Action Machines – Towards a Framework for Model Composition, Exploration and Conformance Testing Based on Symbolic Computation

Wolfgang Grieskamp Nikolai Tillmann

May 2005

Technical Report
MSR-TR-2005-60

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

1 Introduction

Testing is one of the most cost-intensive activities in the industrial software development process. Not only is current testing practice laborious and expensive but often also unsystematic, lacks engineering methodology and discipline, and adequate tool support.

Model-based testing is one of the most promising approaches which address these problems. At Microsoft, model-based testing technology developed by our group Foundations of Software Engineering has been applied in the production cycle since around 2003 [1, 2]. The second generation of our tool set, Spec Explorer [3], which has been deployed in 2004, is now used by various product groups inside of the company for testing features of Windows OS core components, XML web service frameworks, and other areas. (See [4, 5] for references which partially address Spec Explorer; a paper which gives a full overview is currently submitted).

Although the basic concepts of Spec Explorer have proved themselves as valid and useful, user feedback indicates that improvements are needed. This paper reports on ongoing work on the third generation of our model-based testing technology which addresses these issues. In particular, we focus on what we call the problem of *scenario control* (selecting particular scenarios from the potentially infinite set of the model's behaviors) and the problem of *model composition*, meaning the combination of models of specific features into compound models. In fact, we view the first problem as a special case of the second one, and come up with a framework of composable models, where the composition operators are those of *product*, *conformance*, and *action refinement*. We give a to our knowledge novel formalization of the problem in the framework of so-called *action machines*, which use symbolic constraints as the composition glue.

Our approach is implemented as a prototype in the larger framework of XRT (eXploring Runtime), a full-fledge exploration engine for CIL (Common Intermediate Language, the byte-code language of .NET) code, which supports symbolic computation (a paper about XRT is currently submitted). Action machines sit on top of XRT as one of other applications.

This paper is organized as follows. We first give a critical review of the current Spec Explorer tool, pinpointing some of the problems users have with the tool. We then introduce the foundation of action machines and their composition operators. We finally give an outline of the implementation in XRT, which, in addition to the mentioned composition operators on action machines, currently provides as means for modeling basic action machines *abstract state machines* [6] and *scenario machines* [4]. The paper concludes with a discussion.

2 Review of Spec Explorer

Spec Explorer is a tool for model-based testing of reactive, object-oriented software systems. These systems take inputs as well as provide outputs, often as spontaneous reactions. Inputs and outputs can not only be atomic data-type values, like integers, but are invocations of methods which take objects and other complex data structures as parameters and deliver them as results. We call those methods which are relevant for the testing problem *actions*. We partition actions in *controllable* actions (the “inputs” we provide to the system) and *observable* actions (the “outputs” the system produces spontaneously).

Model Programs Models are given in Spec Explorer by *model programs*. As the name suggests, model programs are very much like ordinary programs. This distinguishes them from e.g. axiomatic modeling approaches like provided by the Z notation [7]. Model programs differ from ordinary programs in their intended use: they are not concerned about efficiency, and are formulated on the level of abstraction which is adequate for describing the particular problem at hand.

Spec Explorer uses the Spec# language [8] for writing model programs, though in principle any programming language could be used. Spec# is a conservative extension of C#, which adds to C# high-level value data types like sets, maps, sequences and bags with comprehension notations, universal and existential boolean quantifiers, non-deterministic choice, contracts in form of pre-conditions, post-conditions, and invariants, as well as the ability to enumerate over all instances which have been created for a particular object type.

Exploration and Conformance Testing A model program constitutes for Spec Explorer an *abstract state machine* (ASM) [6], where the states are first-order structures capturing the program state, and the steps (transitions) between states are described by those methods which have been depicted as actions.

Spec Explorer *explores* a model-program in order to generate a representative finite subset of the potential infinite behavior of the ASM [1]. The exploration results in a state graph, where the nodes of the graph represent model states, and the edges transitions between states which are labeled by action invocations.

The exploration algorithm roughly works as follows: in a given model state, figure out those invocations (action-parameter combinations) which are *enabled* in that state. Enabledness is determined by the pre-condition of the method, or by the absence of particular exceptions thrown when the method is executed. The parameters used for the invocations are provided by user defined parameter generators, where some default generators are selected automatically (for example, for objects the default parameter generator delivers the set of all instances created for that object type in a given state). Spec Explorer then computes the successor states of all enabled invocations and continues exploration from there. The search can be pruned in various ways: if a state is visited which is equivalent to a state which has been already visited it will not be visited again (where equivalence comes from state identity or from a user-defined equivalence relation); filters on state are applied; upper bounds on the number of transitions and states are evaluated, and so on. The overall process is very similar to an explicit state model-checker; however, whereas a model-checker makes an existential search looking for some path which violates a condition, exploration for testing aims to produce a set of paths which constitute a finite, representative subset of the model.

Note that for models with infinite state spaces (common for models of object-oriented programs) the extraction of a *complete* finite, representative scenario is indeed not decidable (see [1] for a formal analysis), where completeness here means complete coverage of the model. For that reason, Spec Explorer provides powerful means for the test engineer to analyze the result of exploration, for example by visualization. In general, Spec Explorer's methodology is based on a feedback loop between user-configuration for exploration, automatic exploration, and user-analysis of the exploration result.

The actual conformance testing of an implementation now happens either *offline* or *online* ("on-the-fly") in Spec Explorer. For offline testing, the finite state graph is traversed using standard graph traversals to produce a test suite which is persisted as a stand-alone program. The test suite 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 .NET program, then all test harnessing will be provided automatically. In other cases, the user has to write a wrapper in a .NET language which encapsulates the actual implementation, using .NET's interoperability features.

Sample We look at a sample to demonstrate the basic concepts. The *publisher-subscriber* design pattern is commonly used in object-oriented software systems. In this pattern, various subscriber objects can register at a publisher object to receive asynchronous notification callbacks when information is published via the publisher object. Thus this example both includes objects and infinite state space, as well as reactive behavior.

Excerpts of the model are given in Fig. 1, omitting methods for construction of publisher and subscriber objects and for registration and unregistration. The state of the model is given by the instances of a publisher. A publisher has as instance fields the set of registered subscribers, data which is currently delivered to subscribers (or **null** if no data is delivered), and the set of recipients of the data. The `Publish` method is only enabled if currently no data is delivered. This is a *controllable* action, which we provide as input to the system-under-test. The `Handle` method has a more complex pre-condition. It is enabled whenever there exists a publisher such that this subscriber is currently in the delivery set and the current data of the publisher matches the parameter of the `Handle` method. Note that the `Handle` method is an *observable* action, which comes as spontaneous output from the system under test.

Fig 1 also shows an excerpt from the state graph generated by Spec Explorer from this model. In this excerpt, one publisher and two subscribers are configured (the state graph omits the configuration phase). From state `S6`, a `Publish` invocation is fired, leading to state `S7`, which is an *observation* state where the outgoing transitions are observable actions. The meaning of an observation state is that the system under test has a choice to do *any* of the outgoing transitions, as opposed to a control state (`S6`) where it has to accept *all* of the outgoing transitions. Thus, effectively, our model gives freedom to an implementation to process the subscribers of a publisher in any given order.

```
class Publisher {
  Set<Subscriber> subscriptions = Set{};
  object currentData = null;
  Set<Subscriber> deliveries = Set{};
  void Publish(object data)
    requires currentData == null && data != null;
  {
    currentData = data; deliveries = subscriptions;
  }
}
class Subscriber {
  void Handle(object data)
    let pub =
      Choose{p in enumof(Publisher),
                this in p.deliveries;
             p; default null};
    requires pub != null
             && data.Equals(pub.currentData);
  {
    pub.deliveries -= Set{this};
    if (pub.deliveries == Set{})
      pub.currentData = null;
  }
}
```

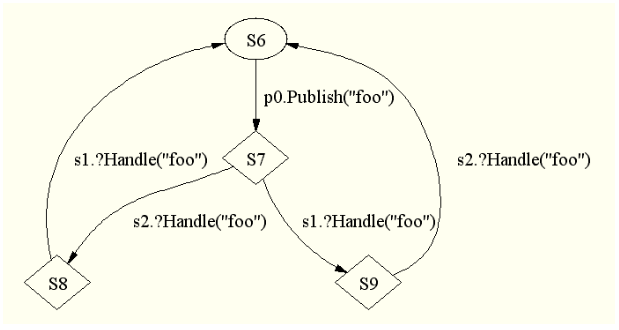

Figure 1: Publisher-Subscriber Model

In order to generate the state graph, we had to annotate the model in Spec Explorer with various information: we specified the parameter passed to the `Publish` method (here, `"foo"`), and restricted the number of publishers and subscribers created, and the order of how creation and registration happens. The problem area of providing this kind of configuration information is called *scenario control*.

Criticism Though Spec Explorer is successfully used at Microsoft for modeling and testing non-trivial features on a daily base, user feedback indicates that major improvements are necessary to use this technology more productively.

Scenario control is the major issue. Currently, scenario control is realized by parameter generators, state filters, additional pre-condition to actions which disable those actions in states where they should not be tested, and so on. Besides messing up the model with information belonging not into the actual model but rather in a separate test purpose description, complex scenario control problems could require non-trivial extensions of the model, maintaining additional state variables and such. A further problem is the need to provide explicit values for parameters (like the `"foo"` value for the `data` parameter of the `Publish` method in Fig. 1). Conceptually, it would be sufficient to use an abstract symbolic value – it doesn't matter which data is published, but only that the published data equals to the handled data. Finally, the way how scenario control is usually achieved in the current Spec Explorer tool spreads fragments of the scenario control information over various places in the model source 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.

Another important issue identified by our users is model composition. At Microsoft, as often in the industry, product groups are usually organized in small feature teams, where often one developer and one tester are responsible for a particular feature. 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.

3 Action Machines

We address the problems identified in the previous section by an improved exploration framework which underlies the forthcoming next generation of Spec Explorer. This framework emphasizes *model composition* and *symbolic exploration.*

Model composition in combination with symbolic exploration adequately solves the scenario control problem. The basic idea is to give a model for scenario control and compose it with the model of the feature. This is in principle not a new idea and has been applied also in the context of finite state machine based testing [9], where the scenario control model is called a test purpose. However, the power of this approach in our setup comes from the availability of symbolic exploration, together with a new modeling style which we introduce: namely scenario models, which capture descriptions of behavior in a use-case oriented style, similar as outlined in [4].

In this section, we will give the formal definition of our framework, which is based on the new notion of *action machines.* We will return to the application of action machines for the scenario control problem in a later section.

3.1 Basic Definitions

We assume an abstract universe of *terms*, $t \in \mathbb{T}$. Terms capture values of the domain of our modeling and implementation languages. Since we are in a symbolic computation world, terms also capture logical variables and term constructors for symbolic operations, e.g. the addition of two symbolic values, or the selection of a field from a symbolic object and a term which represents the state of that symbolic object. The actual structure of terms does not matter for our purposes here.

Over terms, we assume a universe of *constraints*, $c \in \mathbb{C}$. Again, the structure of constraints does not matter here: it can be simple equality constraints or the full predicate calculus. However, we assume that $\mathbb{C}$ has tautologies true and false and is closed under *conjunction* and *implication* (written as $c \wedge c'$ and $c \Rightarrow c'$, respectively), which adhere to the usual laws of boolean algebras. To distinguish the operators of our constraint language from the operators of the meta-logic of our formalization, we use, where necessary, the notation $[\![c \wedge c']\!]$ and $[\![c \Rightarrow c']\!]$. Our constraint universe comes with a decision procedure for checking satisfiability, which is sound (i.e. supports monotonic reasoning) but not necessarily complete. We write $\mathsf{SAT}\ c \in V$, where $V = \{\mathsf{false}, \mathsf{true}, \mathsf{unknown}\}$ is called the verdict.

Let $\mathbb{A}$ be an abstract universe of action names, $a \in \mathbb{A}$. Actions are partitioned into *controlled* actions $\mathbb{A}_C$ and *observed* actions $\mathbb{A}_O$, such that $\mathbb{A} = \mathbb{A}_C \cup \mathbb{A}_O$ and $\mathbb{A}_C \cap \mathbb{A}_O = \varnothing$. Let $\alpha \in \mathbb{I} = \mathbb{A} \times \mathbb{T} \times \mathbb{T}$ be an action invocation, denoted as $\alpha = a(t)/t'$, where t is the input parameter, and t' is the result parameter. For reasons of simplicity, we assume that missing and multiple parameters and results are represented by according terms (e.g. tuples for multiple parameters, and a special term for representing a void result).

An action machine is a given as a tuple

$$\mathbb{M} = (S, T, s)$$

where S is a set of states, $T \subseteq S \times \mathbb{C} \times \mathbb{I} \times S$ is a transition relation, and $s \in S$ is the initial state. Note that we have no assumptions about the internal structure of states. We write $S_{\mathbb{M}}$, $T_{\mathbb{M}}$, $s_{\mathbb{M}}$ for the projections onto components of $\mathbb{M}$, and $s \xrightarrow{c \cdot \alpha} s'$ for a transition $(s, c, \alpha, s') \in T$. A transition indicates that in state s the machine can make a step with invocation α to state s' provided that the constraint c is satisfied.

We lift the transition relation to so-called *environment stepping*. An environment is a constraint. We denote environments with the symbol Γ, $\Gamma \in \mathbb{C}$. In order to deal with the incompleteness of our constraint decision procedure, we distinguish two kinds of environment steps, namely *may* steps and *will* steps, and denote these kinds by $k \in K = \{!, ?\}$, where $k =?$ represents a may-step and $k =!$ a will-step. On step kinds, we have a join operator, written as $k_1 \sqcap k_2$, which results ! if $k_1 = k_2 =!$, and ? otherwise. The fact that an action machine $\mathbb{M}$ makes a step of the kind k in the environment Γ is denoted by

$$\Gamma \wedge c \vdash^{k}_{\mathbb{M}} s \xrightarrow{\alpha} s'$$

which holds iff $s \xrightarrow{c \cdot \alpha} s' \in T_{\mathbb{M}}$ and $\mathsf{SAT}[\![\Gamma \wedge c]\!] \in V$ where $V = \{\mathsf{true}\}$ if $k =!$ and $V = \{\mathsf{true}, \mathsf{unknown}\}$ if $k =?$. Note that, obviously, the will-step relation is subsumed by the may-step relation.

3.2 Composing Action Machines

Basic action machines can result from a variety of sources. They may be constructed from an abstract state machine consisting of guarded update rules, as is the case for Spec Explorer, from a scenario machine, a use case, statechart, temporal logic formula, and even from an actual program. In section 4 we will sketch the basic action machines which we intend to use for the next generation of Spec Explorer.

Whatever the source of a basic action machine is, by adhering to a shared constraint domain and environment as a glue between the machines, we can formalize (and implement!) the composition of those diverse machines. The basic compositions we define in this paper are product machines, conformance machines, and action refinement machines.

Product Machine The product of two action machines results in a machine which steps when both machines step with the same invocation in the same environment. A typical use for product machines is scenario control, where one machine represents the actual model, and the other machine a scenario to which the model shall be restricted.

Let $\mathbb{M}_1 \times \mathbb{M}_2 = (S, T, s)$ denote the product of two action machines, then $S = S_{\mathbb{M}_1} \times S_{\mathbb{M}_2}$, $s = (s_{\mathbb{M}_1}, s_{\mathbb{M}_2})$, and T is a transition relation such that the following rule holds for environment stepping:

$$P1\,\frac{\Gamma \wedge c_1 \vdash^{k_1}_{\mathbb{M}_1} s_1 \xrightarrow{\alpha} s_1' \quad \Gamma \wedge c_2 \vdash^{k_2}_{\mathbb{M}_2} s_2 \xrightarrow{\alpha} s_2'}{\Gamma \wedge c_1 \wedge c_2 \vdash^{k_1 \sqcap k_2}_{\mathbb{M}_1 \times \mathbb{M}_2} (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')}$$

Note that for simplification of presentation, we assume in this and in the following rules that invocations of composed machines match each other. Any kind of unification necessary to achieve this is represented in the constraints.

Conformance Machine The conformance machine represents the behavior of two action machines where the second machine simulates the behavior of the first machine regarding controllable action invocations, and the first machine simulates the second machine regarding observable action invocations. If the alternating simulation is not possible, this machine steps into an error state, indicating a conformance failure. The conformance machine resembles the notion of conformance checking as present in the current Spec Explorer tool, which is closely related to the notion of alternating refinement as defined for interface automata in [10]. Typically, the second machine in this composition represents an implementation; however, we can also think of building chains of conformance of multiple model machines.

Let **error** denote a distinct state for representing conformance failure. $\mathbb{M}_1 \rightsquigarrow \mathbb{M}_2 = (S, T, s)$ denotes the conformance machine, where $S = (S_{\mathbb{M}_1} \times S_{\mathbb{M}_2}) \cup \{\mathbf{error}\}$, $s = (s_{\mathbb{M}_1}, s_{\mathbb{M}_2})$, and T is a transition relation such that the following rules hold for environment stepping (where for all rules, $\alpha = a(t)/t'$ and $(i, j) =$ **if** $a \in \mathbb{A}_C$ **then** $(1, 2)$ **else** $(2, 1)$):

$$C1\;\frac{\begin{array}{c}\Gamma \wedge c_i \vdash^{k}_{\mathbb{M}_i} s_i \xrightarrow{\alpha} s_i' \quad \Gamma \wedge c_j \vdash^{!}_{\mathbb{M}_j} s_j \xrightarrow{\alpha} s_j' \\ \mathsf{SAT}[\![\Gamma \wedge c_i \Rightarrow c_j]\!] = \mathsf{true}\end{array}}{\Gamma \wedge c_i \vdash^{k}_{\mathbb{M}_1 \rightsquigarrow \mathbb{M}_2} (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')}$$

$$C2\;\frac{\begin{array}{c}\Gamma \wedge c_i \vdash^{k}_{\mathbb{M}_i} s_i \xrightarrow{\alpha} s_i' \\ \neg\; \exists\, c_j, s_j' : (\Gamma \wedge c_j \vdash^{!}_{\mathbb{M}_j} s_j \xrightarrow{\alpha} s_j' \wedge \\ \qquad\qquad \mathsf{SAT}[\![\Gamma \wedge c_i \Rightarrow c_j]\!] = \mathsf{true})\end{array}}{\Gamma \wedge c_i \vdash^{!}_{\mathbb{M}_1 \rightsquigarrow \mathbb{M}_2} (s_1, s_2) \xrightarrow{\alpha} \mathbf{error}}$$

We call the machine $\mathbb{M}_i$ the master machine which demands a step, and the machine $\mathbb{M}_j$ the slave machine which must be able to simulate the master's step. Roles of master and slave alternate depending on whether we look at controllable or observable actions. Rule $C1$ describes a successful conformance step: if the master *may* or *will* make a step, then the slave *will* be able to do that step. In addition, the constraint c_j of the slave machine must be implied by the environment and the constraint c_i of the master. Rule $C2$ describes the failure case: if there does not exist a constraint c_j with which the slave will step and which is implied by the environment and the master's constraint, the composed machine steps into the error state.

To understand the intuition behind the requirement that the slave's constraint must be implied by the master's constraint, let us consider an example. Suppose our constraint system is built from interval constraints, such that $x \in l \,..\, u$ denotes that the variable x ranges between a lower bound l and an upper bound u. Let $c_i = [\![x \in 1 \,..\, 2]\!]$ and $c_j = [\![x \in 1 \,..\, 1]\!]$, and $\alpha = a(x)$. Suppose that the master may do the step $\Gamma \wedge c_i \vdash^{?}_{\mathbb{M}_i} s_i \xrightarrow{\alpha} s_i'$, and the slave can do the only step $\Gamma \wedge c_j \vdash^{!}_{\mathbb{M}_j} s_j \xrightarrow{\alpha} s_j'$. The composition of these two machines steps via rule $C2$ into the error state, since $\Gamma \wedge c_i \Rightarrow c_j$ is not satisfiable. Now compare this with the explicit expansion of the range constraint on the parameter x to the action a. Then $\mathbb{M}_i$ can do controllable invocations $a(1)$ and $a(2)$, whereas $\mathbb{M}_j$ can only do $a(1)$, which obviously is a conformance failure. Compare this also with the product machine, $\mathbb{M}_1 \times \mathbb{M}_2$. In that machine, the constraints c_i and c_j would be conjuncted, yielding a machine which can do just the invocation $a(1)$.

Note that our definition of conformance is conservative regarding the treatment of inconclusive solver queries. We require that the slave machine will simulate the master machine for steps even if these steps *may* be only possible in the master. This reflects in the rules in the use of the will-step relation for the slave. As a result, we can have so-called "false negatives", that is conformance failures which are not actually ones, but no "false positives", that is conformance success which is not true. To that end, we are sound but not complete, and have a similar effect as the so-called over-approximation in model-checking. One can also imagine having a less conservative definition of conformance, but this requires further investigation.

Action Refinement Machine The action refinement machine represents the substitution of all invocations of a particular action in one machine (called the super-machine) by the behavior of another machine (called the sub-machine). The gluing is defined by specifying the action which should be substituted in the super-machine and begin and end actions in the sub-machine. Parameters are passed from the action in the super-machine to the begin action in the sub-machine, and the result of the end action is passed back to the super-machine. In the resulting traces of the composed machine, the action of the super-machine will not appear. Action refinement machines are intended to aid the construction of machines which combine models of individual features.

We denote an action refinement machine as $\mathbb{M}_1[a_s \leftarrow_{a_b}^{a_e} \mathbb{M}_2] = (S, T, s)$, where a_s is the substituted action of the super-machine $\mathbb{M}_1$, and a_b and a_e are the begin and end actions of the sub-machine $\mathbb{M}_2$, respectively. Let $\mathsf{susp}{\downarrow}(s,t)$ represent the suspension of the super-machine in some state s, waiting for the result t, and let $\mathsf{susp}{\uparrow}(s)$ represent the suspension of the sub-machine, waiting to be called by the super-machine. The state space of the composed machine is constructed as $S = S_1 \times S_2$ where $S_1 = S_{\mathbb{M}_1} \cup \{\mathsf{susp}{\downarrow}(s,t) : s \in S_{\mathbb{M}_1}, t \in \mathbb{T}\}$ and $S_2 = S_{\mathbb{M}_2} \cup \{\mathsf{susp}{\uparrow}(s) : s \in S_{\mathbb{M}_2}\}$.

The initial state of the action refinement machine is given as $s = (s_{\mathbb{M}_1}, \mathsf{susp}{\uparrow}(s_{\mathbb{M}_2}))$, i.e. the sub-machine is initially suspended. The transition relation T is implied by the following rules for environment stepping, where we abbreviate the action refinement machine with $\mathbb{M} = \mathbb{M}_1[a_s \leftarrow_{a_b}^{a_e} \mathbb{M}_2]$:

$$R1\,\frac{\alpha = a(t)/t_1 \quad a \neq a_s \quad \Gamma \wedge c_1 \vdash_{\mathbb{M}_1}^{k} s_1 \xrightarrow{\alpha} s_1'}{\Gamma \wedge c_1 \vdash_{\mathbb{M}}^{k} (s_1, \mathsf{susp}{\uparrow}(s_2)) \xrightarrow{\alpha} (s_1', \mathsf{susp}{\uparrow}(s_2))}$$

$$R2\,\frac{\begin{array}{c}\alpha_1 = a_s(t)/t_1 \quad \Gamma \wedge c_1 \vdash_{\mathbb{M}_1}^{k_1} s_1 \xrightarrow{\alpha_1} s_1' \\ \alpha_2 = a_b(t)/t_2 \quad \Gamma \wedge c_2 \vdash_{\mathbb{M}_2}^{k_2} s_2 \xrightarrow{\alpha_2} s_2'\end{array}}{\Gamma \wedge c_1 \wedge c_2 \vdash_{\mathbb{M}}^{k_1 \sqcap k_2} (s_1, \mathsf{susp}{\uparrow}(s_2)) \xrightarrow{\alpha_2} (\mathsf{susp}{\downarrow}(s_1', t_1), s_2')}$$

$$R3\,\frac{\alpha = a(t)/t_2 \quad a \neq a_e \quad \Gamma \wedge c_2 \vdash_{\mathbb{M}_2}^{k} s_2 \xrightarrow{\alpha} s_2'}{\Gamma \wedge c_2 \vdash_{\mathbb{M}}^{k} (\mathsf{susp}{\downarrow}(s_1, t_1), s_2) \xrightarrow{\alpha} (\mathsf{susp}{\downarrow}(s_1, t_1), s_2')}$$

$$R4\,\frac{\alpha = a_e(t)/t_1 \quad \Gamma \wedge c_2 \vdash_{\mathbb{M}_2}^{k} s_2 \xrightarrow{\alpha} s_2'}{\Gamma \wedge c_2 \vdash_{\mathbb{M}}^{k} (\mathsf{susp}{\downarrow}(s_1, t_1), s_2) \xrightarrow{\alpha} (s_1, \mathsf{susp}{\uparrow}(s_2'))}$$

Here, rule $R1$ represents stepping of the super machine where the submachine is suspended. Rule $R2$ represents the call of the sub-machine from the super-machine, where the input parameters of the substituted action are passed as the input parameters of the begin action of the sub-machine. The result parameter of the super-machine, t_1, is remembered in its suspension in the resulting state of the composed step. Rule $R3$ describes the case where the sub-machine performs it steps. Finally, rule $R4$ describes the case when the sub-machine returns; here, the result parameter of the sub-machines invocation of a_e must be the stored result t_1 of the result of the substituted step of the super-machine.

3.3 Exploring Action Machines

Action machines can be explored by various means for various purposes. Here we present a class of explorers which do exhaustive exploration using a notion of state subsumption to prune the search. This kind of explorer is also the one which has been experimentally implemented in the XRT framework.

Let $\mathbb{M} = (S, T, s)$ be an action machine. We define state subsumption as a partial ordering on pairs of environments and states, written as $(\Gamma_1, s_1) \sqsubseteq (\Gamma_2, s_2)$, which has the following properties: $\mathsf{SAT}[\![\Gamma_2 \Rightarrow \Gamma_1]\!] = \mathsf{true}$, and for all constraints c_1, c_2 and and invocations α, if $\Gamma_2 \wedge c_2 \vdash^k_{\mathbb{M}} s_2 \xrightarrow{\alpha} s$, then also $\Gamma_1 \wedge c_1 \vdash^k_{\mathbb{M}} s_1 \xrightarrow{\alpha} s$. That means if $\mathbb{M}$ can do a step in Γ_2 and s_2 then it can do the same step in Γ_1 and s_1.

Subsumption helps us to prune exploration since when we encounter an environment-state pair (Γ, s) which is subsumed by another pair which has been already explored, we do not need to continue exploring (Γ, s) because we have already captured its outgoing transitions.

```
let $\Gamma_0 = [\![\mathsf{true}]\!]$
var $\mathit{frontier} = \{(\Gamma_0, s_{\mathbb{M}})\}$
var $\mathit{explored} = \varnothing$
var $\mathit{transitions} = \varnothing$
while $\mathit{frontier} \neq \varnothing$
    let $(\Gamma, s) \in \mathit{frontier}$
    $\mathit{frontier} := \mathit{frontier} \setminus \{(\Gamma, s)\}$
    $\mathit{explored} := \mathit{explored} \cup \{(\Gamma, s)\}$
    foreach $\Gamma \wedge c \vdash^?_{\mathbb{M}} s \xrightarrow{\alpha} s'$
        $\mathit{transitions} := \mathit{transitions} \cup \{s, c, \alpha, s'\}$
        if $\neg\ \exists(\Gamma'', s'') \in \mathit{explored} \mid (\Gamma'', s'') \sqsubseteq ([\![\Gamma \wedge c]\!], s')$
            $\mathit{frontier} := \mathit{frontier} \cup \{([\![\Gamma \wedge c]\!], s')\}$
```

Figure 2: Subsumption Explorer for Action Machines

Figure 2 shows the algorithm for exploration in the presence of state subsumption. The algorithm maintains as its state a frontier of environments and states which still need to be explored. Initially, the frontier contains the environment representing the tautology constraint true and the initial state of the action machine. The algorithm furthermore has a state variable *explored* which captures those environments and states which have been encountered during exploration, and a variable *transitions* which holds the set of transitions it has found so far.

As long as there are elements in the frontier, the algorithm continues exploration, selecting one pair of environment and state from the frontier, and removing it. The choice of which pair is selected governs the search strategy (depth-first, breadth-first, or some priority search), which is kept open here. The algorithm then tries all steps which are possible from the given environment Γ and state s. If a step is possible, then it is added to the set of found transitions. We then check whether the resulting environment and state is subsumed by any of the environments and states which have been explored so far. Only if that is not the case, we add the resulting environment $\Gamma \wedge c$ and state s' to the frontier.

Theorem 1 *Let $\mathbb{M} = \mathbb{M}_1 \rightsquigarrow \mathbb{M}_2$ be a conformance machine. If the subsumption explorer terminates on $\mathbb{M}$, it discovered all conformance failures.*

Proof (sketch): We need to show that if we stop exploration at $p_2 = (\Gamma_2, s_2)$ since we have already explored $p_1 = (\Gamma_1, s_1)$ and $p_1 \sqsubseteq p_2$ holds, then p_2 does not reveal any new conformance failures which are not already discovered by exploring p_1. If the failure directly happens by stepping from p_2, this immediately follows from the construction of the subsumption relation, which ensures that any step possible in p_2 is also possible in p_1. Now suppose a failure happens after a number of steps starting from p_2, say in $p_2' = (\Gamma_2', s)$, with $\Gamma_2' \wedge c \vdash^{!}_{\mathbb{M}} s \xrightarrow{\alpha}$ **error**. Since the environment in p_2 is stronger than the environment in p_1, and all steps in p_2 are also possible in p_1, yielding the same target states, there must exist also a sequence of steps from p_1 which leads as to $p_1' = (\Gamma_1', s)$. The question now reduces to the problem whether, under the assumption that Γ_2' is stronger then Γ_1', $\Gamma_1' \wedge c \vdash^{!}_{\mathbb{M}} s \xrightarrow{\alpha}$ **error** holds. According to rule $C2$, this is the case if (a) $\Gamma_1' \wedge c \vdash^{k}_{\mathbb{M}_i} s \xrightarrow{\alpha} s_i$ (b) no constraint c_j exists such that $\Gamma_1' \wedge c_j \vdash^{!}_{\mathbb{M}_j} s \xrightarrow{\alpha} s_j$ and $\mathsf{SAT}[\![\Gamma_1' \wedge c \Rightarrow c_j]\!]$. Both follow obviously since Γ_2' is stronger then Γ_1': (a) $\mathbb{M}_i$ can do under Γ_1' at least the steps it can do under Γ_2', and (b) if there does not exist a step under the stronger environment which satisfies the constraint, then it also does not exist in the weaker environment.

4 Implementation

We have implemented action machines in the framework of the Exploring Runtime, XRT, realizing basic action machines for abstract state machines and scenario machines, the composition operators on action machines, and the subsumption explorer. In this section we give a glance of this work.

4.1 XRT

XRT is an exploration framework for CIL, the byte-code language of .NET. Processing arbitrary .NET managed assemblies, it provides functionality for introspection of the program, rewriting the program, and executing the rewritten program in an environment similar to an explicit state model-checker, where many states of the program can co-exist and are stored in an efficient way for exploration.

In addition to that, XRT provides an extension for symbolic execution, using terms to represent symbolic values, and connecting to an underlying constraint solver (currently Simplify [11], a theorem prover, but the architecture is generic such that other solvers can be attached as well). Symbolic execution roughly works as follows. The user – or a program rewriter – can create free logical variables for representing any value in the .NET type hierarchy. A standard rewriter on the program now substitutes certain primitive instructions – like addition on numbers – by a special handler that creates terms when the inputs are terms instead of actually performing the computation. For example, if the program contains the fragment `int x = Symbolic.Create<int>(); return x + 1`, then the return value will be the term $x + 1$ instead of a concrete value.

At certain points of execution with symbolic values, their interpretation is required for continuing execution, for example, if the program branches over a boolean value which is symbolic. In the case of the branch, the solver will be queried whether the boolean value is true in the current environment. If solver's verdict is true or false, the according branch will be taken; if it is unknown, a choice point for exploration is created, representing both branches as a possible program execution, and extending the environment by the assumption that the boolean value is true in the one branch and false in the other.

XRT provides symbolic representation of the full program state, including objects, arrays which can be symbolic themselves and accessed with symbolic indices, and type polymorphism, whose discussion goes behind the scope of this paper. For reasons of efficiency, a major design aspect of XRT is to represent state hybrid, with a concrete state part and a symbolic state part, where information is synchronized automatically between both representations. This is very close to the design of action machines, where we have an underlying abstract state (the concrete state part) and the environment (the symbolic state part).

4.2 Action Machines in XRT

The implementation of action machines in XRT uses an abstraction which closely resembles the foundations presented in the previous section. Each action machine implements an interface which provides a method to enumerate its possible steps. For efficiency reasons, the demanded step of a composition context is passed into that enumerator: this way, the action machine implementation can use the demand to prune the search for its step immediately. Figure 3 gives an excerpt of the interfaces and, for illustration of the principles, parts of the implementation of the product machine's state (note the use of C# 2.0 iterators with the **yield** keyword). The step enumerator of the product machine enumerates the steps of the first machine under the given step demand, and then enumerates the steps of the second machine using as the demanded the generated step of the first machine. The resulting step object combines the step both machines can do together under the constraint of the demanded step. The `Commit` method on a step produces the resulting state from that step, propagating the environment which resulted from the final composition of steps in a

```
interface IActionMachine {
  IMachineState InitialState { get; }
}
interface IMachineState {
  IEnumerable<IStep> GetSteps(IStep demand);
}
interface IStep {
  Action Action { get; }
  ISymbolicState Environment { get; }
  IMachineState Commit(ISymbolicState unifiedEnv);
}
class ProductMachineState : IMachineState {
  IMachineState state1,state2;
  public IEnumerable<IStep> GetSteps(IStep demand){
     foreach (IStep step1 in state1.GetSteps(demand)
       foreach (IStep step2 in state2.GetSteps(step1)
         yield return new ProductStep(step1,step2);
  }
  class Step : IStep {
     IStep step1, step2;
     public Action Action { get { return step2.Action; } }
     public ISymbolicState Environment { get { return step2.Environment; } }
     public IMachineState Commit(ISymbolicSTate unifiedEnv) {
        return new ProductMachineState(
              step1.Commit(unifiedEnv), step2.Commit(unifiedEnv));
     }
 }
```

Figure 3: Action Machine Implementation in XRT

larger context down to each individual step.

We have currently implemented abstract state machines and scenario machines as basic action machines. Abstract state machines are constructed from model programs in a very similar way as in Spec Explorer: in each state, all enabled actions are tried to produce successor states. However, the parameter generation mechanism is more flexible: parameters can be free symbolic variables, with and without domain restrictions, and the constraints of the step results from pre-conditions as well as branching on symbolic values in the actual method body.

Scenario machines are a new addition and follow the ideas developed in [4]. Basically, the scenario is given as a normal program which contains invocations of the actions being modeled. However, these methods are not actually executed, but substituted by an XRT code rewriter to suspend execution and yield a new step of the scenario machine. Thereby, for any output of the invocation, a new symbolic variable is created, representing the (for the scenario machine itself) unknown result. The modeler can add axioms over these results if desired by according assume and assert statements. A simple sample of a scenario machine for our publisher-subscriber model as given in Figure 1 looks as follows:

```
void PubSubScenario(){
  Publisher p = CreatePublisher();
  Subscriber c1 = CreadSubscriber();
  Subscriber c2 = CreadSubscriber();
  p.Register(c1); p.Register(c2);
  while (_){
    if (_) p.Publish(_); else _.Handle(_);
    }
  }
```

Here, the keyword "_" of Spec# denotes the creation of a free, anonymous symbolic variable. This scenario effectively constraints our model, if composed as a product machine, to creating one publisher, two subscribers, let them register, and then – for an arbitrary number of times – publish and handle data. Note that the independent exploration of this machine by the subsumption explorer yields a small state graph which expresses exactly the control flow of the scenario. The state graph is finite because of state subsumption, such that the while-loop will be actually represented as a cycle in the graph.

5 Conclusion

We presented a framework which supports model composition and exploration, addressing practical problems derived from the feedback of applying our model-based testing technology in the daily production process at Microsoft. Our approach is implemented on top of the exploring runtime XRT, and will become the core of the next generation of the Spec Explorer tool.

Related Work The approach of using model composition for scenario control is not new. The TGV tool [9], for example, uses finite automatons for describing so-called test purposes which are combined with models given in IOLTS. The TorX tool [12] has a customized description language which serves similar purposes in the ioco context. Our approach generalizes from those by allowing composition of arbitrary models in the product machine for the purpose of scenario control.

Our notion of conformance is closely related to alternating refinement in the context of interface automata, which was first introduced in [10], and studied in the context of game theory in [13]. To the best of our knowledge, we are the first which generalize alternating refinement to the symbolic case. A symbolic version of ioco theory has recently been proposed in [14]. This work does not deal with the level of detail we presented here, for example with the incompleteness of a constraint solver.

Using a hybrid explicit/symbolic state representation for exploration is not new. In [15] such a framework is introduced in the context of the Java Pathfinder model checker. XRT is similar to this extension of Java Pathfinder for symbolic execution, but goes beyond it by supporting symbolic execution for the full managed CIL, including symbolic structures, objects, and type polymorphism. More important for the context of this paper is the application of the symbolic state part as the glue for composition, which is new to the best of our knowledge.

Future Work This work is still at an early stage. One line of future work – which is rather implementation oriented – will be to make symbolic exploration accessible to our end customers. How do we represent the failing satisfiablity check of a constraint to the user such that she can understand it? Obviously, for reasons of accessibility, we want to reduce the amount of symbolic state to an essential minimum. One way to do so is to force enumeration of symbolic values earlier than actually required for the symbolic exploration, thereby still avoiding state explosion. We are looking at heuristics how to do that.

Another line of future work is extending our means of representing scenarios. Currently, scenario machines are given entirely programmatically, using symbolic values to create choice points, and procedural abstraction for scenario composition. We might want to adapt more declarative means of describing scenarios and their compositions.

In this paper, we only discussed some basic compositions of action machines. But there are more, which need further investigation. One essential among those are morphisms on the vocabulary of action machines. For example, one action machine might "implement" the actions of another machine using different types and methods. This is in fact a standard situation in conformance testing. We have an experimental implemetation for this composition, but it requires further analysis.

References

[1] 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.

[2] 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.

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

[4] Wolfgang Grieskamp, Nikolai Tillmann, and Margus Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 2004. In press, available online.

[5] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA'04*, volume 29 of *Software Engineering Notes*, pages 55–64. ACM, July 2004.

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

[7] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

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

[9] J.C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming - Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1-2):123–146, 1997.

[10] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Vardi. Alternating refinement relations. In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 163–178, 1998.

[11] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking, 2003.

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

[13] Luca de Alfaro. Game models for open systems. In Nachum 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.

[14] L. Frantzen, J. Tretmans, and T.A.C Willemse. Test generation based on symbolic specifications. In Jens Grabowski and Brian 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*.

[15] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, April 2003.