t{\restriction}M = \delta_M(s{\restriction}M, \langle m, \boldsymbol{v}\rangle)$ for some action $\langle m, \boldsymbol{v}\rangle$ in $\pi(s)$, and $\#(m)^t = \#(m)^s + 1$. In general, OTF 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 OTF. 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 OTF states such that $t{\restriction}M = \delta_M(s{\restriction}M, a)$ for some $a \in \Gamma^{\mathrm{i}}_M(s{\restriction}M)$, and $\#^s = \#^t$.

By a *test run* of OTF we mean a trace $\boldsymbol{s} = s_0 s_1 \ldots s_k \in S^{+}_{\mathrm{OTF}}$ 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 OTF algorithm. We write s_{OTF} for the current state of OTF. We say that an action a is *legal* (in the current state) if a is enabled in $s_{\mathrm{OTF}}{\restriction}M$, a is *illegal* otherwise. Initially s_{OTF} is the initial state of OTF. The following steps are repeated subject to additional termination conditions (see following section):

Step 1 (observe) Assume s_{OTF} is an observation state (*timeout* is false). OTF waits for an observable action from the IUT until $\Delta(s_{\mathrm{OTF}}{\restriction}M)$ 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 terminates with a *FAILURE* verdict.
2. If a is legal, OTF makes an observable step (s_{OTF}, s) and sets s_{OTF} to s. OTF continues from Step 1.

If no observable action happened, OTF makes an observable step by setting *timeout* to true.

Step 2 (control) Assume s_{OTF} is a control state (*timeout* is true). Assume $\pi(s_{\mathrm{OTF}}) \neq \emptyset$. OTF chooses an action $a \in \pi(s_{\mathrm{OTF}})$, such that the probability of the method of a being m is $\mathit{prob}(s_{\mathrm{OTF}}{\restriction}M, m)$, and invokes a in the IUT. There are two cases:
1. If a is not enabled in IUT, the test run terminates with a *FAILURE* verdict.
2. If a is enabled in IUT, OTF makes a controllable step (s_{OTF}, s), where s is an observation state, and sets s_{OTF} to s. OTF continues from Step 1.

Step 3 (terminate) Assume $\pi(s_{\mathrm{OTF}}) = \emptyset$. If $s_{\mathrm{OTF}}{\restriction}M$ is the goal state then the test run terminates with a *SUCCESS* verdict, otherwise the test run terminates with a *FAILURE* verdict.

Notice that the timeout event in Step 1 happens immediately if $\Delta(s_{\mathrm{OTF}}{\restriction}M) = 0$. 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_{OTF} and, upon failure, a transition $\delta_M(s_{\mathrm{OTF}}, \mathit{fail}) = s_{\mathrm{OTF}}$, such that *fail* is never enabled in M_{IUT}. For example, if a timeout happens in a nonaccepting state and the subsequent control 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. A cleanup action is a controllable action that has been marked by the user as an action that will help drive the system to an accepting state. For example, actions like closing a file or aborting a transition may be labeled as cleanup actions, whereas actions like opening a new file or starting a new task would not be so labeled by the user.

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.

Overview The chat system lets clients enter or exit the chat session. Clients that have entered the session are called *members* of the chat session and may post messages. The chat system forwards each post, in the order received, to all members of the chat session. The system may forward the current message to members in any order. The specification prescribes that all members must receive the current message before any member receives the next queued message.

System State The state of the system consists of four global variables:

```
class Client {}

static Set<Client> members    = Set{};
static Set<Client> nonmembers = Set{};
static Seq<string> messages   = Seq{};
static Set<Client> recipients = Set{};
```

State is defined as the tuple `<members, nonmembers, messages, recipients>`. Each instance of the class `Client` is a distinct abstract value. `Set` and `Seq` types denote sets and sequences of values. `Set{}` denotes the empty set; `Seq{}` denotes the empty sequence.

Acccepting state condition The method `IsAcceptingState()` is true for states that may terminate a test sequence. A valid run always ends in a state where there are no entries in the queue of pending messages and all clients have exited the chat session.

```
static bool IsAcceptingState() {
  return members.IsEmpty && messages.IsEmpty;
}
```

Controllable Actions There are four controllable actions in the chat system, see Figure 1. Controllability and observability of actions are attached to the respective methods by using .NET attributes. An action is enabled if the Boolean expressions given by the `requires` clauses are true with respect to the actual parameters and the values of the state variables in the current state.

```
[Action] static Client Create()
  requires true;
{
  Client c = new Client();
  nonmembers += Set{c};
  return c;
}
[Action] static void Enter(Client c)
  requires c in nonmembers;
{
  members += Set{c};
  nonmembers -= Set{c};
}
[Action] static void Exit(Client c)
  requires c in members;
{
  nonmembers += Set{c};
  members -= Set{c};
}
[Action] static void Post(Client c, string content)
  requires c in members;
{
  if (messages.IsEmpty) recipients = members;
  messages += Seq{content};
}
```

Fig. 1. Controllable actions of chat system example.

`Create` It is always possible to create new clients. A new client is initially added to the set of nonmember clients.

`Enter` A client that is not already a member of the session may join. The variables `members` and `nonmembers` reflect the change in status.

`Exit` An existing member of the session may leave it.

`Post` A client may post a new message for all to receive, provided it is member of the session. The state is updated to include a new message at the end of the queue of pending messages. If the queue of pending messages is empty, then the new message becomes the current message whose `recipients` will be the current members.

Observable Actions There are two observable actions, see Figure 2.

```
[Action(Kind=ActionAttributeKind.Observable)]
static void Received(Client c, string msg)
  requires !messages.IsEmpty && msg == messages.Head;
  requires c in recipients;
{
  recipients -= Set{c};
}
[Action(Kind=ActionAttributeKind.Observable)]
static void Finished(string msg)
  requires !messages.IsEmpty && msg == messages.Head;
  requires (recipients * members).IsEmpty;
{
   messages = messages.Tail;
   recipients = messages.IsEmpty ? Set{} : members;
}
```

Fig. 2. Observable actions of chat system example.

`Received` This may be observed only when the system has a posted message to forward and only for a client that is in the set of expected recipients for this message. The method is marked as an observable action; think of it as a notification callback that occurs whenever the chat system forwards a particular message to a particular client. After we observe this transition, we update the state by removing the client from the set of clients that may receive this message. This reflects the constraint of the specification that messages are delivered just one time per recipient.

`Finished` This transition may be observed when there are no more recipients (that haven't already exited the chat session) for the current message. The operator `*` is set intersection.

By exploring the chat model program for a maximum of one Client instance, one message input value ("hello") and a maximum message queue length of one, we get an interface automaton as shown in Figure 3.

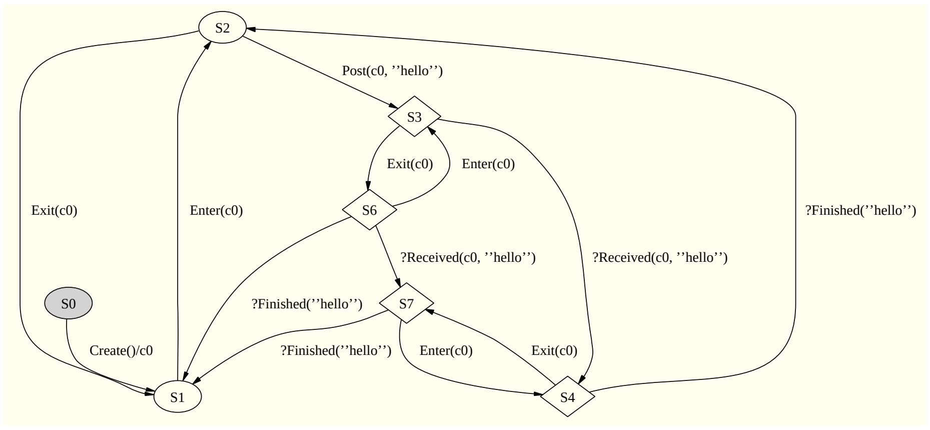

Fig. 3. Partial interface automaton view of the chat model, generated by SpecExplorer. Diamods represent unstable states and observable actions are prefixed by '?'.

Configuration 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 configure the test. First, we have to establish conformance bindings, which are isomorphic mappings between the signature of the model program, henceforth simply called P, and IUT. Next, we provide parameters for the `Post` call, here we just use the strings "hello" and "goodby". We also restrict the number of clients to be at most 100 (if we wouldn't restrict it at all, we might have a garbage collection problem, since we remember all generated clients). We do not give any weights, which means that each operation has equal probability to being chosen.

Execution Our methodology also requires that objects in the model that are passed as input arguments, must have a 1:1 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 on-the-fly algorithm showed a number of conformance discrepancies with respect to a TCP/IP-based implementation of this specification written in C#. Our experience matched that of our users: when our customers discover discrepancies using our tool, typically about half originate in the model and only half are due to coding errors in the implementation under test.

5 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 [7], where the game approach is proposed as a general framework for dealing with system refinement and composition. The paper [7] 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 on-the-fly testing is not new. It has been introduced in the context of labeled transition systems using ioco theory [6, 16, 18] that is implemented in the TorX tool [17]. ioco theory is a formal testing approach based on labeled transition systems (that are sometimes also called I/O automata); the main difference to interface automata is the input enabledness requirement on systems that takes a closed view of the system under test. There are other important differences. *Regarding the modeling approach*: In this paper states are full first-order structures from mathematical logic. The update semantics of an action method is given by an abstract state machine [12]. This enables us to use state-based evaluation of expressions to adapt the algorithm for different purposes, e.g. by using state-based action weight expressions, state-based parameter generation for actions, etc. It also allows to reason about dynamically instantiated object instances, which is essential in testing object-oriented systems. Support for dynamic object graphs is also present in the Agedis tools [14]. *Regarding the conformance relation*: In ioco theory tests can terminate in arbitrary states, and accepting states are not part of the theory. Instead of timeouts the ioco theory uses quiescence to represent the absence of observable actions. An extension of ioco theory to symbolic transition systems has recently been proposed in [9].

The main difference between aternating refinement and ioco is that, unlike in alternating refinement, IUT is supposed to accept any controlable action in ioco. This difference boils down to the fundamental difference between alternating refinement used for open systems, versus refinement (trace inclusion) used for closed systems. This difference is discussed at length in [7]. One may argue for the favor of both. We believe that the open system view for testing is more appropriate. It also allows the view of testing as a game with a nice separation of concerns of how you test (strategies) from what you test for (conformance relation). This separation of concerns is for example clear in the on-the-fly algorithm where the use of strategies is orthogonal to the conformance relation checking part. Until now, ioco has only been used in the context of testing, whereas alternating refinement is also of fundamental importance for closely related verification problems.

An early version of a model-based on-the-fly 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 [11]. Besides on-the-fly testing, the main purpose of Spec Explorer is to provide offline test case generation support from model programs, where test cases are represented in form of finite game strategies [15, 5]. Spec Explorer is being used daily by several Microsoft product groups. The tool can be obtained from [1].

6 Ongoing work and open problems

There are a number of open problems in testing large, reactive systems. Among these are achieving and measuring coverage and controlling test scenarios. Some of these can be recast as problems of test strategy in the game-based sense. For this a unifying formal testing theory based on games and first-order state seems promising.

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. In M. Huisman, editor, *Cassis International Workshop, Marseille*, LNCS. Springer, 2004.
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. 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.
8. 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.
9. 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*.
10. U. Glässer, Y. Gurevich, and M. Veanes. Abstract communication model for distributed systems. *IEEE Transactions on Software Engineering*, 30(7):458–472, July 2004.
11. W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 2004. In press, available online.
12. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
13. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 2005. To appear in special issue dedicated to FMCO 2003, preliminary version available as Microsoft Research Technical Report MSR-TR-2004-27.

14. 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.
15. 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.
16. 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.
17. 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.
18. 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.