Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer

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

May 2005

Technical Report
MSR-TR-2005-59

Testing is one of the costliest aspects of commercial software development. Model-based testing is a promising approach addressing these deficits. At Microsoft, model-based testing technology developed by the Foundations of Software Engineering group in Microsoft Research has been used since 2003. The second generation of this tool set, Spec Explorer, deployed in 2004, is now used on a daily basis by Microsoft product groups for testing operating system components, .NET framework components and other areas. This paper provides a comprehensive survey of the concepts of the tool and their foundations.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Contents

1 Introduction

Testing is one of the costliest aspects of commercial software development. Not only laborious and expensive, it also often lacks systematic engineering methodology, clear semantics and adequate tool support.

Model-based testing is one of the most promising approaches for addressing these deficits. At Microsoft, model-based testing technology developed by the Foundations of Software Engineering group in Microsoft Research has been used internally since

2003 [18, 6]. The second generation of this tool set, Spec Explorer [1], deployed in 2004, is now used on a daily basis by Microsoft product groups for testing operating system components, .NET framework components and other areas. While we can refer the reader to papers [19, 27, 12, 11, 33] that describe some aspects of Spec Explorer, this paper provides a comprehensive survey of the tool and its foundations.

Spec Explorer is a tool for testing reactive, object-oriented software systems. The inputs and outputs of such systems can be abstractly viewed as parameterized action labels, that is, as invocations of methods with dynamically created object instances and other complex data structures as parameters and return values. Thus, inputs and outputs are more than just atomic data-type values, like integers. From the tester's perspective, the system under test is controlled by invoking methods on objects and other runtime values and monitored by observing invocations of other methods. This is similar to the "invocation and call back" and "event processing" metaphors familiar to most programmers. The outputs of reactive systems may be unsolicited, for example, as in the case of event notifications.

Reactive systems are inherently nondeterministic. No single agent (component, thread, network node, etc.) controls all state transitions. Network delay, thread scheduling and other external factors can influence the system's behavior. In addition, a system's specification may leave some choices open for the implementer. In these cases, the freedom given to the implementer may be interpreted as nondeterminism, even if a given version of the system doesn't exploit the full range of permitted behavior. Spec Explorer handles nondeterminism by distinguishing between *controllable* actions invoked by the tester and *observable* actions that are outside of the tester's control.

Reactive systems may be "large" in terms of the number of possible actions they support and the number of runtime states they entail. They can even have an unbounded number of states, for example, when dynamically instantiated objects are involved. Spec Explorer handles infinite states spaces by separating the description of the *model* state space which may be infinite and finitizations provided by *user scenarios* and *test cases*.

The following sections provide a detailed overview of Spec Explorer foundations.

Section 2 introduces the methodology used by Spec Explorer with a small example, a distributed chat server. The system's behavior is described by a *model program* written in the language Spec# [2], an extension of C#. A model program defines the state variables and update rules of an *abstract state machine* [21]. The states of the machine are first-order structures that capture a snapshot of variable values in each step. The machine's steps (i.e., the transitions between states) are invocations of the model program's methods that satisfy the given state-based *preconditions*. The tool *explores* the machine's states and transitions with techniques similar to those of explicit state model checkers. This process results in a finite graph that is a representative subset of model states and transitions. Spec Explorer provides powerful means for visualizing the results of exploration. Finally, Spec Explorer produces test cases for the explored behavior that may be run against the system under test to check the consistency of actual and predicted behavior.

Subsequent sections give a more in-depth look at the semantic foundations of Spec Explorer. In Section 3, we introduce *model automata*, an extension of interface automata over states that are first-order structures. The basic conformance notion, *alter-*

nating simulation [4, 14], is derived from interface automata. Model automata also include the concept of *accepting states* familiar in formal languages, which characterize those states in which a test run is conclusive. Model automata include states and transitions, but they extend traditional model-based testing by admitting open systems whose transitions are not just a subset of the specification's transitions and by treating states as first-order structures of mathematical logic.

Section 4 gives techniques for *scenario control*, in cases where the model describes a larger state space than the tester wants to cover. Scenario control is achieved by *method restriction*, *state filtering*, *state grouping* and *directed search*. This section also introduces the exploration algorithm.

Section 5 describes our techniques for test generation. Traditionally, test generation and test execution are seen as two independent phases, where the first generates an artifact, called the *test suite*, that is then interpreted by the second phase, test execution. We call this traditional case *offline testing*. However, test generation and test execution can be also folded in one process, where the immediate result of test execution is used to prune the generation process. This we call *online testing* (also called "on-the-fly" testing in the literature). Online testing is particularly useful for reactive systems with large state spaces where deriving an exhaustive test suite is not feasible. In the testing framework presented here, both the online case and the offline case are viewed as special cases of the same general testing process. In the offline case the input to the test execution engine (discussed in Section 6) is a test suite in form of a model automaton of a particular form. In the online case the input to the test execution engine is a dynamic unfolding of the model program itself, i.e. the test suite has not been explicitly precomputed.

Section 6 discusses the conformance relation (alternating refinement) that is used during both online and offline testing. We address the problem of harnessing a distributed system with an observationally complete "wrapper" and of creating bindings between abstract entities (such as object identities) found in the model and the system under test.

The paper closes with a survey of related work in Section 7 and a discussion of open problems in Section 8.

2 A Sample: Chat

To illustrate the basic concepts and the methodology of Spec Explorer, we look at a simple example: a distributed chat system. We will also refer back to this sample to illustrate points made in later sections.

The chat system is a distributed, reactive system with an arbitrary number of clients. Each client may post text messages that will be delivered by the system to all other clients that have entered the chat session. The system delivers pending messages in FIFO order with local consistency. In other words, a client always receives messages from any given sender in the order sent. However, if there are multiple senders, the messages may be interleaved arbitrarily.

Figure 1 shows the Spec# model of the chat system. The model consists of a class that represents the abstract state and operations of a client of a chat session. Each in-

```
class Client {
  bool entered;
  Map<Client,Seq<string>> unreceivedMsgs;

  [Action] Client() {
    this.unreceivedMsgs = Map;
    foreach (Client c in enumof(Client), c != this){
      c.unreceivedMsgs[this] = Seq{};
      this.unreceivedMsgs[c] = Seq{};
    }
    entered = false;
  }

  [Action] void Enter()
    requires !entered; {
    entered = true;
  }

  [Action] void Send(string message)
    requires entered; {
    foreach (Client c in enumof(Client), c != this, c.entered)
      c.unreceivedMsgs[this] += Seq{message};
  }

  [Action(Kind=ActionAttributeKind.Observable)]
  void Receive(Client sender, string message)
    requires sender != this &&
           unreceivedMsgs[sender].Length > 0 &&
           unreceivedMsgs[sender].Head == message; {
    unreceivedMsgs[sender] = unreceivedMsgs[sender].Tail;
  }
}
```

Figure 1: Model program written in Spec# specifying the possible behavior of a chat system. The `Map` and `Seq` data types are special high-level value types of Spec# that provide convenient notations like display and comprehensions (`Seq{}` denotes the empty sequence). The `Action` attribute indicates that a method is an action of the abstract state machine given by the model program. The **enumof**`(T)` form denotes the set of instances of type `T` that exist in the current state. The **requires** keyword introduces a method precondition.

stance of the class will contain two variables. The variable `entered` records whether the client instance has entered the session. A mapping `unreceivedMsgs` maintains separate queues for messages that have been sent by other clients but not yet received by this client. Messages in the queues are “in flight”. Note that the model program isn’t an example implementation. No client instance of an implementation could be expected to maintain queues of messages it has not yet received! Not surprisingly, modeling the expected behavior of a distributed system is easier than implementing it.

We model four actions:

- The `Client` constructor creates an instance of a new client. The state of the system after the constructor has been invoked will include empty message queues between the new client and all previously created client instances. These queues can be thought of as virtual one-way “channels” between each pair of client instances. There will $n(n-1)$ queues in the system overall if there are n clients.

- The `Enter` action advances the client into a state where it has entered the chat session. A Boolean-valued flag is enough to record this change of state.
- The `Send` action appends a new message to the queues of unreceived messages in all other clients which have entered the session.
- The `Receive` method extracts a message sent from a given sender from the sender's queue in the client.

Typically the terms “input” and “output” are used either relative to the model or relative to the system. To avoid possible confusion, we use the following terminology: The `Send` *action* is said to be *controllable* because it can be invoked by a user to provide system input. The `Receive` action is *observable*; it is an output message from the system.

For a model like in Figure 1, Spec Explorer extracts a representative behavior according to user-defined parameters for scenario control. To do this Spec Explorer uses a state exploration algorithm that informally works as follows: In a given model state (starting with the initial state) determine those invocations – action/parameter combinations – which are *enabled* by their preconditions in that state; compute successor states for each invocation; repeat until there are no more states and invocations to explore.

The parameters used for the invocations are provided by parameter generators which are state dependent; if in a given state the parameter set is empty, the action will not be considered. Default generators are selected automatically (for example, for objects the default parameter generator delivers the **enumof**`(T)` collection). Enabledness is determined by the precondition of the method. Besides of the choice of parameters, the exploration can be pruned by various other scenario control techniques (see Section 4).

Figure 2 shows a scenario extracted from the chat model as a model automaton (cf. Section 3). State filters restrict the number of clients and avoid the case where the same message is sent twice by a client. The `message` parameter of the `Send` method is restricted to the value `"hi"`. Additional method restrictions avoid sending any messages before both the two clients have been created and entered the session (cf. Section 4 for a discussion of scenario control).

The nodes of the graph in Figure 2 represent distinct states of the system as a whole. The arcs are transitions that change the system state. Each state in the graph is either *passive* or *active*. Ovals represent active states where a client may give the system new work to do. Diamonds represent passive states where the client may wait for an action from the system or transition into an active state after a state-dependent timeout occurs.

We say that a model and an implementation under test (IUT) *conform* if the following conditions are met: The IUT must be able to perform all transitions outgoing from an active state. The IUT must produce no transitions other than those outgoing from a passive state. Every test must terminate in an accepting state. In other words, these conditions mean that the system being tested must accept all inputs provided by the tester and only produce outputs that are expected by the tester. Further, to prevent unresponsive systems from passing the test, tests must end in an expected final state.

Note that in some passive states there is a race between what the tester may do and what the system may do. The timeout transition, here represented by a transition

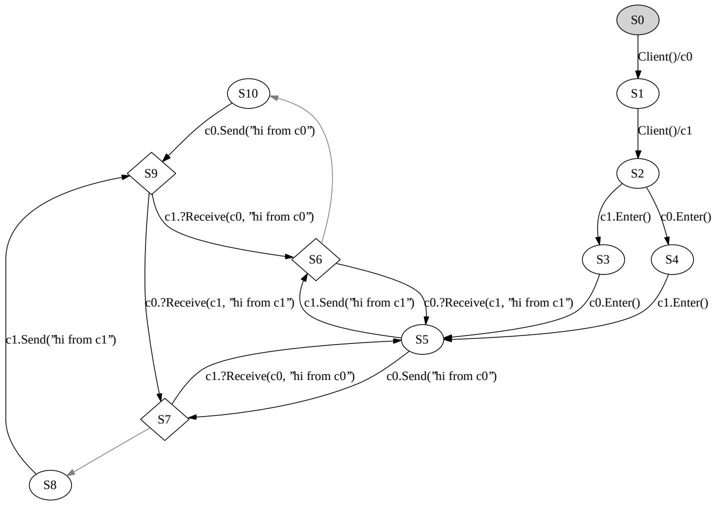

Figure 2: A model automaton of a scenario, extracted from the chat model program and visualized by Spec Explorer, with two clients (c0 and c1) and a fixed message send by each client (“hi from ...”). The initial state is shown in gray. Actions labels prefixed by “?” indicate observable actions. Labels without prefix indicate controllable actions. Active states (where no observable actions are expected) are shown as ovals. Passive states (where the tester may observe system actions) are shown as diamonds. The unlabeled transitions represent an internal transition (“timeout”) from passive to active.

with no label, indicates that an internal transition from a passive state to an active state occurred without observing output from the system. In other words, nothing was observed in the time the tester was willing to wait (cf. Section 5).

The scenario shown in Figure 2 does not yet reveal the chat system’s desired property of local consistency, i.e. preserving the ordering of messages from one client. For that scenario we need at least three clients, where one of them posts at least two messages. In this case we should observe that the ordering of the two messages is preserved from the receiver’s point of view, regardless of any interleaving with other messages. Figure 3 illustrates local consistency. The full model automaton underlying this view has too many transitions to be amenable for depiction in this context. However, the projected view (where some states are grouped together) shows that there is no path where the “bye” message of client c0 is received before the “hi” message of the same client.

The actual conformance testing of an implementation based on model automata as described above happens either *offline* or *online* (“on-the-fly”) in Spec Explorer. For offline testing, the model automaton is reduced to an automaton that represents a test suite and can then be compiled to a stand-alone program. The resulting program encodes the complete oracle as provided by the model. For online testing, model exploration and conformance testing are merged into one algorithm. If the system-under-test is a non-distributed .NET program, then all test harnessing will be provided automatically by the tool. In other cases, the user has to write a wrapper in a .NET language which encapsulates the actual implementation, using .NET’s interoperability features. This technique has been used to test a very wide variety of systems, including distributed

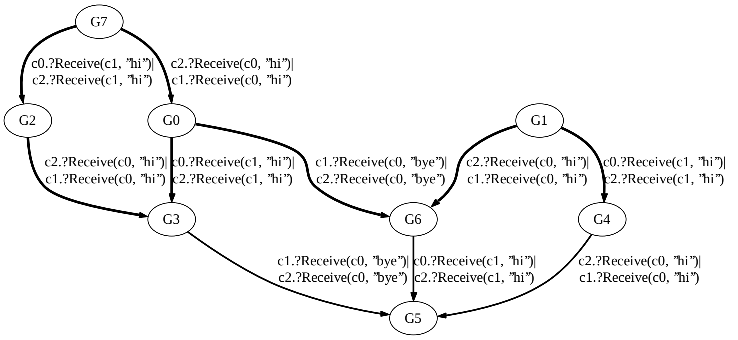

Figure 3: A *projection* on a model automaton extracted and visualized by Spec Explorer for a scenario with three clients, where client c0 sends two messages (“hi” and “bye”) in order, client c1 sends one message (“hi”), and client c2 does not send any message. The projection groups states equivalent under a user defined grouping function into one node, and in the visualization merges arcs between these nodes. In this case, the range of the grouping function is a mapping of clients to the sequences of messages which have been sent by that client but not yet received by *all* the other clients. This visualization also hides all transitions with actions different from `Receive`, and hides loops on grouped states.

systems and components that run in the OS kernel.

3 Model programs and model automata

In this section we describe the semantic framework on which Spec Explorer is built upon. We introduce the notion of *model automata* as an extension of interface automata [15, 14] over first-order structures. Instead of the terms “input” and “output” that are used in [14] 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.

3.1 States

A *state* is a first-order structure over a vocabulary Σ; its universe of values is denoted by $\mathcal{U}$ and is assumed to be fixed for all states. The vocabulary symbols are *function symbols*; each function symbol has a fixed arity and is given an interpretation or meaning in s. The interpretation of some function symbols in Σ may change from state to state. The function symbols whose interpretation may change are called *state variables* or dynamic functions [21]. The set of state variables is denoted by $\mathcal{V}$. Any ground term over $\Sigma - \mathcal{V}$ has the same interpretation in all states. As far as this paper is concerned, all state variables are either nullary or unary. A *dynamic universe* C is a dynamic unary Boolean function; we say that o *is in* C if $C(o) = \mathit{true}$.

We assume that $\mathcal{U}$ contains a distinguished element for representing undefinedness, denoted by `undef`. All functions in Σ have total interpretations in $\mathcal{U}$, mapping to the `undef` value as required. Note that we do not assume that `undef` can be denoted by a symbol in Σ.

Given a first-order structure s over Σ and a vocabulary $V \subseteq \Sigma$, we use the notation $s{\restriction}V$ for the reduct of s to the vocabulary V. We write $S{\restriction}V$ for $\{s{\restriction}V : s \in S\}$.

By a *state based expression* E we mean a term over Σ that may contain (logical) variables, i.e. placeholders for values. If E contains no (logical) variables, E is said to be *ground* or *closed*. If E contains variables all of which are among $\mathbf{x} = x_1, \ldots, x_n$ we indicate this by $E[\mathbf{x}]$, and given closed terms $\mathbf{v} = v_1, \ldots, v_n$ over $\mathcal{F} \cup \mathcal{V}$ we write $E[\mathbf{v}]$ for the closed expression after substituting or replacing each x_i in E by v_i for $1 \leq i \leq n$. The value of a closed state based expression E in a state s is denoted by $E(s)$. We write $E(S)$ for $\{E(s) : s \in S\}$, where s is a set of states.

3.2 Model automata

Definition 1 An *model automaton* M has the following components:

- A set S of *states* over a *vocabulary* Σ, where Σ contains a finite sub-vocabulary $\mathcal{V}$ of *state variables* or *dynamic functions*.
- A nonempty subset S^{init} of S called *initial states*.
- A subset S^{acc} of S called *accepting states*.
- A set *Acts* of *actions* that are (ground) terms over $\Sigma - \mathcal{V}$. *Acts* is a disjoint union of *controllable actions Ctrl* and *observable actions Obs*.
- A *transition relation* $\delta \subseteq S \times \mathit{Acts} \times S$

M is *deterministic* if for any state s and action a there is at most one state t such that $(s, a, t) \in \delta$, in which case we write $\delta(s, a) = t$. *In this paper we consider only deterministic model automata.*

When it is clear from the context, we often say automata for model automata. Notice that actions have the same interpretation in all states, this will allow us later to relate actions in different states in a uniform way. Intuitively, a state is uniquely defined by an interpretation of the symbols in $\mathcal{V}$.

For a given state $s \in S$, we let $\mathit{Acts}(s)$ denote the set of all actions $a \in \mathit{Acts}$ such that $(s, a, t) \in \delta$ for some t; we say that a is *enabled* in state s. We let $\mathit{Ctrl}(s) = \mathit{Acts}(s) \cap \mathit{Ctrl}$ and $\mathit{Obs}(s) = \mathit{Acts}(s) \cap \mathit{Obs}$;

In order to identify a component of a model automaton M, we sometimes index that component by M, unless M is clear from the context. When convenient, we denote M by the tuple

$$(S^{\mathrm{init}}, S, S^{\mathrm{acc}}, \mathit{Obs}, \mathit{Ctrl}, \delta).$$

We will use the notion of a sub-automaton and a reduct when we define tests in Section 5.1. Tests are special automata that have been expanded with new state variables and actions but that preserve the transitions when the additional state variables and actions are ignored.

Definition 2 A model automaton M is a *sub-automaton* of a model automaton N, in symbols $M \subseteq N$, if $S_M \subseteq S_N$, $S_M^{\mathrm{init}} \subseteq S_N^{\mathrm{init}}$, $S_M^{\mathrm{acc}} \subseteq S_N^{\mathrm{acc}}$, $\mathit{Ctrl}_M \subseteq \mathit{Ctrl}_N$, $\mathit{Obs}_M \subseteq \mathit{Obs}_N$, and $\delta_M \subseteq \delta_N$.

We lift reduction on vocabularies, $S{\restriction}V$, to automatons:

Definition 3 Given an automaton M and a vocabulary $V \subseteq \Sigma_M$, we write $T{\restriction}V$ for the following automaton N, called the *reduct* of M to V:

- $S_N = S_M{\restriction}V$, $S_N^{\mathrm{init}} = S_M^{\mathrm{init}}{\restriction}V$, $S_N^{\mathrm{acc}} = S_M^{\mathrm{acc}}{\restriction}V$,
- $Acts_N$ is the set of all a in $Acts_M$ such that a is a term over V,
- $\delta_N = \{(s{\restriction}V, a, t{\restriction}V) : (s,a,t) \in \delta_M, a \in Acts_N\}$.

M is called an *expansion* of N.

A reduct of an automaton to a subset of the state variables may collapse several states into a single state, which is illustrated later. Therefore, projection does not always preserve determinism. In this paper, projections are used in a limited way so that the resulting automaton is always deterministic.

3.3 Model programs

A model program P declares a finite set $\mathcal{M}$ of action methods and a set of state variables $\mathcal{V}$. A state of P is given by the values (or interpretations) of the state vocabulary symbols Σ that occur in the model program. The value of a state variable in $\mathcal{V} \subseteq \Sigma$ may change as a result of program execution. Examples of function symbols whose interpretation does *not* change are built-in operators, data constructors, etc. A nullary state variable is a normal (static) program variable that may be updated. A unary state variable represents either an instance field of a class (by mapping from object identities to the value of that field) or a dynamic universe of objects that have been created during program execution.

Each action method m, with variables $\mathbf{x}$ as its formal input parameters, is associated with a state based Boolean expression $Pre_m[\mathbf{x}]$ called the *precondition* of m. The execution of m in a given state s and with given actual parameters $\mathbf{v}$, produces a sequel state where some of the state variables have changed. In general the execution of m may also do other things, such as write to an external file, or prompt up a dialog box to a user, but abstractly we consider m as an *update rule* that is a function that given a state and actual parameters for m that satisfy the precondition of m, produces a new state t where some state variables in $\mathcal{V}$ have been updated.

A model program can be written in a high level specification language such as AsmL [22] or Spec# [7], or in a programming language such as C# or VB. A guarded update rule in P is defined as a parameterized method, similar to the way methods are written in a normal program. A guarded update rule defined by a method is called an *action method*.

The model automaton M_P defined by a model program P is a complete unwinding of P as defined below. We omit the subscript P from M_P when it is clear from the context. Since model programs deal with rich data structures, states are not just abstract entities without internal structure, but full first-order structures. We define actions and the state to state transition function δ_M that represents execution of actions. Unlike an explicit transition system with given sets of nodes and arcs, the states and transitions of a model program must be *deduced* by executing sequences of atomic actions starting in the initial state. For this reason, we use the term *exploration* to refer to the process of producing δ_M.

The set of initial states S^{init} is the singleton set containing the state with the initial values of state variables as declared in P. The set of all states S is the least set that contains S_M^{init} and is closed under the transition relation δ_M defined below.

Example 1 Consider the Chat example. The set $\mathcal{V}$ contains the dynamic universe `Client` for the instances to that type (denoted as **enumof**`(Client)` in Spec#), and a unary dynamic function for the `entered` and `unreceivedMsgs` instance fields. In the initial state of the model, say s_0, there are no elements in `Client`, and the unary functions that represent the instance fields map to `undef`. In addition, Σ contains other function symbols such as the empty sequence, `Seq{}`, the binary Boolean function `in` that in this context checks if a given element is in the domain of a map, etc. All the symbols in $\Sigma - \mathcal{V}$ have the same interpretation or meaning in all states in S_M, whereas the interpretation of symbols in $\mathcal{V}$ may change from state to state.

3.4 State exploration

Actions are not considered as abstract labels but have internal structure. The vocabulary of non-variable symbols $\Sigma - \mathcal{V}$ is divided into the following disjoint sub-vocabularies: a set $\mathcal{F}$ of function symbols for operators and constructors on data, and a set $\mathcal{M}$ of function symbols for methods.

An *action* over $(\mathcal{M}, \mathcal{F})$ is a term $m(v_1, \ldots, v_k)$ where $m \in \mathcal{M}$, $k \geq 0$ is the arity of m, and each v_i is a term over $\mathcal{F}$. Each parameter of m is either an input parameter or an output parameter. We assume that all the input parameters precede all the output parameters in the parameter list of m. When the distinction between input parameters and output parameters is relevant we denote $m(v_1, \ldots, v_k)$ by $m(v_1, \ldots, v_l)/v_{l+1}, \ldots, v_k$, where $v_1, \ldots, v_l$, $l \leq k$, are input parameters. The set of all actions over $(\mathcal{M}, \mathcal{F})$ is denoted by $Acts_{\mathcal{M},\mathcal{F}}$, or simply $Acts$, when $\mathcal{F}$ and $\mathcal{M}$ are clear from the context. Any two terms over $\mathcal{F}$ are equal if and only if they denote the same value in $\mathcal{U}$, and the value of a term over $\mathcal{F}$ is the same for all states in S_M. The symbols in $\mathcal{M}$ have the term interpretation, i.e. $m(\mathbf{v})$ and $m'(\mathbf{v})$ are equal if and only if m and m' are the same symbol and $\mathbf{v}$ and $\mathbf{w}$ are equal.

Given an action $a = m(\mathbf{v})/\mathbf{w}$ and a state s, a is enabled in s (i.e. $a \in Acts_M(s)$) if the following conditions hold:

- $Pre_m[\mathbf{v}]$ is true in s;
- The invocation of $m(\mathbf{v})$ in s yields the output parameters $\mathbf{w}$.

Let $Acts_m(s)$ denote the set of all enabled actions with method m in state s. The set of all enabled actions $Acts_M(s)$ in a state s is the union of all $Acts_m(s)$ for all action methods m; s is called *terminal* if $Acts_M(s)$ is empty. Notice that $Acts_m(s)$ may be infinite if there are infinitely many possible parameters for m. The set $Acts_M$ is the union of all $Acts_M(s)$ for all s in S_M.

Given $a = m(\mathbf{v})/\mathbf{w} \in \mathit{Acts}_M(s)$, we let $\delta_M(s,a)$ be the target state of the invocation $m(\mathbf{v})$. The invocation of $m(\mathbf{v})$ in a state s can be formalized using ASM theory [21]. Formally, a method invocation produces a set of *updates* that assign new values to some state variables that are then applied to s to produce the target state with the updated values. The interested reader should consult [22] for a detailed exposition of the update semantics of AsmL programs, or [17] that includes the update semantics for the core language constructs.

Example 2 To illustrate how exploration works, let us continue from Example 1. We can invoke the `Client` constructor method in state s_0, since the precondition is *true*. This invocation produces an update that adds a new object, say `c0`, to the dynamic universe `Client`. Let s_1 be the resulting state. We have explored the transition $\delta(s_0, \texttt{Client()/c0}) = s_1$.

From state s_1 we can continue exploration by invoking `c0.Enter()`.[1] The precondition $\mathit{Pre}_{\texttt{Enter}}[\texttt{c0}]$ requires that `c0` is a member of `Client` (due to the type declaration) and that `c0.entered` is false. Thus `c0.Enter()` is enabled in s_1. The invocation produces updates on the dynamic unary function for `entered`. Let the new target state be s_2. We have thus explored the transition $\delta(s_1, \texttt{c0.Enter()}) = s_2$.

3.5 Controllable and observable actions

In order to distinguish behavior that can be controlled from behavior that can only be observed, the methods in $\mathcal{M}$ are split into *controllable* and *observable* ones. This induces, for each state s, a corresponding partitioning of $\mathit{Acts}_M(s)$ into controllable actions $\mathit{Ctrl}_M(s)$ and observable actions $\mathit{Obs}_M(s)$ which are enabled in s. The action set Acts_M is partitioned accordingly into Obs_M and Ctrl_M.

Example 3 In the Chat server model we have three controllable action methods `Client`, `Enter`, and `Send`, and we have a single observable action method `Receive`. The reason why `Receive` is observable is that it corresponds to a reaction of the system under test that cannot be controlled by the tester.

In Spec Explorer, observable and controllable actions can either be indicated by attaching corresponding .NET attributes to the methods in the source text of the model program, or by using the actions settings part of the project configuration for the model.

3.6 Accepting states

The model program has an *accepting state condition* that is a closed Boolean state based expression. A state s is an *accepting state* if the accepting state condition is true in s. The notion of accepting states is motivated by the requirement to identify model states where tests are allowed to terminate. This is particularly important when testing distributed or multi-threaded systems, where it is not always feasible to stop the testing process in an arbitrary state, i.e. prior tests must first be finished before new tests can be started. For example, as a result of a controllable action that starts a thread in the

[1] The notation $o.f(\ldots)$ is the same as $f(o,\ldots)$ but provides a more intuitive object-oriented view when o is an object and f a field or a method of o.

IUT, the thread may acquire shared resources that are later released. A successful test should not be finished before the resources have been released.

Formally, there is an implicit controllable *succeed* action and a special terminal *goal* state g in S_M, s.t. for all accepting states s, $\delta_M(s, \mathit{succeed}) = g$. It is assumed that in the IUT the corresponding method call takes the system into a state where no observable actions are enabled. Thus, ending the test in an accepting state, corresponds to choosing the *succeed* action.

In every terminal non-accepting state s there is an implicit controllable *fail* action such that $\delta_M(s, \mathit{fail}) = s$. It is assumed that the corresponding action in the implementation is not enabled in any state. In other words, as will become apparent from the specification relation described below, if a terminal non-accepting model state is reached, the test case fails.

Example 4 A natural accepting state condition in the Chat example is to exclude the initial state and states where pending messages have not yet been received. In such a state there are no observable actions enabled:

```
enumof(Client).Size > 0 &&
Forall{ c in enumof(Client), s in c.unreceivedMsgs.Keys;
        c.unreceivedMsgs[s].Length == 0}
```

3.7 State invariants

The model program may also have *state invariants* associated with it. A state invariant is a closed Boolean state based expression that must hold in all states. The model program *violates* a state invariant φ if φ is false in some state of the model, in which case the model program is not valid. A state invariant is thus a safety condition on the transition function or an axiom on the reachable state space that must always hold.

Example 5 We could add the following state invariant to the Chat example:

```
Forall{ c in enumof(Client); c notin c.unreceivedMsgs.Keys }
```

It says that no client should be considered as a possible recipient of his own messages. This state invariant would be violated, if we had by mistake forgotten the `c != this` condition in the **foreach**-loop in the body of the `Client` method in Figure 1.

Execution of an action is considered to be an atomic step. In Example 5 there are "internal states" that exists during execution of the `Client` action; however, these internal states are not visible in the transition relation and will not be considered for invariant checking by the Spec Explorer tool.

4 Techniques for scenario control

We saw in section 3 how the methods of a model program can be unwound into a model automaton with controllable and observable actions. In typical practice, the model program defines the *operational contract* of the system under test without regard for any particular test purpose. Hence, it is not unusual that a model program may correspond to an automaton with a large or even infinite number of transitions. When this happens we may want to apply techniques for selectively exploring the transitions of the model program. These techniques are ways of limiting the scenarios that will be considered. They allow us to produce automata that are specialized for various *test purposes* or goals that the tester wishes to achieve. This is also a useful technique for analyzing properties of the model, regardless of whether an implementation is available for testing.

In the remainder of this section, we introduce techniques for scenario control used by Spec Explorer. We define each technique as a function that maps a model automaton M into a new automaton M' with the property described. These techniques take advantage of the fact that states are first-order structures that may be queried and classified. The techniques also rely on the fact that the transition labels are structured into action names with parameter lists (terms and symbolic identifiers).

We will describe the following techniques:

- *Parameter selection* limits exploration to a finite but representative set of parameters for the action methods.
- *Method restriction* removes some transitions based on user-provided criteria.
- *State filtering* prunes away states that fail to satisfy a given state-based predicate.
- *Directed search* performs a finite-length walk of transitions with respect to user-provided priorities. States and transitions that are not visited are pruned away. There are several ways that the search may be limited and directed.
- *State grouping* selects representative examples of states from user-provided equivalence classes [18, 11].

4.1 Parameter selection

Using the action signatures of section 3, we define parameter selection in terms of a relation, D, with $(s, m, \mathbf{v}) \in D$ where $s \in S$, $m \in Acts$, and $\mathbf{v}$ are tuples of elements in $\mathcal{F}$ with as many entries as m has input parameters.

The result of applying parameter selection D to M is an automaton M' whose transition relation is a subset of the transition relation of M. A transition $\delta_M(s, m(\mathbf{v})/\mathbf{w}) = t$ of M is included as a transition of M' if $(s, m, \mathbf{v})$ is in D. The initial states of M' are the initial states of M. The states of M' consist of all states that are reachable from an initial state using the transition rules of M'.

Note that if there is no $\mathbf{v}$ such that $(s, m, \mathbf{v}) \in D$, no transition for m will be available in the state s: parameter selection can also prune away actions, and overlaps to that end with method restriction.

Implementation. The Spec Explorer tool provides a user interface for parameter selection with four levels of control. Rather than populate the relation D in advance, the tool uses expressions that encode the choice of parameters and evaluates these expressions on demand.

Defaults Spec Explorer uses the type system of the modeling language as a way to organize default domains for parameter selection. The user may rely upon built-in defaults provided by the tool for each type. For example, all action input parameters of type **bool** will be restricted in all states to the values **true** and **false** by default. Moreover, all input parameters which represent object instances will default to all available instances of the object type in the given state.

Per Type If the tool's built-in parameter domain for a given data type is insufficient, the user may override it. This is done by giving an expression whose evaluation in a each state provides defaults for parameters of the given type.

Per Parameter The user may specify the domain of individual parameters by a state-based expression, overriding defaults associated with the parameter's type. If not otherwise specified, the tool will combine the domains associated with individual parameters of a method (either defined directly with the parameter or with the parameter's type) to build a Cartesian product or a pairwise combination of the parameter domains.

Per Method The user can also define parameter tuples for a given method explicitly by providing a state based expression which delivers a set of tuples. This allows one to express full control over parameter selection, expressing dependencies between individual parameter selections.

Example 6 For the Chat example given in section 2, the `Send` action has an implicit parameter **this** and an explicit parameter `message`. By default, the parameter domain of parameter `this` ranges over all client instances, while `message` ranges over some predefined strings. These domains come from the defaults associated with the types of the parameters, `Client` and **string** respectively. We can change the default by associating the domain `Set{"hi"}` with the parameter `message`. Combined with the default for type `Client`, this would be equivalent to providing explicit parameter tuples with the expression `Set{c` **in** **enumof**`(Client); <c, "hi">}`.

4.2 Method restriction

An action m is said to be *enabled* in state s if the preconditions of m are satisfied. We can limit the scenarios included in our transition system by strengthening the preconditions of m. We call this *method restriction*.

To do this the user may supply a parameterized, state-based expression e as an additional precondition of m. The action's parameters will be substituted in e prior to evaluation.

The result of applying method restriction e to M is an automaton M' whose transition relation is a subset of the transition relation of M. A transition $\delta_M(s, m(\mathbf{v})/\mathbf{w}) = t$ of M is included as a transition of M' if $e[\mathbf{v}](s)$ is `true`. The initial states of M' are the initial states of M. The states of M' consist of all states that are reachable from an initial state using the transition rules of M'.

Example 7 In the Chat sample, we used method restriction to avoid that clients send messages before all configured clients are created and entered the session. To that end, we used an auxiliary type representing the *mode* of the system, which is defined as follows:

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

Mode CurrentMode {
  get {
    if (enumof(Client).Size < 2)
      return Mode.Creating;
    if (Set{c in enumof(Client), !c.entered;c}.Size < 2)
      return Mode.Entering;
    return Mode.Sending;
  }
}
```

Now we can use expressions like `CurrentMode == Mode.Creating` to restrict the enabling of the actions `Client`, `Enter` and `Send` to those states where we want to see them.

Note that in this sample we are only restricting controllable actions. It is usually safe to restrict controllable actions since it is the tester's choice what scenarios should be tested. Restricting observable actions should be avoided, since their occurrence is not under the control of the tester and may result in inconclusive tests.

4.3 State filtering

A state filter is a set S_f of states where $S^{\mathrm{init}} \subseteq S_f$. Applying state filter S_f to automaton M yields M'. A transition $\delta_M(s, m(\mathbf{v})/\mathbf{w}) = t$ of M is included as a transition of M' if $t \in S_f$. The initial states of M' are the initial states of M. The states of M' consist of all states that are reachable from an initial state of M' using the transition rules of M'.

Implementation. Spec Explorer allows the user to specify the set S_f in terms of a state-based expression. A state s is considered to be in S_f if $e(s)$ is `true`.

Example 8 In the Chat sample, we used a state filter to avoid states in which the same message is posted more than once by a given client before it has been received. The filter is given by the expression:

```
  Forall{c in enumof(Client), s in c.unreceivedMsgs.Keys,
         m1 in c.unreceivedMsgs[s], m2 in c.unreceivedMsgs[s]; m1 != m2}
```

This has the effect of pruning away all transitions that result in a state which does not satisfy this expression. Note that in the case of the Chat sample, this filter in combination with the finite parameter selection and finite restriction on the number of created clients makes the extracted scenario finite, since we can only have distinct messages not yet received by clients, and the number of those messages is finite.

```
var $frontier = \{(s,a,t) \mid s \in S^{\text{init}}, (s,a,t) \in \delta\}$
var $included = S^{\text{init}}$
var $\delta' = \varnothing$
while $frontier \neq \varnothing \wedge InBounds$
    choose $(s,a,t) \in frontier$
    $frontier := frontier \setminus \{(s,a,t)\}$
    if $t \in included \vee IncludeTarget(s,a,t)$
        $\delta' := \delta' \cup \{(s,a,t)\}$
        if $t \notin included$
            $frontier := frontier \cup \{(t,a',t') \mid (t,a',t') \in \delta\}$
            $included := included \cup \{t\}$
```

Figure 4: Directed search in Spec Explorer

4.4 Directed search

When the number of states of M is large, it is sometimes useful to produce M' using a stochastic process that traverses (or explores) M incrementally. Bounded, nondeterministic search is a convenient approach. The version used in Spec Explorer allows the user to influence the choice of scenarios by fixing the probability space of the random variables used for selection. Transitions and states are explored until user-provided bound conditions are met, for example, when the maximum number of explored transitions exceeds a fixed limit. Suitably weighted selection criteria influence the kinds of scenarios that will be covered by M'.

For the purposes of exposition, we can assume that the directed search algorithm operates on a model automaton that has already been restricted using the methods described in sections 4.1 through 4.3 above.

The general exploration algorithm is given in Figure 4. It assumes two auxiliary predicates:

- *InBounds* is true if user-given bounds on the number of transitions, the number of states, etc. are satisfied.
- *IncludeTarget*(s,a,t) is true for those transitions (s,a,t) that lead to a desired target state. By default, *IncludeTarget* returns true. (We will see in Section 4.5 an alternative definition.)

In the algorithm the variable *frontier* represents the transitions to be explored and is initially set to all those transitions which start in an initial state. The variable *included* represents those states of M' whose outgoing transitions have been already added to the frontier, and is initially set to the initial states of M. The variable δ' represents the computed transition relation of the subautomaton M'. The algorithm continues exploring as long as the frontier is not empty and the bounds are satisfied. In each iteration step, it selects some transition from the frontier, and updates δ', *included* and *frontier*.

Upon completion of the algorithm, the transitions of M' are the final value of δ'. The initial states of M' are the initial states of M. The states of M' consist of all states that are reachable from an initial state of M' using the transitions of M'. (This will be the same as the final value of *included*.)

The freedom for directing search of this algorithm appears in the **choose** operation. We can affect the outcome by controlling the way in which choice occurs. We consider two mechanisms: *per-state weights* and *action weights*.

Per-state weights prioritize user-specified target states for transitions of controllable actions. The weight of state s is denoted by ω_s. At each step of exploration the probability of choosing a transition whose target state is t is

$$prob(t) = \begin{cases} 0, & \text{if } t \notin T; \\ \omega_t / \sum_{s \in T} \omega_s, & t \in T. \end{cases}$$

where $T = \{t \mid (s, a, t) \in frontier, a \in Ctrl\}$.

As an alternative to per-state weights, we can introduce action weights that prioritize individual transitions.

Let $\omega(s, m, \delta')$ denote the weight of action method m in state s with respect to the current step of the exploration algorithm and the transitions found so far in δ'. If $m_1, \ldots, m_k$ are all the controllable action methods enabled in s, then the probability of an action method m_i being chosen is

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

The state of the exploration algorithm, namely, the set of transitions already selected for inclusion (δ'), may affect an action method's weight. This occurs in the case of *decrementing action weights* where the likelihood of selection decreases with the number of times a method has previously included in δ'. A more detailed exposition of action weights is given in [33].

Implementation. Weights are given in Spec Explorer as state-based expressions that return non-negative integers.

4.5 State grouping

State grouping is a technique for controlling scenarios by selecting representative states with respect to an equivalence class. We use a state-based grouping expression G to express the equivalence relation. If $G(s) = G(t)$ for states s and t, then s and t are of member of the same group under grouping function G. The G-group represented by a state s is the evaluation of G with respect to state s, namely $G(s)$. $S_{/G}$ denotes the set of all G-groups represented by the elements of set S.

State groupings are useful for visualization and analysis (as we saw in Section 2), but they can also be used as a practical way to prune exploration to distinct cases of interest for testing, in particular to avoid exploring symmetric configurations.

For a given model automaton M and a state $s \in S_M$, let $[s]_{G_i}$ denote the set of states which are equivalent under one grouping G_i, i.e. the set $\{s' \mid s' \in S_M, G_i(s') = G_i(s)\}$.

We can limit exploration with respect to state groupings $G_1, \ldots, G_n$ by using state-based expressions the yield the desired number of representatives of each group. Let $B_1, \ldots, B_i$ be state-based *bound* expressions which evaluate to a nonnegative integer for each i, $1 \leq i \leq n$.

Pruning based on state-grouping bounds can be interpreted in the context of the bounded search algorithm shown in Figure 4, if the $\mathit{IncludeTarget}(s, a, t)$ predicate is defined as $\exists(i \in \{1 \ldots k\})\#([t]_{G_i} \cap \mathit{included}) < B_i(t)$. In other words, a newly visited target state is included if there exists at least one state grouping of the target state whose bound has not yet been reached. Note that the effect of pruning with state grouping depends on the strategy used by the exploration algorithm, i.e. the order in which states and transitions are explored.

Implementation. Spec Explorer visualizes a state grouping G of model automaton M as a graph. The nodes of the graph are elements of $S_{/G}$. Arc a is shown between $G(s)$ and $G(t)$ if $(s, a, t) \in \delta_M$.

Figure 3 is a drawing produced by Spec Explorer using this technique.

Example 9 Recall the model automaton for the Chat sample in Figure 2. Here, after two clients have been constructed, two different orders in which the clients enter the session, as well as two different orders in which clients send the message `"hi"` are represented. We might want to abstract from these symmetries for the testing problem at hand. This can be achieved by providing a state grouping expression which *abstracts* from the object identities of the clients:

```
Bag{c in enumof(Client); <c.entered,Bag{<s,m> in c.unreceivedMsgs; m}>}
```

In the resulting model automaton, the scenarios where clients enter in different order and send messages in different order are not distinguished. Note that with n clients there would be $n!$ many orders that are avoided with the grouping. The use of groupings has sometimes an effect similar to partial order reduction in model-checking.

5 Test generation

Model based test generation and test execution are two closely related processes. In one extreme case, which is also the traditional view on test generation, tests are generated in advance from a given specification or model where the purpose of the generated tests is either to provide some kind of coverage of the state space, to reach a state satisfying some particular property, or to generate random walks in the state space. We call this *offline testing* since test execution is a secondary process that takes the pregenerated tests and runs them against an implementation under test to find discrepencies between the behavior of the system under test and the predicted behavior. Tests may include aspects of expected behavior such as expected results, or may be intended just to drive the system under test, with the validation part done separately during test execution. In another extreme, both processes are intertwined into a single process where tests are generated on-the-fly as testing progresses. We call this mode of testing *online testing*, or *on-the-fly testing*.

In the testing framework presented here, both the online case and the offline case are viewed as special cases of a general testing process in the following sense. In the offline case the input to the test execution engine (dicussed in Section 6) is a test suite in form of a model automaton that is pregenerated from the model program. In the online case the input to the test execution engine is a dynamic unfolding of the model program itself, i.e. the test suite has not been explicitly precomputed.

5.1 Test suites an test cases

Let M be a finitization of the automaton M_P of the model program P; M has been computed using techniques described in Section 4. Recall that M is a finite sub-automaton of M_P. A test suite is just another automaton T of a particular kind that has been produced by a traversal of M as discussed in Section 5.2. A *path of T from s_1 to s_n* is a sequence of states $(s_1, s_2, \ldots, s_n)$ in T such that there is a transition from s_i to s_{i+1} in T.

Definition 4 A *test suite* generated from an automaton M is an automaton T such that:

1. The states in T may use new state variables called *test variables*, i.e. $\mathcal{V}_M \subseteq \mathcal{V}_T$.
2. The set of action methods $\mathcal{M}_T$ of T contains a new controllable action (method) *Observe* of arity 0 and a new observable action (method) *Timeout* of arity 0 that are not in Σ_M. *Observe* and *Timeout* are called *test actions* and corresponding transitions in T are called *test transitions*. For any test transition $\delta_T(s,a) = t$, $s{\upharpoonright}\Sigma_M = t{\upharpoonright}\Sigma_M$.
3. The reduction of T to Σ_M is a sub-automaton of M, i.e. $T{\upharpoonright}\Sigma_M \subseteq M$.
4. An accepting state is reachable from every state in S_T.
5. For all nonterminal states $s \in S_T$, either
   (a) s is *active*: $Ctrl_T(s) \neq \varnothing$ and $Obs_T(s) = \varnothing$, or
   (b) s is *passive*: $Obs_T(s) \neq \varnothing$ and $Ctrl_T(s) = \varnothing$.

   The target state of a transition is passive if and only if it is an *Observe*-transition.
6. For all transitions $\delta_T(a,s) = t$, there is no path in T from t to s.

By a *test case* in T we mean the sub-automaton of T that includes a single initial state of T and is closed under δ_T. Given a state $s \in S_T$, $s{\upharpoonright}\Sigma_M$ is called the *corresponding state* of M.

Here is an intuitive explanation for each of the conditions: 1) The use of test variables makes it possible to represent traversals of M, i.e. to record history that distinguishes different occurrences of corresponding states in M. 2) The *Observe* action encodes the decision to wait for an observable action. The *Timeout* action encodes that no other observable action happened. Test actions are not allowed to alter the corresponding state of M. 3) For all states $s \in S_T$, all properties of the corresponding state of M carry over to s. Typically, there may be several initial states in T; all test cases start in the corresponding initial state of M. Moreover, each transition in T, other than a test transition, must correspond to a transition in M. Note that the source and the target of any test transition must correspond to the same state in S_M. 4) It must be possible to end each test case in an accepting state. In particular, each terminal state must correspond to accepting state of M. 5) The strategy of a test, whether to be passive and expect an observable action or to be active and invoke a controllable action is made explicit by the *Observe* action. If several controllable actions are possible in a given active state, one is chosen randomly. 6) The test suite does not loop, i.e. T is a directed asyclic graph (dag). This guarantees termination of a test, either due to a conformance failure or due to reaching a terminal accepting state.

The distinction between a test suite and a single test case will only be relevant during test execution, when the distiction is irrelevant we say that T is a *test*. Note that if M itself satisfies all these properties, M can be considered as a test (that is a single test case because M has a single initial state). A test T is *control deterministic* if for all active states $s \in S_T$, $Ctrl_T(s)$ is a singleton set. A test T is *observationally deterministic* if for all passive states $s \in S_T$, $Obs_T(s)$ is a singleton set. A test is *deterministic* if it is both control deterministic and observationally deterministic.

Given a test T and an active state $s \in S_T$, we write $T(s)$ for a choice of an action $a \in Ctrl_T(s)$.

Implementation In Spec Explorer tests are represented explicitly in the offline case as sets of action sequences called test segments. Test segments are linked together to encode branching with respect to observable actions. In a deterministic test, the segments correspond to test sequences in the traditional sense. Some segments may be used multiple times, there is an additional test variable that records the number of times each segment has been used to guarantee termination of test execution.

Example 10 Consider the following model program P with $\mathcal{M} = \{\mathtt{F}, \mathtt{G}, \mathtt{H}\}$ where all action methods are controllable and have arity 0, and $\mathcal{V} = \{\mathtt{mode}\}$.

```
enum Mode = {A,B,C}
Mode mode = A;
void F() requires mode == A {mode = B;}
void G() requires mode == B {mode = C;}
void H() requires mode == B {mode = C;}
```

Suppose that the accepting state condition is that `mode` is `C`. Consider also a model program P' that is P extended with an action `I` that takes the model back to its initial state:

```
void I() requires mode == C {mode = A;}
```

The full exploration of P (P') yields a finite automaton $M = M_P$ ($M' = P_{P'}$) shown in Figure 5.

States of M are denoted by the value of `mode`. A deterministic test T for M, as illustrated in Figure 5, uses an additional state variable, say n, that represents the "test

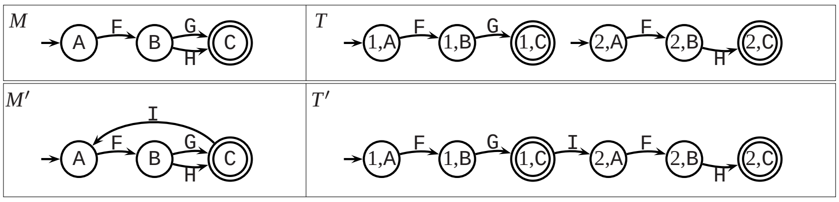

Figure 5: Automaton M (M') for the model program P (P') in Example 10; T is a test for M and M'; T' is a test for M'.

case number" (similarly for T'). Each state of T is labeled by the pair $(n, \texttt{mode})$. In Spec Explorer the test T is represented by the action sequences $(\texttt{F}, \texttt{G})$ and $(\texttt{F}, \texttt{H})$. Note that M itself is a test for M, where $M(\texttt{B})$ is a random choice of $\texttt{G}$ or $\texttt{H}$.

Each $m \in \mathcal{M}$ with formal input parameters $\mathbf{x}$ is associated in Spec Explorer with a positive real valued state based expression $\mathit{Weight}_m[\mathbf{x}]$ whose value by default is 1. The weight of an action $a = m(\mathbf{v})/\mathbf{w}$ in a state s is given by $\mathit{Weight}_m[\mathbf{v}]$. A random choice of an action $a \in \mathit{Ctrl}_T(s)$ in an active state s has probability

$$\frac{\mathit{Weight}_T^s(a)}{\sum_{b \in \mathit{Ctrl}_T(s)} \mathit{Weight}_T^s(b)}.$$

5.2 Traversal algorithms

Given M, a traversal algorithm produces a test suite T for M, for a particular test purpose. A test purpose might be to reach some state satisfying a particular condition, to generate a transition coverage of M, or to just produce a set of random walks. There is an extensive literature on different traversal algorithms from deterministic finite state machines [26], that produce test suites in form of test sequences. When dealing with nondeterministic systems, the game view of testing was initially proposed in [3] and is discussed at length in [34].

In the following we discuss the main definitions of test purposes used in our framework. The definitions can be analyzed separately for control determinisitic and control nondeterminisitic (stochastic) tests. For ease of presentation we limit the discussion to control deterministic tests. We introduce first the following notion.

Definition 5 An *alternating path P of T starting from s* is a tree with *root s*:

- P has no subtrees and is called a *leaf*, or
- P has, for each state $t \in \{\delta_T(s,a) \mid a \in \mathit{Acts}_T(s)\}$ an immediate subtree that is an alternating path of T starting from t.

In the case when T is deterministic, any alternating path is also a path and vice versa. The difference arises in the presence of observational nondeterminism. Intuitively, an alternating path takes into account *all* the possible observable actions in a passive state, whereas a path is just a branch of some alternating path. We say that an alternating path P *reaches* a set S of states if each leaf of P is in S.

Definition 6 Let T be a test for M.

1. Given a subset $S \subseteq S_M$, T *covers* S if $S \subseteq S_T{\restriction}\Sigma_M$.
2. T *covers all transitions* of M if $T{\restriction}\Sigma_M = M$.
3. Given a subset $S \subseteq S_M$, T *can reach* S if there is a path from some initial state of T to a state t such that $t{\restriction}\Sigma_M \in S$.
4. Given a subset $S \subseteq S_M$, T *is guaranteed to reach* S if, for some initial state s of T, there is an alternating path of T from s to a state t such that $t{\restriction}\Sigma_M \in S$.
5. Given a grouping G for M, T *covers* G, if $G(S_M) = G(S_T{\restriction}\Sigma_M)$.

For active tests, the definitions are more or less standard, and execution of a test case produces the given coverage. In the case of reactive systems with observable actions, assumptions have to be made about fairness and the probabilities of the observable actions. In the general case, we have extended the Chinese postman tour algorithm to nondeterministic systems. For alternating reachability a version of Dijkstra's shortest path can be extended to alternating paths, that is used, if possible, to generate tests that are guaranteed to reach a a set of states. Both algorithms are discussed in [27]. For computing test cases that optimize the expected cost, where the system under test is viewed as nature, algorithms from Markov decision process theory can be adapted [9].

Implementation In Spec Explorer the algorithms discussed in [27] have been implemented for the purposes of reaching a set of states and for state and transition coverage. In Spec Explorer the desired set of states is always specified by a state based expression. Each action is associated with a cost and a weight using a state based expression as well. Weights are used to calculate action probabilities. Spec explorer uses the value iteration algorithm for negative Markov decision problems, that is discussed in [9], to generate tests that optimize the expected cost where the observable actions are given probabilities.

For certain traversal algorithms in Spec Explorer, such as a random walks, the tests are limited to a maximum number of steps. Once the maximum number has been reached the test continues with a shortest path to an accepting state. For other test purposes, such as transition coverage, one may need to reexceute some of the test segments in order to observe different observable actions from the same underlying state. In such cases there is a limit on the number of tries related to each segment that always limits each test to a finite number of steps. In those cases the tests are not explicitly represented as automata but implicitly by a program that produces the tests dynamically (during test execution) from the segments generated from M.

Example 11 Consider M in Example 10. The test T for M that is illustrated in Figure 5 covers all transitions of M.

5.3 Online test generation

Rather than using pregenerated tests, in *online* testing or *on-the-fly* testing, test cases are created dynamically as testing proceeds. Online testing uses to the model program "as is". The online technique was motivated by problems that we observed while testing large-scale commercial systems; it has been used in an industrial setting to test operating system components and Web service infrastructure.

We provide here a high level description of the basic OTF algorithm as a transformation of M. Given a model progam P let $M = M_P$. OTF is a transformation of M that given a desired number of test cases n and a desired number k of steps in each test case, produces a test suite T for M. The OTF transformation is done lazily during test execution. We present here the mathematical definition of T as a nondeterministic unfolding of M, where the choices of observable actions reflect the observable actions that take place during test execution. The choices of controllable actions are random selections of actions made by the OTF algorithm. It is assumed that an accepting state is reachable from every state of M.

T has the test variables *TestCaseNr*, *StepNr* that hold integer values, and a Boolean test variable *active*. OTF produces test cases T_i with $\mathit{TestCaseNr} = i$ for $1 \leq i \leq n$.

Each T_i is an unfolding of M produced by the following nondeterministic algorithm. Consider a fixed T_i. Let s_0 be the initial state of M. We let s denote the current state of M. Initially $s = s_0$, $\mathit{StepNr} = 0$, and $\mathit{active} = \mathit{true}$.

The following steps are repeated first until $\mathit{StepNr} = k$, or s is a terminal state, and after that until s is an accepting state.

1. Assume $\mathit{active} = \mathit{true}$. If $\mathit{Ctrl}_M(s) \neq \emptyset$ and $\mathit{Obs}_M(s) \neq \emptyset$ choose randomly to do either either (1a) or (1b), else if $\mathit{Ctrl}_M(s) \neq \emptyset$ do (1a), otherwise do (1b).

   (a) Choose randomly a controllable action $a \in \mathit{Ctrl}_M(s)$, let $t = \delta_M(a, s)$, and let a be the only action enabled in the current state and let $\mathit{StepNr} := \mathit{StepNr} + 1, \mathit{active} := \mathit{true}, s := t$.

   (b) Let *Observe* be the only action enabled in the current state and switch to passive mode $\mathit{active} := \mathit{false}$.

2. Assume $\mathit{active} = \mathit{false}$. All actions in $\mathit{Obs}_M(s)$ and *Timeout* are enabled in the current state, no controllable actions are enbaled. Choose nondeterministically an action $a \in \mathit{Obs}_M(s) \cup \{\mathit{Timeout}\}$, let $\mathit{StepNr} := \mathit{StepNr} + 1$, $\mathit{active} := \mathit{true}$, and if $a \neq \mathit{Timeout}$ let $s := \delta_M(a, s)$.

A single test case in T is formally an unfolding of M from the initial state in form of an alternating path from the initial state, all of whose leaves are accepting states and either the length of each branch is at least k, and ends in a first encounter of an accepting state, or the length of the branch is less than k and ends in a terminal accepting state. Since T is created during execution, only a single path is created for each test case that includes the actual observable actions that happened.

Implementation The implementaion of OTF in Spec Explorer uses the model program P. Action weights are used in the manner explained in Example 10, to select controllable actions from among a set of actions for which parameters have been generated from state based parameter generators. Besides state based weights one can also associate *decrementing weights* with action methods. Then the likelihood of selection decreases with the number of times a method has previously been used, i.e. the weight expression depends on the test variables. The OTF algorithm is dicussed in more detail in [33].

6 Test execution

We discuss here the conformance relation that is used during testing. The testing process assumes that the implementation under test is encapsulated in an observationally complete “wrapper”, that is discussed first. This is needed in order to be able to guarantee termination of test execution. We then discuss how object bindings between the implementation and the model world are maintained and how these bindings affect the conformance relation. Next, the conformance relation is defined formally as an extension of alternating simulation. Finally, we discuss how action bindings are checked and how the conformance engine works.

6.1 Observational completness of implementation under test

The actual implementation under test may be a distributed system consisting of subsystems, a (multithreaded) API, a GUI, etc. We think of the behavior of the IUT as an automaton I that provides an interleaved view of the behavior of the subsystems if there are several of them. The implementation uses the same set of function symbols $\mathcal{F}$ and action methods $\mathcal{M}$ as the model. Values are interpreted in the same universe $\mathcal{U}$. For testability, the IUT is assumed to have a wrapper N that provides an *observationally complete* view of the actual behavior in the following sense:

1. The action method vocabulary $\mathcal{M}_N$ of N is $\mathcal{M}$ extended with the test actions *Observe* and *Timeout* used in tests.

2. The reduct of N to $\Sigma_{M_{\mathrm{IUT}}}$ is M_{IUT}.

3. For each state $s \in S_N$, $\mathit{Observe} \in \mathit{Ctrl}_N(s)$ and, given $t = \delta_N(s, \mathit{Observe})$, $\mathit{Obs}_N(t) \neq \emptyset$, and $\mathit{Obs}_I(t) \subseteq \mathit{Obs}_N(t) \subseteq \mathit{Obs}_I(t) \cup \{\mathit{Timeout}\}$.

4. Only *Observe* transitions to a state of N where observable actions are enabled.

Implementation The timeout action is approximated by using a state based expression that determines the amount of time to wait for observable actions from I to occur. In general, the timeout may not necessarily indicate absence of other actions, e.g. if the waiting time is too short.

6.2 Object bindings

The universe $\mathcal{U}$ includes an infinite subuniverse $\mathcal{O}$ of *objects*. Each object $o \in \mathcal{O}$ has a name and any two distinct object names in a given automaton denote distinct objects in

$\mathcal{O}$. An automaton M and an automaton N may use distinct objects in their actions. We want to compare the executions of M and N modulo a partial isomorphism from the set of objects used in M to the set of objects used in N. The isomorphism is partial in the sense that it only relates objects that have been encountered in actions. The isomorphism between objects extends naturally to arbitrary values in the so called background universe that includes maps, sets, sequences, etc. The theory of background is worked out in detail in [8], where objects are called reserve elements.

By an *object binding function* σ from M to N we mean a partial injective (one-to-one) function over $\mathcal{O}$ that induces a partial isomorphism, also denoted by σ, from actions in M to actions in N. Given an action a of M, we write $\sigma(a)$ or $a\sigma$ for the corresponding action in N. Given that $\sigma(o) = o'$, we say that *o is bound to o' in σ* and denote it by $o \mapsto_\sigma o'$; we omit σ and write $o \mapsto o'$ when σ is clear from the context.

6.3 Refinement of model automata

The refinement relation from a model P_1 to an implementation P_2 is formalized as the refinement relation between the underlying automata

$$M_i = (S_i^{\text{init}}, S_i, S_i^{\text{acc}}, \mathit{Obs}_i, \mathit{Ctrl}_i, \delta_i), \quad \text{for } i \in \{1, 2\}.$$

The following definitions of alternating simulation and refinement for model automata extend the corresponding notions of interface automata as defined in [14]. We denote the universe of finite object binding functions by *Bind*.

Definition 7 An *alternating simulation* from M_1 to M_2 is a relation $\rho \subset S_1 \times \mathit{Bind} \times S_2$ such that, for all $(s, \sigma, t) \in \rho$,

1. for each action $a \in \mathit{Ctrl}_1(s)$ there is a smallest extension θ of σ such that $a\theta \in \mathit{Ctrl}_2(t)$ and $(\delta_1(s, a), \theta, \delta_2(t, a\theta)) \in \rho$;
2. for each action $a \in \mathit{Obs}_2(t)$ there is a smallest extension θ of σ such that $a\theta^{-1} \in \mathit{Obs}_1(s)$ and $(\delta_1(s, a\theta^{-1}), \theta, \delta_2(t, a)) \in \rho$.

The intuition behind alternating simulation is as follows. Consider fixed model and implementation states. The first condition ensures that every controllable action enabled in the model must also be enabled in the implementation modulo object bindings, and that the alternating simulation relation must hold again after transitioning to the target states where the set of object bindings may have been extended for objects that have not been encoutered before. The second condition is symmetrical for observable actions, going in the opposite direction. The role of object bindings is important, if a model object is bound to an implementation object then the same model object cannot subsequently be bound to a different implementation object and vice versa, since that would violate injectivity of an object binding function.

In the special case when no objects are used, it is easy to see that the projection of ρ to states is an alternating simulation from M_1 to M_2 viewed as interface automata, provided that controllable actions are considered as input actions and observable actions are considered as output actions [14]. In general though, alternating simulation with object bindings cannot be reduced to alternating simulation because object bindings are not known in advance and may be different along different paths of execution; this is illustrated with the following example.

Example 12 Consider the chat example. Let $M_1 = M_{\mathtt{Chat}}$ and let M_2 be the automaton of a chat system implementation. Consider the following sequence of transitions in M_1:

$$(s_0, \mathtt{Create()/c1}, s_1), \quad (s_1, \mathtt{Create()/c2}, s_2)$$

In other words, two clients are created one after another. Assume these are the only controllable actions enabled in s_0 and s_1. The same method call in the initial state of the implementation, say t_0 would result in different objects being created each time `Create` is invoked. For example, the following transitions could be possible in the implementation:

$$\begin{array}{ll} (t_0, \mathtt{Create()/d1}, t_1), & (t_1, \mathtt{Create()/d2}, t_2), \\ (t_0, \mathtt{Create()/e1}, t_3), & (t_3, \mathtt{Create()/e2}, t_4), \ldots \end{array}$$

There is an alternating simulation from M_1 to M_2 where `c1` is bound to `d1` and `c2` is bound to `d2` along one possible path, or where `c1` is bound to `e1` and `c2` is bound to `e2` along another path.

Definition 8 A *refinement* from M_1 to M_2 is an alternating simulation from M_1 to M_2 such that $S_1^{\mathrm{init}} \times \{\varnothing\} \times S_2^{\mathrm{init}} \subset \rho$.

A refinement relation is essentially an alternating simulation relation that must hold from all initial states (with no initial object bindings). We say that M_1 *specifies* M_2, or M_2 *conforms to* or *is specified by* M_1, if there exists a refinement from M_1 to M_2. Again, it is easy to see that when there are no objects then the refinement relation reduces essentially to refinement of interface automata as defined in [14]. The following example shows a case when refinement does not hold due to a conflict with object bindings.

Example 13 Let M_1 be as in Example 12, and let M_3 be the automaton of a buggy implementation that in successive calls of `Create` returns the same object that is created initially after the first call. For example the transitions of M_3, where t_0 is the initial state, could be:

$$(t_0, \mathtt{Create()/d1}, t_1), \quad (t_1, \mathtt{Create()/d1}, t_2)$$

Let us try to build up a refinement relation ρ iteratively following definitions 7 and 8. Initially $(s_0, \varnothing, t_0) \in \rho$. After the first iteration,

$$(s_0, \varnothing, t_0), (s_1, \{\mathtt{c1} \mapsto \mathtt{d1}\}, t_1) \in \rho.$$

After another invocation of `Create` from s_1 there are two distinct objects `c1` and `c2` in s_2. It is not possible to further extend ρ, since one would need to extend $\{\mathtt{c1} \mapsto \mathtt{d1}\}$ with the binding $\mathtt{c2} \mapsto \mathtt{d1}$ that would identify two distinct model objects with the same implementation object.

6.4 Checking enabledness of actions

We describe in more detail, given a model automaton $M = M_P$ and an implementation automaton N, a procedure for deciding if an action a of M and an action b of M can be bound by extending a given set of object bindings σ. It is assumed here that the signatures of M and N are such that $\mathcal{F} = \mathcal{F}_M = \mathcal{F}_N$ and $\mathcal{M} = \mathcal{M}_M = \mathcal{M}_N$.

Implementation Spec Explorer provides a mechanism for the user to bind the action methods in the model to methods with matching signatures in the IUT. Abstractly, two methods that are bound correspond to the same element in $\mathcal{M}$.

In the following we describe how, in a given model state s with a given set σ of object bindings, a controllable action $a = m(\mathbf{v})/\mathbf{w}$ is chosen in M and how its enabledness in N is validated.

1. Input parameters $\mathbf{v}$ for m are generated in such a way that the precondition $Pre_m[\mathbf{v}]$ holds in s. All object symbols in $\mathbf{v}$ must already be bound to corresponding implementation objects, otherwise a can not be bound to any implementaion action.

2. The method call $m(\mathbf{v})$ is executed in the model and the method call $m(\mathbf{v}\sigma^{-1})$ is executed in the implementation.

3. The method call in the model produces output parameters $\mathbf{w}$ and the method call in the implementation produces output parameters $\mathbf{w}'$. Values in $\mathbf{w}$ and $\mathbf{w}'$ are compared for equality and σ is extended with new bindings, if an extension is possible without violating injectivity, otherwise the actions cannot be bound.

Conversely, in order to check enabledness of an observable implementation action $a = m(\mathbf{v})/\mathbf{w}$ in the model the following steps are taken.

1. A binding error occurs if there is an implementation object in $\mathbf{v}$ that is not in σ and a corresponding model object cannot be created. If σ can be extended to σ', $Pre_m[\mathbf{v}\sigma']$ is checked in s. If the precondition does not hold, a is not enabled in the model.

2. The method call $m(\mathbf{v}\sigma')$ is executed in the model yielding output parameters $\mathbf{w}'$.

3. This may yield a conformance failure if either σ' cannot be extended or if the values do not match.

Example 14 For example, calling a controllable action a in the model may return the value 1, but IUT throws an exception, resulting in a conformance failure. A binding violation occurs if for example the implementation returns an object that is already bound to a model object, but the model returns a new object or an object that is bound to a different implementation object.

6.5 Conformance automaton

The conformance automaton is a machine that takes a model M, a test T and an observationally complete implementation wrapper N. It executes each test in T against N. The conformance automaton keeps track of the set of object bindings. The following variables are used:

- A variable *verdict*, that may take one of the values *Undecided*, *Succeeded*, *Failed*, *TimedOut*, or *Inconclusive*.
- A set of object bindings β that is initially empty.
- The current state of T, s_T.
- The current state of N, s_N.

For each initial state s_0 of T, the following is done. Let $s_T = s_0$. Let s_N be the initial state of N. The following steps are repeated while $\mathit{verdict} = \mathit{Undecided}$.

Observe: Assume s_T is passive. Observe an action $b \in \mathit{Obs}_N(s_N)$ and let $s_N := \delta_N(b, s_N)$. There are two cases:

1. If β can be extended to β' such that $a = b\beta'^{-1} \in \mathit{Obs}_M(s_T)$ then $\beta := \beta'$.
    (a) If $a \in \mathit{Obs}_T(s_T)$ then $s_T := \delta_T(a, s_T)$.
    (b) Otherwise $\mathit{verdict} := \mathit{Inconclusive}$.
2. Otherwise, if $a = \mathit{Timeout}$ then $\mathit{verdict} := \mathit{TimedOut}$ else $\mathit{verdict} := \mathit{Failed}$.

Control: Assume s_T is active. Let $a = T(s_T)$ and let $s_T := \delta_T(a, s_T)$. There are two cases:

1. If β can be extended to β' such that $b = a\beta' \in \mathit{Ctrl}_N(s_N)$ then $\beta := \beta'$ and $s_N := \delta_N(b, s_N)$.
2. Otherwise $\mathit{verdict} := \mathit{Failed}$.

Finish: Assume s_T is terminal. Let $\mathit{verdict} := \mathit{Succeeded}$

An inconclusive verdict corresponds to the case when the test case has eliminated some possible observable actions, i.e., an observable action happens but the test case does not know how to proceed, although the observable action is enabled in the model. One may consider a class of *complete* tests T such that for each passive state s, $\mathit{Obs}_T(s_T) \supseteq \mathit{Obs}_M(s_T)$, to avoid inconclusive verdicts. The test produced by the OTF transformation is complete in this sense. The *TimedOut* verdict is a violation of the specification from T to N but not a violation of the specification from M to I. However, if the verdict is *Failed* then I does not conform to M, which follows from the assumption that the reduct of T to Σ_M is a subautomaton of M and that the reduct of N to Σ_I is I.

Implementation The inconclusive verdict is currently not implemented in Spec Explorer; it is the testers responsibility to guarantee that the test is complete or to tolerate a failure verdict also for inconclusive tests.

The implementation of the conformance automaton does not know the full state of N. The description given above is an abstract view of the behavior. In particular, the choice of an observable action a in $Obs_N(s_N)$ corresponds to the implementation wrapper producing an action a, which is guaranteed by observational completness of N.

7 Related work

Extension of the FSM-based testing theory to nondeterministic and probabilistic FSMs got some attention a while ago [20, 28, 35]. The use of games for testing is pioneered in [3]. A recent overview of using games in testing is given in [34]. 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 [14], where the game approach is proposed as a general framework for dealing with system refinement and composition. The paper [14] was influential in our work for formulating the testing problem by using alternating simulation of automata. The notion of alternating simulation was first introduced in [4].

Model-based testing allows one to test a software system using a specification (a.k.a. model) of the system under test [6]. There are other model-based testing tools [5, 23, 24, 25, 31]. To the best of our knowledge, Spec Explorer is currently alone in supporting the game approach to testing. Our models are Abstract State Machines [21]. In Spec Explorer, the user writes models in AsmL [22] or in Spec# [7].

The basic idea of online or on-the-fly testing is not new. It has been introduced in the context of labeled transition systems using ioco theory [10, 30, 32] and has been implemented in the TorX tool [31]. 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 [16].

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 smulation 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 model 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 that are encoded in test suites.

There are other important differences between ioco and our approach. In ioco theory tests can in general 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 in Spec Explorer are essentially used to model special observable actions that switch the tester from passive to active mode, and in that sense influence the action selection strategies in tests. Typically a timeout is enabled in a passive state where also other observable actions are enabled, thus timeouts do not, in general, represent absence of other observable actions. 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) [21]. 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 test generation. We can also reason about dynamically created object instances, which is essential in testing object-oriented systems. Support for dynamic object graphs is also present in the Agedis tools [23].

Generating test cases from finite model automata is studied in [27, 9]. Some of the algorithms reduce to solving negative Markov decision problems with the total reward criterion, in particular using value iteration [29] and the result that linear programming yields a unique optimal solution for negative MDPs after eliminating vertices from which the target state is not reachable [13, Theorem 9].

The predecessor of Spec Explorer is the AsmLT tool [6]. In AsmLT accepting states and timeouts were not used. The use of state groupings was first studied in [18] and extended in [11] to multiple groupings.

8 Conclusion

We have presented the concepts and foundations of Spec Explorer, a model-based testing tool which provides a comprehensive solution for the testing of reactive object-oriented software system. Based on an accessible and powerful modeling notation, Spec#, Spec Explorer covers a broad range of problems and solutions in the domain, including dynamic object creation, nondeterminism and reactive behavior, model analysis, offline and online testing and automatic harnessing.

Being used on a daily basis internally at Microsoft, user feedback indeed also indicates that improvements to the approach are necessary. We identify various areas below, some of which we currently tackle in the design and implementation of the next generation of the tool.

Scenario control Scenario control is the major issue where improvements are needed. Currently, scenario control is realized by parameter generators, state filters, method restriction, state grouping, and so on. For some occasions, describing scenario control can be more challenging than describing the functionality of the test oracle. This is partly because of the lack of adequate notations for scenario control, partly because fragments of the scenario control are spread over various places in the model and configuration dialog settings, making it hard to understand which scenarios are captured.

It would be desirable to centralize all scenario control related information as one “aspect” in a single document, which can be reviewed in isolation. Moreover, scenario-oriented notations like use cases would simplify formulating certain kind of scenarios.

We are currently working on an extension of our approach which allows to write scenarios in an arbitrary modeling style, like abstract state machine or use case. The scenario control can be seen as an independent model, which can be reviewed and explored on its own. *Model composition* combines the scenario control model with the functional model.

Model Composition Another important issue identified by our users is model composition. At Microsoft, as often in the industry as a whole, product groups are usually organized in small feature teams, where one developer and one tester are responsible for a particular feature. In this environment it must be possible to model, explore and test features independently. However, for integration testing, the features also need to be tested together. To that end, Spec Explorer users would like to be able to compose compound models from existing models. For the next generation of the tool, we view the scenario control problem as a special instance of the model composition problem.

Symbolic exploration The current Spec Explorer tool requires to use ground data in parameters provided for actions. This restriction is sometimes artificial and required only by underlying technical constraints of the current tool. Consider the Chat sample: it does not really matter which data is send by a client, but only that this same data eventually arrives at the other clients. For the next generation of the tool, we are therefore generalizing exploration to the symbolic case. We will use an exploration infrastructure which connects to an underlying constraint solver.

Measuring coverage and testing success One major problem of model-based testing is developing adequate coverage and test sufficiency metrics. Coverage becomes particularly difficult in the case of internal nondeterminism in the implementation: how can behavioral coverage be achieved for observable actions of the implementation? Testing success is often measured in industry by rates of bug detection; however, model-based testing might show lower bug counts since bugs can be discovered during modeling and resolved before any test is ever run.

Failure analysis and reproduction cases Understanding the cause of a failure after a long test run is related to a similar problem in the context of model-checking. There might be a shorter run which also leads to the error and which should be used as the reproduction case passed to the developer. Moreover, in the case of nondeterministic systems, it is desired to have a reproduction sample which discovers the error reliably with every run. Generating reproduction cases is actually closely related to the problem of online testing, but here we want to drive the IUT into a certain state where a certain error can be discovered. Some of these problems can be recast as problems of test strategy generation in the game-based sense. We are currently extending the work started in [9] to online testing, that uses Markov decision theory for optimal strategy generation from finite approximations of model automata.

Continuing testing after failures If a failure is detected by model-based testing – offline or online – testing can usually not be continued from the failing state, since the model's behavior is not defined for that case. In practice, however, the time from when a bug is discovered and when it is fixed might be rather long, and it should be possible to continue testing even in the presence of bugs. Current practice is to modify scenarios to deal with this problem, but there could be more systematic support for dealing with this standard situation.

References

[1] Spec Explorer tool. `http://research.microsoft.com/specexplorer`, public release January 2005.

[2] Spec# tool. `http://research.microsoft.com/specsharp`, public release March 2005.

[3] R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proc. 27th Ann. ACM Symp. Theory of Computing*, pages 363–372, 1995.

[4] 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.

[5] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In Börger, Gargantini, and Riccobene, editors, *Abstract State Machines 2003*, volume 2589 of *LNCS*, pages 87–107. Springer, 2003.

[6] 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.

[7] 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.

[8] A. Blass and Y. Gurevich. Background, reserve, and Gandy machines. Technical Report MSR-TR-2000-143, Microsoft Research, 2000. In CSL'2000, LNCS Vol. 1862, pages 1–17.

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

[10] 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.

[11] C. Campbell and M. Veanes. State exploration with multiple state groupings. In D. Beauquier, E. Börger, and A. Slissenko, editors, *12th International Workshop on Abstract State Machines, ASM'05, March 8–11, 2005, Laboratory of Algorithms, Complexity and Logic, University Paris 12 – Val de Marne, Créteil, France*, pages 119–130, 2005.

[12] 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.

[13] L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In *International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 66–81. Springer, 1999.

[14] 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.

[15] 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.

[16] 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*.

[17] U. Glässer, Y. Gurevich, and M. Veanes. Abstract communication model for distributed systems. *IEEE Transactions on Software Engineering*, 30(7):458–472, July 2004.

[18] 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.

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

[20] S. Gujiwara and G. V. Bochman. Testing non-deterministic state machines with fault-coverage. In J. Kroon, R. Heijunk, and E. Brinksma, editors, *Protocol Test Systems*, pages 363–372, 1992.

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

[22] 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.

[23] 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.

[24] C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *The Sixth World Conference on Integrated Design and Process Technology, IDPT'02*, Pasadena, California, June 2002.

[25] V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Bourdonov. UniTesK: Model based testing in industrial practice. In *1st European Conference on Model Driven Software Engineering*, pages 55–63, Nuremberg, Germany, December 2003.

[26] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123, Berlin, Aug 1996.

[27] 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.

[28] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *IFIP TC6 9th International Workshop on Testing of Communicating Systems*, pages 125–140. Chapman & Hall, 1996.

[29] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, New York, 1994.

[30] 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.

[31] 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.

[32] 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.

[33] M. Veanes, C. Campbell, W. Schulte, and P. Kohli. On-the-fly testing of reactive systems. Technical Report MSR-TR-2005-03, Microsoft Research, January 2005.

[34] M. Yannakakis. Testing, optimization, and games. In *Proceedings of the Nineteenth Annual IEEE Symposium on Logic In Computer Science, LICS 2004*, pages 78–88. IEEE, 2004.

[35] W. Yi and K. G. Larsen. Testing probabilistic and nondeterministic processes. In *Testing and Verification XII*, pages 347–61. North Holland, 1992.