Action Machines: a Framework for Encoding and Composing Partial Behaviors

Wolfgang Grieskamp Nicolas Kicillof Nikolai Tillmann

Revision July 2006

Technical Report
MSR-TR-2006-11

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

Abstract

We describe *action machines*, a framework for encoding and composing partial behavioral descriptions. Action machines encode behavior as a variation of labeled transition systems where the labels are observable activities of the described artifact and the states capture full data models. Labels may also have structure, and both labels and states may be partial with a symbolic representation of the unknown parts. Action machines may stem from software *models* or *programs*, and can be composed in a variety of ways to synthesize new behaviors. The composition operators described here include synchronized and interleaving parallel composition, sequential composition, and alternating simulation. We use action machines in analysis processes such as model checking and model-based testing. The current main application is in the area of model-based conformance testing, where our approach addresses practical problems users at Microsoft have in applying model-based testing technology.

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, lacking engineering methodology and discipline, and adequate tool support. Model-based testing is one of the most promising approaches to addressing these problems. At Microsoft, model-based testing technology developed by our research group has been applied in the production cycle since 2003 [10, 3]. The second generation of our tool set, Spec Explorer [4, 14], which has been deployed in 2004, is now used by various product groups in the company for testing features of various product lines.

Although the basic concepts of our approach have proven to be valid and useful, user feedback indicates that improvements are needed, in particular in the area of *scenario-control* (selecting a representative set of scenarios from a potentially infinite model space) and of *model composition* (the combination of models of specific features into compound models). We view the first problem as a special case of the second one, where a scenario is described by a particular partial model, which is composed with the main model in order to obtain a sliced version of the latter.

This paper introduces *action machines*, a framework for encoding and composing partial behavioral descriptions. These descriptions can be given as state machines, scenario machines, model programs, or even regular programs, in diagrammatic or textual notation, and can be composed in a variety of ways yielding new behaviors exposed in the uniform encoding of action machines. We can use them, individually or in composition, in analysis processes such as model checking and model-based testing.

Action machines serve as a semantic foundation as well as the abstraction for the actual implementation. Semantically, action machines are a variation of labeled transition systems (LTS), where labels represent observable activities (henceforth *actions*) of the described artifact, and states capture full data models. Action labels may also have structure (e.g. represent method invocations with arguments). Labels and states may be partial, with a symbolic representation of the unknown parts. States may have attached constraints over symbolic parts. In contrast to LTS, action machines have a more general notion of initial states and provide a concept of *accepting states*, similar to that of finite state machines, which makes them suitable for sequential composition. The implementation of action machines is based on XRT [13] (Exploring Runtime), a software model-checker and virtual execution framework for CIL (Common Intermediate Language, the byte-code language of .NET). XRT provides symbolic state representation and exploration of .NET programs.

The main contributions of this paper are the following. We motivate the need for a composition framework of partial behaviors by showing that our approach addresses practical requirements users at Microsoft have when applying model-based testing technology. We present a *natural semantics* [17] of the abstraction of action machines and some of their main composition operators, namely *parallel* composition, *sequential* composition, and *alternating simulation*. Alternating simulation is the central notion of conformance used in model-based testing with action machines and earlier tools. We present a novel approach to the formulation of alternating simulation in a context where the underlying decision procedures for dealing with partial, symbolic

state are incomplete, using the notion of *may* and *must* steps of action machines. We also describe *subsumption exploration* of action machines and discuss and prove its properties, which imply that under certain conditions on action machines subsumption exploration does not fail to find error states. The paper concludes with a sketch of the implementation, which is close to the natural semantics, and a discussion of related and future work.

2 Review of Model-Based Testing with Spec Explorer

Spec Explorer is a tool for model-based testing of reactive, object-oriented software systems. These systems receive inputs and provide outputs, often as spontaneous reactions. Inputs and outputs are not just atomic data-type values, like integers, but 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 into *controllable* actions (the "inputs" we provide to the system under test by calling an action method) and *observable* actions (the "outputs" the system produces by spontaneously invoking action methods itself).

Model Programs Models are given in Spec Explorer by means of *model programs*. As the name suggests, model programs are very much like ordinary programs. This distinguishes them from e.g. axiomatic modeling approaches like those provided by the Z notation [23]. Model programs differ from ordinary programs in their intended use: they are not concerned with efficiency, and are formulated on a level of abstraction adequate for describing the particular problem at hand. Spec Explorer uses the Spec# language [1] for writing model programs. Spec# is a conservative extension of C#, adding to it high-level value data types like sets, maps, sequences and bags with comprehension notations; universal and existential boolean quantifiers; non-deterministic choice; contracts in the form of preconditions, postconditions, 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) [15], where states capture program state, and steps (transitions) between states are described by invocations of methods which have been marked as actions. Spec Explorer *explores* a model-program in order to generate a representative finite subset of the potentially infinite behavior of the ASM [10]. The exploration results in a state graph, where nodes represent model states and edges, action invocations. The exploration algorithm roughly works as follows: for a given model state, figure out those invocations (action-parameter combinations) which are *enabled* in that state. Enabledness is determined by the precondition of the method. Parameters used for invocations are provided by user-defined parameter generators, where some default generators are selected automatically (for example, the default parameter generator for objects delivers the set of all constructed instances 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, using filters, bounds, state grouping [10], and so on. The overall process is very similar to an explicit state model-checker. However, where a model-checker makes an existential search looking for some path which violates a condition, exploration for testing aims to produce a finite, representative subset of the model.

In general, for models with infinite state spaces (a frequent situation for faithful models of object-oriented programs), the extraction of a *complete* finite, representative scenario is indeed not decidable (see [10] for a formal analysis), where completeness here means complete coverage of the model. But even finite models can produce a huge amount of states and transitions. For that reason, Spec Explorer provides various means for the test engineer to manually slice the model and analyze the result by exploration and visualization. In general, Spec Explorer's methodology is based on a feedback loop between user-configuration, automatic exploration, and user-analysis of the exploration result; we call this process *scenario control*.

The actual conformance testing of an implementation 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. For a comprehensive discussion of Spec Explorer's concepts, the reader is referred to [4].

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

Excerpts of the model are given in Fig. 1 (top), omitting methods for construction of publisher and subscriber objects and for registration and unregistration. The state of the model consists of publisher and subscriber instances. A publisher has a field containing the set of registered subscribers, and a subscriber has a field representing the sequence of data it has received but not yet handled (its "mailbox"). The model simply describes how data is published by delivering it to the mailboxes of subscribers, and how it is consumed by a subscriber in the order it was published. The precondition of the publishing method states that only non-null data can be published, and the handling method of the subscriber has a precondition which enables it only if the mailbox is not empty, and if the data parameter equals to the first value in the mailbox. Note that the `Handle` method is an *observable* action, which comes out as spontaneous output from the system under test (SUT).

Fig 1 (bottom) 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 `S3`, a `Publish` invocation is fired, leading to state `S4`, which is an *observation* state where the outgoing transitions are observable actions. The meaning of an observation state is that the SUT has an internal choice to do *one* of the outgoing transitions, as opposed to a control state (`S3`)

```
class Publisher {
  Set<Subscriber> subscribers = Set{};
  [Action(ActionKind.Controllable)]
  void Publish(object data)
  requires data != null; {
    foreach (Subscriber sub
                    in subscribers)
      sub.mbox += Seq{data};
  }
}
```

```
class Subscriber {
  Seq<object> mbox = Seq{};
  [Action(ActionKind.Observable)]
  void Handle(object data)
  requires mbox.Count > 0;
  requires mbox.Head.Equals(data); {
    mbox = mbox.Tail;
  }
}
```

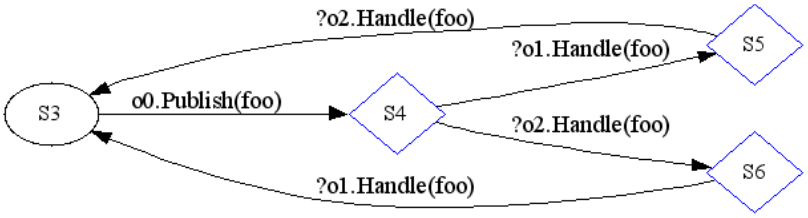

Figure 1: Publisher-Subscriber Model

where it must 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.

In order to generate the state graph, we had to annotate the model in Spec Explorer with scenario control information: we specified the parameter passed to the `Publish` method (here, `"foo"`), and restricted the number of publishers and subscribers created, and the order in which creation and registration happens.

In practice, models written with Spec Explorer are significantly larger than this simple example; yet they are rarely on the scale of real programs. In our applications at Microsoft, models typically introduce about 10 to 30 actions, with upto 2000 lines of model code. Yet, these models are able to test features with a code-base which is larger by an order of magnitude or more. This stems from the level of abstraction chosen in modeling.

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

Scenario control is a major issue. Currently, scenario control is realized by parameter generators, state filters, additional preconditions on actions to disable them in states where they should not be tested, and so on. Besides messing up the model with information belonging not in the actual model but rather in a separate test purpose description, complex scenario control problems could require non-trivial model extensions, maintaining additional state variables and so on. A further problem is the need to provide explicit values for parameters not discriminated by the model, which could stay unspecified, 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 what data is published, only that the published data equals the handled data. Finally, the way how scenario control is usually achieved in the current Spec Explorer tool spreads fragments of scenario control information all over 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.

For example, consider again the publish-subscribe example discussed above. Instead of slicing the model program itself, it would be adequate to define a test purpose in a scenario-oriented style independent of the model program, for example using a regular-expression notation as below:

```
new Publisher()/p; new Subscriber()/s1; new Subscriber/s2;
p.Register(s1); p.Register(s2);
p.Publish("foo"); p.Publish("bar"); _.Handle(_)*
```

This scenario describes traces where one publisher instance is created (named `p`), two subscriber instances are created (named `s1` and `s2`), the subscribers register, data is published and then a non-specified number of `Handle` actions on any subscribers are observed (where we do not care about the parameters). This is obviously a *partial* description of the publish-subscribe problem: we use placeholders ("_") for parameters we don't care about, and we do not express the logic which connects publishing with handling. However, this partial scenario model, when composed with a model program, effectively and easily slices a set of fully instantiated test cases.

Besides scenario control, another important issue identified by our users is composition of feature models. At Microsoft, as often in the industry, product groups are usually organized in small feature teams. 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 have a general way to compose larger models from existing ones.

Finally, users want to become independent of particular modeling notations, like Spec#. Instead, they want to model in programming languages like C#, which is justified by the fact that newer versions of C# provide high-level programming features which are close to those of Spec#. Moreover, different stakeholders in the modeling process want to use higher-level modeling paradigms than model programs, like UML diagrams. Yet, all these different modeling notations and styles should be compatible in composition and configuration for the testing problem.

We conclude from these observations that the ability to express partial behavioral models in a uniform encoding, and the composition of those behavioral models on various levels, is a most desirable feature for the model-based testing approach. This is adressed by our framework of *action machines*. The remainder of this paper is mainly dedicated to the formal definition of action machines. We provide a natural (operational) semantics in Sec. 3. This semantics closely resembles the actual implementation, which in turn is part of the forthcoming new tool generation of Spec Explorer, which is briefly discussed in Sec. 5.

3 Action Machines

In this section, we formalize the basic framework of action machines and assign operational meaning to the main operators of parallel composition, sequential composition, and alternating simulation, using the style of natural semantics [17].

3.1 Basic Definitions

Action machines are based on a notion of terms over a given signature, with actions being special terms. Action machine states are pairs of *environments* (representing the data state) and *control points* (representing the control state). We will describe these concepts, as well as the construction of machines and their transitions, in detail below.

Terms, Constraints, and Actions We assume an abstract universe of *terms* over a given signature, $t \in \mathbb{T}$. Terms capture values in the domain of our modeling and implementation languages, constraints, as well as action labels. Since we are in a symbolic computation world, terms also contain variables, $\mathbb{V} \subseteq \mathbb{T}$.

We assume that there is a certain class of terms which represent constraints, $\mathbb{C} \subseteq \mathbb{T}$. The actual structure of constraints does not matter: constraints can be just term equivalences or the full predicate calculus. However, we assume that $\mathbb{C}$ has the tautology true, contains equivalences between terms (written as $t_1 \equiv t_2$), and is closed under conjunction ($c_1 \wedge c_2$). To distinguish this operator of our constraint language from conjunction in the meta-logic of our formalization we use, where necessary, the notation $[\![c_1 \wedge c_2]\!]$.

Terms have an interpretation in a mathematical value domain, $\mathbb{D}$, which is described by a function $\xi_{\mathbb{T}} \in (\mathbb{V} \to \mathbb{D}) \times \mathbb{T} \to \mathbb{D}$. Given a value assignment to variable terms, the interpretation function delivers the value of a term in $\mathbb{D}$. For constraint terms, we denote the truth value in the interpretation image as $\mathsf{true}_{\mathbb{D}}$.

The further explicit structure of terms does not matter for the purposes of this formalization. To aid intuition, and for use in examples, let us nevertheless describe the structure of terms that represent *action labels* in an instance of the framework where actions stand for method invocations. We write $m(t_1, \ldots, t_n)/t$ for action terms, where m is a symbol identifying the invoked method, t_i are input arguments, and t is the return value. The symbol m behaves for the term language like a free constructor (self-interpreting function). Henceforth, two action labels are equivalent exactly when the action symbols are equal and the input and output terms are equivalent. In another instance of the framework we can, for example, distinguish action labels for *starting* the execution of a method and for *finishing* its execution and delivering a result. In yet another version, action labels may represent messages which are exchanged via communication channels.

Environments An *environment*, $e \in \mathbb{E}$, is a representation of a *(partial) global data state*. While action machines can work on different kinds of representations of environments, we use in this paper a particular instantiation which we refer to as the *environment reference model*.

Let $\mathbb{L}$ be a countable set of *locations* (global state variables). An environment is syntactically represented by a pair (α, c), where $\alpha \in \mathbb{L} \nrightarrow \mathbb{T}$ is a partial function from locations to terms, and $c \in \mathbb{C}$ is a constraint. A *model* of an environment is represented by a (total) function $\Gamma \in \mathbb{L} \to \mathbb{D}$; Γ is *valid* for e, written as $\Gamma \models e$, as follows:

$$\Gamma \models e \Leftrightarrow \exists\, v : \mathbb{V} \to \mathbb{D} \mid \xi_{\mathbb{T}}(v, c_e) = \mathsf{true}_{\mathbb{D}} \wedge \forall\, l \in \mathsf{dom}(\alpha_e) \cdot \Gamma(l) = \xi_{\mathbb{T}}(v, \alpha_e(l))$$

Thus a model is valid if there exists a variable assignment v such that the environment constraint evaluates to true and each location used by the environment is mapped by the model into the interpretation of its associated term under v. Note that locations not used by the environment can have arbitrary assignments in the model.

The interpretation of an environment is the set of its models, written as $\xi_{\mathbb{E}}(e) = \{\Gamma \mid \Gamma \models e\}$. Environments are partially ordered by *subsumption* which directly maps to set inclusion between interpretations:

$$e_1 \sqsupseteq e_2 \Leftrightarrow \xi_{\mathbb{E}}(e_1) \supseteq \xi_{\mathbb{E}}(e_2)$$

Subsumption $e_1 \sqsupseteq e_2$ indicates that e_1 is more general (contains less information) than e_2. This can be because e_1 fixes less locations than e_2, or because its constraint is weaker. Equivalence on environments, as derived from the subsumption ordering, is written as $e_1 \equiv e_2$, and coincides with model set equality.

With the ordering $\sqsubseteq = \sqsupseteq^{-1}$, environments build a *complete lattice* [5] with *meet* (least upper bound) $e_1 \sqcup e_2 = \xi_{\mathbb{E}}(e_1) \cup \xi_{\mathbb{E}}(e_2)$, *join* (greatest lower bound) $e_1 \sqcap e_2 = \xi_{\mathbb{E}}(e_1) \cap \xi_{\mathbb{E}}(e_2)$, and top and a bottom elements $\top_{\mathbb{E}}$ and $\bot_{\mathbb{E}}$, where $\top_{\mathbb{E}} = \mathbb{L} \to \mathbb{D}$ and $\bot_{\mathbb{E}} = \varnothing$. (Note that we have swapped symbols for meet and join compared to the mathematical literature, in order to match the intuition with set operations on environment interpretations.)

A few further syntactic operations are required on environments. Let $\nu \in \mathbb{L}$ denote a distinguished "scratch" location used for storing an action label. The introduction of ν greatly simplifies the formalization of action machine synchronization, since with help of it we can express the unification of transition labels and target states via the single concept of environment joining. We write $e[t]$ to denote the environment where the term t is assigned to the location ν, and all other locations are mapped to the assignment in e (henceforth, $\mathsf{dom}(\alpha_{e[t]}) = \mathsf{dom}(\alpha_e) \cup \{\nu\}$, $\alpha_{e[t]}(\nu) = t$ and $\forall\, l \in \mathsf{dom}(\alpha_{e[t]}) \cdot l \neq \nu \Rightarrow \alpha_{e[t]}(l) = \alpha_e(l)$). Note that since ν is a regular location apart of that we have a special syntax to denote its assignment in an environment, the meaning of all operations on environments carry over.

Finally, we define the reduction of an environment to the locations used in another environment: $(e_1 \downarrow e_2)$ denotes the environment $(\{l \mapsto t \in \alpha_{e_1} \mid l \in \mathsf{dom}(\alpha_{e_2})\}, c_{e_1})$.

Computable Operations on Environments Recall that we are aiming at an operational semantics for action machines. But environment operations like subsumption or joining are not computable in arbitrary term domains, and what range of the computable part can be actually decided in an implementation depends on the power of underlying decision procedures (that is, a constraint solver or theorem prover). Our goal is to give an operational semantics which is independent of these decision procedures – thereby reflecting the actual implementation architecture, in which we can

deploy different such procedures. To this end, we loosely axiomatize some operations on environments, reflecting the minimal requirements on the decision procedures.

Let *kind* $k \in \mathbb{K} = \{!, ?\}$ indicate the *verdict* of a decision procedure in the underlying computation domain regarding the feasibility of a decision: $k = ?$ indicates that a property could not be conclusively proven, and $k = !$ indicates that a property could be conclusively proven. On kinds, there is a join operator, written $k_1 \sqcap k_2$, where $k_1 \sqcap k_2 = !$ if both $k_i = !$, and $k_1 \sqcap k_2 = ?$ if one or both $k_i = ?$.

First, we assume a *computable approximation to subsumption*, written as $e_1 \sqsupseteq_! e_2$, where $e_1 \sqsupseteq_! e_2 \Rightarrow e_1 \sqsupseteq e_2$ for all environments e_1, e_2. Thus, the decision procedure for subsumption strictly respects the model semantics, but it might fail to detect some cases.

Second, we assume a *computable approximation to joining*; here we distinguish between a conclusive and an inconclusive join, as indicated by the kind k. We write $(e_1 \sqcap_k e_2) \mapsto e_3$ to indicate that a join *does* or *may* exist. This operator is a relation on *syntactic* environment representations, and is related to the model semantics as follows: $(e_1 \sqcap_k e_2) \mapsto e_3 \Rightarrow (e_1 \sqcap e_2 \equiv e_3) \wedge (k = ! \Rightarrow e_3 \not\equiv \bot_{\mathbb{E}})$, while $(\nexists e_3 \cdot (e_1 \sqcap_! e_2) \mapsto e_3) \Rightarrow e_1 \sqcap e_2 \equiv \bot_{\mathbb{E}}$. Thus, if the operational join is defined, then the resulting environment is equivalent to the model join (and non-trivial if $k = !$). Also, if the computable procedure conclusively determines that two environments cannot be joined, it means they have no models in common.

Third, we require a *computable approximation to extending an environment by a constraint*. Let c be a constraint. We write $(e_1 \wedge_k c) \mapsto e_2$ to denote that e_2 is the extension of e_1 by c, where k indicates the feasibility of this extension. The constraint c might share variables with e_1. Extension is explained as follows: let e_2' be constructed as $(\alpha_{e_1}, [\![c_{e_1} \wedge c]\!])$, then $(e_1 \wedge_k c) \mapsto e_2 \Rightarrow (e_2 \equiv e_2') \wedge (k = ! \Rightarrow e_2 \not\equiv \bot_{\mathbb{E}})$.

Machines Let $\mathbb{E}$ denote an environment domain as described above. An action machine is given as a tuple $\mathbb{M} = (C, A, I, T)$. C is a set of so-called *control points*, and $A \subseteq C$ is a set of *accepting control points*. $I \subseteq \mathbb{E} \times \mathbb{K} \times \mathbb{E} \times C$ is the *initialization transition relation*, and $T \subseteq \mathbb{E} \times C \times \mathbb{K} \times \mathbb{E} \times C$ is the *(regular) transition relation*.

We call a pair of an environment and a control point a *(machine) state* and write it as $e \cdot c \in \mathbb{E} \times C$. Initialization transitions from I relate an environment with an initial state and a kind, k, under which the state is obtained (k is explained below). We write $e_1 \xrightarrow{k}_{\mathbb{M}} e_2 \cdot c_2$ for initialization transitions. Regular transitions from T lead from states to states; the action label is contained in the special location ν of the target environment. For readibility, we write $e_1 \cdot c_1 \xrightarrow{k\ t}_{\mathbb{M}} e_2 \cdot c_2$ for regular transitions, which is syntactic sugar for $(e_1, c_1, k, e_2[t], c_2) \in T$.

Initialization transitions are allowed to refine the environment, but not to change it. This is imposed by the following property which holds for every action machine: $\forall (e_1 \xrightarrow{k} e_2 \cdot c_2) \in I \cdot e_1 \sqsupseteq e_2$. Such a refinement could be, for example, the allocation of a new location, or strengthening the environment constraint. Note that, in contrast, a regular transition can indeed change the environment by updating the contents of a location.

The kind of a transition, k, reflects the verdict of an underlying decision procedure regarding its feasibility: if $k = ?$, we call the transition a *may-transition*, meaning that

the feasibility proof has been inconclusive; if $k = !$, we call it a *must-transition*, whose feasibility has been successfully proven. The distinction between may-transitions and must-transitions is important for our notion of alternating simulation and conformance in the presence of incomplete decision procedures, as defined later in this paper. During the construction of composed machines, we need to propagate the transition kind adequately from the transitions of the constituting machines.

This definition of action machines is driven by the demand for a notion of composition which captures synchronization as well as concatenation. In synchronization, environments of the constituting machines (including the action label at location ν) are joined, and control states are combined using Cartesian product. Here, the environment plus action labels constitute the composition "glue". In concatenation, both accepting control points (identifying where to transition from one to the next machine) and initialization transitions provide the composition "glue". In particular, initialization transitions allow to instantiate a machine in the context of an environment obtained by running another machine.

3.2 Given Action Machines

A given action machine can result from a variety of sources. It may be constructed from an abstract state machine consisting of guarded update rules (as is the case in the Spec Explorer tool), from a state machine or statechart, from a sequence diagram or some other interaction-based notation, from a temporal logic formula, and even from an actual program. This paper is agnostic about given action machines. Whatever their source, by adhering to the encoding as an action machine, they will integrate into our framework. However, for illustration purposes, we define an abstract version of the *guarded-update* machine.

The transitions of a guarded-update machine are solely determined by the environment of the machine states. A transition from a source environment to a target environment exists iff an action's guard evaluates to true in the source environment, and the target environment results from applying the updates of the action to the source. This style of behavior description is very similar to the primary way how behaviors are described in the Spec Explorer tool (see the publish-subscribe example in Sec. 2).

A guarded-update machine, $\mathbb{M}_{I,R} = (C, A, I, T)$ is defined by a given initialization transition and a set of rules R, $(t, p, u) \in R$, where $t \in \mathbb{T}$ is an action label term, $p \in \mathbb{T}$ is a constraint, and $u \in \mathbb{E} \to \mathbb{E} \times C$ is an update function which maps a given environment to a new configuration. We have $C = \{\diamond, \circ\}$ and $A = \{\circ\}$, i.e. the machine has exactly two control states, one of which is accepting and the other is not.

The transition relation T is defined by the following rule, where $\mathsf{rename}(t_1, t_2)$ denotes the terms resulting from renaming all variables in t_1 and t_2 consistently to be distinct from other variables in use:

$$U1\ \frac{\begin{array}{c}(t_0, p_0, u) \in R \quad (t, p) = \mathsf{rename}(t_0, p_0) \\ (e[t] \wedge_k p) \mapsto e'[t'] \quad e''[t''] \cdot c' = u(e'[t'])\end{array}}{e \cdot c \xrightarrow{k\ t''}_{\mathbb{M}_{I,R}} e'' \cdot c'}$$

While this definition hides a great deal of detail in the update function u, it shows the approach by which we manage to embed given behavioral description techniques, including programs, into action machines.

3.3 Compositions

We now define core compositions of action machines, namely parallel composition, sequential composition, and alternating refinement.

Parallel Composition The parallel composition of two action machines results in a machine that transitions both machines simultaneously on a set of synchronized actions and interleaves transitions on other actions. For synchronized actions, a transition is only possible if the action labels and the target environments unify. A typical use of parallel composition is scenario control, where one machine represents the actual model and the other machine is a scenario to which the model is to be restricted.

Let $\mathbb{M}_1 \parallel_\rho \mathbb{M}_2 = (C, A, I, T)$ denote the parallel machine, where $\rho \subseteq \mathbb{T} \times \mathbb{T}$ is a relation characterizing action label pairs which are to be synchronized. By parameterizing via ρ, we capture the range from fully synchronized parallel composition (ρ relates all actions) to fully interleaved parallel composition (ρ is empty) in one formalization.

We have $C = C_{\mathbb{M}_1} \times C_{\mathbb{M}_2}$ and $A = A_{\mathbb{M}_1} \times A_{\mathbb{M}_2}$. The initialization transition relation I is defined by the following rule:

$$P1\frac{e \xrightarrow{k_1}_{\mathbb{M}_1} e_1 \cdot c_1 \quad e \xrightarrow{k_2}_{\mathbb{M}_2} e_2 \cdot c_2 \quad (e_1 \sqcap_{k_3} e_2) \mapsto e' \quad k = k_1 \sqcap k_2 \sqcap k_3}{e \xrightarrow{k}_{\mathbb{M}_1 \parallel_\rho \mathbb{M}_2} e' \cdot (c_1, c_2)}$$

Thus both machines initialize synchronously, and only those initial states are feasible for which the environments can be joined. The kind of initialization, k, is determined by joining the contributing individual kinds. Note that there might be no feasible initial state in the composed machine.

The regular transition relation, T, is determined by the following rules:

$$P2\frac{\begin{array}{c}\rho(t_1, t_2) \quad e \cdot c_1 \xrightarrow{k_1\ t_1}_{\mathbb{M}_1} e_1 \cdot c_1' \quad e \cdot c_2 \xrightarrow{k_2\ t_2}_{\mathbb{M}_2} e_2 \cdot c_2' \\ (e_1[t_1] \sqcap_{k_3} e_2[t_2]) \mapsto e'[t'] \quad k' = k_1 \sqcap k_2 \sqcap k_3\end{array}}{e \cdot (c_1, c_2) \xrightarrow{k'\ t'}_{\mathbb{M}_1 \parallel_\rho \mathbb{M}_2} e' \cdot (c_1', c_2')}$$

$$P3\frac{\neg\, \rho(t_1, t_2) \quad (i, j) \in \{(1,2), (2,1)\} \quad e \cdot c_i \xrightarrow{k\ t_i}_{\mathbb{M}_i} e' \cdot c_i' \quad c_j' = c_j}{e \cdot (c_1, c_2) \xrightarrow{k\ t_i}_{\mathbb{M}_1 \parallel_\rho \mathbb{M}_2} e' \cdot (c_1', c_2')}$$

Depending on whether $\rho(t_1, t_2)$ forces synchronization, rule $P2$ or $P3$ is chosen. In the synchronized case, the target environments of the transitions of both composed machines are attempted to join, and the equivalence of the action labels of both machines is attempted to establish. If either of those steps is not possible, no synchronized transition will be produced for the particular instantiation of the rule. In the interleaving case, one machine transitions, whereas the other stays at the same control point. Note that

because of potential sharing of parts of the environment, in the interleaving case one machine's transition can very well influence or even disable the transitions the other machine can make in subsequent states of the composed machine.

Sequential Composition Informally, sequential composition of two action machines amounts to executing one of them and then the other one. More technically, the resulting machine from sequentially composing $\mathbb{M}_1$ and $\mathbb{M}_2$ behaves since its initialization as $\mathbb{M}_1$; when it reaches an accepting state in $A_{\mathbb{M}_1}$, it also offers transitions from $\mathbb{M}_2$. One challenge in defining sequential composition is to avoid the introduction of label non-determinism when execution is continued with $\mathbb{M}_2$, as we will illustrate with an example after the definitions.

Let $\mathbb{M}_1;\ \mathbb{M}_2 = (C, A, I, T)$ be a sequential composition machine. We have $C = (C_{\mathbb{M}_1} \setminus A_{\mathbb{M}_1}) \uplus C_{\mathbb{M}_2} \uplus (A_{\mathbb{M}_1} \times \mathit{InitialControl}(\mathbb{M}_2))$, where $\mathit{InitialControl}(\mathbb{M}) = \{c : e \xrightarrow{k}_{\mathbb{M}} e' \cdot c\}$ denotes the set of initial control points of a machine. Thus the control points of the composed machines are the non-accepting control points of $\mathbb{M}_1$, plus the control points of $\mathbb{M}_2$, plus some new combined control points signaling that the sequence machine is in the process of turning into the second machine. Accepting control points in the resulting machine are accepting control points in the second machine plus combined control points made up from two accepting control points: $A = A_{\mathbb{M}_2} \uplus (A_{\mathbb{M}_1} \times (A_{\mathbb{M}_2} \cap \mathit{InitialControl}(\mathbb{M}_2)))$. The following two rules define the initialization relation:

$$S1\ \frac{e \xrightarrow{k}_{\mathbb{M}_1} e' \cdot c_1 \quad c_1 \notin A_{\mathbb{M}_1}}{e \xrightarrow{k}_{\mathbb{M}_1;\,\mathbb{M}_2} e' \cdot c_1}$$

$$S2\ \frac{e \xrightarrow{k_1}_{\mathbb{M}_1} e_1 \cdot c_1 \quad c_1 \in A_{\mathbb{M}_1} \quad e_1 \xrightarrow{k_2}_{\mathbb{M}_2} e_2 \cdot c_2 \quad k = k_1 \sqcap k_2}{e \xrightarrow{k}_{\mathbb{M}_1;\,\mathbb{M}_2} e_2 \cdot (c_1, c_2)}$$

Rule $S1$ just copies those initialization transitions from $\mathbb{M}_1$ that lead to a non-accepting control point. Rule $S2$ adds an initialization transition leading to a combined control point, for every accepting initial control point of $\mathbb{M}_1$. This intermediate control point forms a machine state with the resulting environment from the initialization transition. As we will show in the regular transition rules, this state exhibits the behaviors of both components.

We next define the relation T. When running $\mathbb{M}_1$, similar to initialization, we have rules for the case leading to an accepting ($S3$) and to a non-accepting ($S4$) control point in $\mathbb{M}_1$. These rules apply to the situations where we start from a proper $\mathbb{M}_1$ control point ($c = c_1$) and from a compound control point ($c = (c_1, \_)$). The remaining rule $S5$ defines the case where we run $\mathbb{M}_2$, again either from a proper control point in $\mathbb{M}_2$ or from a compound control point:

$$S3\ \frac{c \in \{c_1, (c_1, \_)\} \quad e \cdot c_1 \xrightarrow{k\ t_1}_{\mathbb{M}_1} e' \cdot c_1' \quad c_1' \notin A_{\mathbb{M}_1}}{e \cdot c \xrightarrow{k\ t_1}_{\mathbb{M}_1;\,\mathbb{M}_2} e' \cdot c_1'}$$

$$S4\ \frac{c \in \{c_1, (c_1, \_)\} \quad e \cdot c_1 \xrightarrow{k_1\ t_1}_{\mathbb{M}_1} e_1 \cdot c_1' \quad c_1' \in A_{\mathbb{M}_1} \quad e_1 \xrightarrow{k_2}_{\mathbb{M}_2} e_2 \cdot c_2 \quad k = k_1 \sqcap k_2}{e \cdot c \xrightarrow{k\ t_1}_{\mathbb{M}_1;\,\mathbb{M}_2} e_2 \cdot (c_1', c_2)}$$

$$S5\ \frac{c \in \{c_2, (\_, c_2)\} \quad e \cdot c_2 \xrightarrow{k\ t_2}_{\mathbb{M}_2} e' \cdot c_2'}{e \cdot c \xrightarrow{k\ t_2}_{\mathbb{M}_1;\,\mathbb{M}_2} e' \cdot c_2'}$$

We illustrate this construction using an example where we omit environments and kinds for simplicity. Consider $c_i \xrightarrow{t_i}_{\mathbb{M}_i} c_i$ as a single looping transition of two machines ($i \in 1\,..\,2$), where c_i is accepting. Sequential composition produces the following three transitions: $(c_1, c_2) \xrightarrow{t_1} (c_1, c_2)$ (this transition stays in the loop of the first machine), $(c_1, c_2) \xrightarrow{t_2} c_2$ (this transition leads into the second machine), and $c_2 \xrightarrow{t_2} c_2$ (this transition stays in the loop of the second machine).

The definition of sequential composition is more complicated than intuition might suggest since its goal is to avoid the introduction of label non-determinism. We could simplify the definition as follows: when a transition in $\mathbb{M}_1$ leads into an accepting control point, we duplicate the $\mathbb{M}_1$ transition to also lead into an initial state of $\mathbb{M}_2$. However, this would introduce label non-determinism of the produced traces. Consider again the looping example above. Using the alternate construction, we would get transitions $c_1 \xrightarrow{t_1} c_1$, $c_1 \xrightarrow{t_1} c_2$, $c_2 \xrightarrow{t_2} c_2$. Label non-determinism is not forbidden in our approach, but avoiding it results in behaviors that are easier to understand and analyze by tools and humans.

The way how we avoid label non-determinism in sequential composition has a consequence which should be noted: when the first machine transitions into an accepting control point, then in the composition the resulting environment will already contain the effect of initialization of the second machine (rules $S2$ and $S4$); otherwise we could not construct (c_1, c_2), which describes the behavior of both machines. This would be avoided in the alternative construction which introduces label non-determinism. However, since the effect of initialization is only strengthening of the environment, we prefer this drawback compared to the introduction of label non-determinism.

Alternating Simulation The alternating simulation machine represents the behavior of two action machines where the machines simulate each other regarding a partitioning of actions into *controlled* and *observed* ones. The second machine simulates all controllable actions of the first one, whereas the first machine simulates all observable actions of the second one. The notion of alternating simulation goes back to [2].

There are various applications of alternating simulation; the most common use in our framework is for conformance checking of reactive behavior, in which case the first machine is a model and the second an implementation (or a refined model). In this

application, a controllable action corresponds to an input provided by the first machine to the second one, which must be accepted by the latter; whereas an observable action corresponds to an output from the second machine, which must be acceptable for the first one. However, besides checking conformance, alternating simulation is also a composition operator, since it restricts the behavior of the two machines to one that is relevant to the interaction between both. This application has been explored in the context of *interface automata* [7].

We write $\mathbb{M}_1 \sim_\rho \mathbb{M}_2 = (C, A, I, T)$ for an alternating simulation machine, where $\rho \subseteq \mathbb{T} \times \mathbb{T}$ is a predicate characterizing controllable action-label pairs. When $\rho(t_1, t_2)$ is true, then actions t_1 (produced by $\mathbb{M}_1$) and t_2 (produced by $\mathbb{M}_2$) are considered to be controllable, otherwise they are observable. Let $\perp$ denote a value for representing a conformance error control point, then $C = (C_1 \times C_2) \uplus \{\perp\}$, and $A = A_1 \times A_2$. Initialization transitions I of alternating simulation are defined as follows:

$$A1\ \frac{e \xrightarrow{!}_{\mathbb{M}_1} e_1 \cdot c_1 \quad e \xrightarrow{!}_{\mathbb{M}_2} e_2 \cdot c_2 \quad (e_1 \sqcap_! e_2) \mapsto e' \quad (e' \downarrow e_1) \sqsupseteq_! e_1 \quad (e' \downarrow e_2) \sqsupseteq_! e_2}{e \xrightarrow{!}_{\mathbb{M}_1 \sim_\rho \mathbb{M}_2} e' \cdot (c_1, c_2)}$$

$$A2\ \frac{(i,j) \in \{(1,2),(2,1)\} \quad e \xrightarrow{k_i}_{\mathbb{M}_i} e_i \cdot c_i \quad \neg\,(e \xrightarrow{!}_{\mathbb{M}_j} e_j \cdot c_j \quad (e_1 \sqcap_! e_2) \mapsto e' \quad (e' \downarrow e_1) \sqsupseteq_! e_1 \quad (e' \downarrow e_2) \sqsupseteq_! e_2)}{e \xrightarrow{!}_{\mathbb{M}_1 \sim_\rho \mathbb{M}_2} e_i \cdot \perp}$$

Rule $A1$ represents a successful initialization transition, rule $A2$ a transition which leads into a simulation error. The rules state that, for any pair of initialization transitions, the composed initialization is only successful if both sub-transitions are *must*, i.e. proven to be feasible. Furthermore, the join of the resulting environments must exist, and must not specialize any parts of the individual initialization environments. This is expressed by using the reduction of the joined environment w.r.t the individual environments ($(e' \downarrow e_i)$). If any of these conditions does not hold – for example, if one machine can make an initialization transition from a starting environment e but the other one cannot, or if some transitions are *may*-transitions – an error is produced.

The motivation behind this definition is that we want alternating simulation to constitute a conservative over-approximation in the presence of partial environments and incomplete decision procedures. We thus admit false error reports, but we do not tolerate false success reports. Consider first the case where one of the sub-transitions is only a *may*-transition. This means we cannot guarantee that the machines can successfully initialize simultaneously. Consider now the case where the joined target environment does not subsume one of the individual target environments. This means that one machine would have specialized the environment in parts on which the other machine depends, which could potentially influence the other machine's future behavior. Note that the subsumption condition nevertheless allows each machine to add new private locations during initialization, which is captured by the reduct on the joined environment. The machines can also initialize shared locations, as long as the contents of those locations are specialized to the same degree.

The regular transitions T are defined by the following rules:

$$A3\ \frac{\begin{array}{c}(i,j) \in \{(1,2) : \rho(t_1,t_2)\} \cup \{(2,1) : \neg\, \rho(t_1,t_2)\} \\ e \cdot c_i \xrightarrow{k\ t_i}_{\mathbb{M}_i} e_i \cdot c_i' \quad e \cdot c_j \xrightarrow{!\ t_j}_{\mathbb{M}_j} e_j \cdot c_j' \\ (e_i[t_i] \sqcap_! e_j[t_j]) \mapsto e'[t'] \quad (e'[t'] \downarrow e_i[t_i]) \sqsupseteq_! e_i[t_i]\end{array}}{e \cdot (c_1,c_2) \xrightarrow{k\ t'}_{\mathbb{M}_1 \sim_\rho \mathbb{M}_2} e' \cdot (c_1',c_2')}$$

$$A4\ \frac{\begin{array}{c}(i,j) \in \{(1,2) : \rho(t_1,t_2)\} \cup \{(2,1) : \neg\, \rho(t_1,t_2)\} \\ e \cdot c_i \xrightarrow{k\ t_i}_{\mathbb{M}_i} e_i \cdot c_i' \\ \neg\,(e \cdot c_j \xrightarrow{!\ t_j}_{\mathbb{M}_j} e_j \cdot c_j' \quad (e_i[t_i] \sqcap_! e_j[t_j]) \mapsto e'[t'] \quad (e'[t'] \downarrow e_i[t_i]) \sqsupseteq_! e_i[t_i])\end{array}}{e \cdot (c_1,c_2) \xrightarrow{k\ t_i}_{\mathbb{M}_1 \sim_\rho \mathbb{M}_2} e_i \cdot \bot}$$

Both rules define i and j such that $\mathbb{M}_i$ is the "master" machine to be simulated in this transition by the "slave" machine $\mathbb{M}_j$ (the roles of master and slave machines for a particular transition depend on whether the transition is controllable or observable). In the success case, rule $A3$, for any transition the master can make, the slave must be able to follow with a *must*-transition which has the following property: the environment resulting from joining master's and slave's target environments, if reduced to the locations of the master, must subsume the environment from the master's transition. In other words, executing the slave is not allowed to influence directly or indirectly the behavior of the composed machine.

Let us consider an example to illustrate this construction. Let $e \cdot c_i \xrightarrow{m(t_i)}_{\mathbb{M}_i} e_i \cdot c_i'$ be transitions of a master and a slave, where $\mathbb{M}_1$ is the master and $\mathbb{M}_2$ is the slave. Let $e_1 = [l_1 \mapsto t_1 \mid t_1 > 0]$ and $e_2 = [l_2 \mapsto t_2 \mid t_2 > 1]$ be the target environments of transitions of master and slave. Let us expand the symbolic parts of the environments using the constraints. Then, $\mathbb{M}_1$ can transition with $m(0), m(1), m(2), \ldots$, whereas $\mathbb{M}_2$ can only transition with $m(1), m(2), m(3), \ldots$. This conformance failure is detected, since the join and subsequent enrichment of the environments yields $[l_1 \mapsto t_1, l_2 \mapsto t_2 \mid t_1 > 0 \wedge t_2 > 1 \wedge t_1 \equiv t_2]$, which does not subsume e_1. On the other hand, consider the roles of $\mathbb{M}_1$ and $\mathbb{M}_2$ swapped, such that $\mathbb{M}_2$ is the master. This case succeeds. In the composed result, the additional transition labeled with $m(0)$ of the slave $\mathbb{M}_1$ will be suppressed, since it is not demanded by the master $\mathbb{M}_2$.

4 Subsumption Exploration

Action machines can be explored by various means for different purposes. Here we present a class of explorers that do exhaustive exploration using environment subsumption to prune the search. Subsumption can help us prune exploration since when we encounter a machine state $e \cdot c$, and we have already explored a machine state $e' \cdot c$ such that $e' \sqsupseteq e$, then the transitions outgoing from $e' \cdot c$ are "captured" under certain conditions by those outgoing from $e \cdot c$.

Figure 2 shows the basic algorithm for state exploration in the presence of state subsumption. The algorithm maintains a set of *open* and of *closed* machine states. It

repeatedly picks an open state, closes it, computes its transitions, and adds to the set of open states those target machine states for which it does not find an already closed subsuming state.

The pruning technique used in subsumption exploration is justified by a property of action machines called *subsumption consistency*. Let $\mathbb{M}$ be an action machine. $\mathbb{M}$ is called *subsumption-consistent*, if for all e_1, e_2 reachable by initialization from $\top_{\mathbb{E}}$ (and subsequent regular must or may-transitions) such that $e_1 \sqsupseteq e_2$, and for every control point $c \in C_{\mathbb{M}}$, the following holds: $e_2 \cdot c \xrightarrow{k\ t}_{\mathbb{M}} e_2' \cdot c' \in T_{\mathbb{M}}$ implies $e_1 \cdot c \xrightarrow{k\ t}_{\mathbb{M}} e_1' \cdot c' \in T_{\mathbb{M}}$ with $e_1' \sqsupseteq e_2'$. This definition could be relaxed by requiring only the existence of a transition with a subsuming instead of an identical label. However, to simplify the presentation of the following argument, we use identical labels.

Theorem 1 (Subsumption consistency extends to exploration paths) *Let $\mathbb{M}$ be a subsumption-consistent action machine. For all environments e_1 and e_2, reachable from $\top_{\mathbb{E}}$ by initialization and subsequent transitions, such that $e_1 \sqsupseteq e_2$, if there exists a path $e_2 \cdot c \xrightarrow{t_1;\ldots;\ t_n}_* e_2' \cdot c'$, then there exist also e_1', with $e_1' \sqsupseteq e_2'$, and a path $e_1 \cdot c \xrightarrow{t_1;\ldots;\ t_n}_* e_1' \cdot c'$.*

Proof We need to show that the path in the more specialized environment e_2 can be simulated in the environment e_1. This is proved by induction on path length. For paths of length $n = 0$, the property is trivial. For paths with length $n > 1$, we have $e_2 \cdot c \xrightarrow{t_1} e_2'' \cdot c''$ as the first transition in the path, and by subsumption consistency then also $e_1 \cdot c \xrightarrow{t_1} e_1'' \cdot c''$, such that $e_1'' \sqsupseteq e_2''$. From here, the hypothesis can be applied inductively. □

Thus if we are interested in reaching certain control states (like the $\bot$ state in alternating simulation), subsumption exploration is a valid approach *provided* the explored action machine is subsumption consistent. This argument is independent of may or must-transitions, since subsumption exploration does not distinguish between those.

Yet, which action machines are subsumption consistent? For given action machines in the general case, we need to assume it. The given guarded-update machine as mentioned in a previous section is a likely candidate to satisfy subsumption consistency by construction (depending on its actual embodiment). If a guard holds in a stronger environment, it also holds in a weaker environment. And updates performed on a stronger environment naturally produce a new environment stronger than the one resulting from the same updates on a weaker environment.

var $open = \{e \cdot c \mid \top_{\mathbb{E}} \xrightarrow{k}_{\mathbb{M}} e \cdot c\}$
var $closed = \varnothing$
while $open \neq \varnothing$
  choose $e \cdot c \in open$; $open := open \setminus \{e \cdot c\}$; $closed := closed \cup \{e \cdot c\}$
  foreach $e \cdot c \xrightarrow{k\ t}_{\mathbb{M}} e' \cdot c' \in T_{\mathbb{M}} \mid \neg\, (\exists (e'' \cdot c') \in closed \mid e'' \sqsupseteq_! e')$
    $open := open \cup \{e' \cdot c'\}$

Figure 2: Subsumption Explorer for Action Machines

For compositions, we conjecture that all operators we have introduced in this paper preserve subsumption consistency *provided* they are applied to subsumption consistent sub-machines. We do not have a formal proof for this (which is future work) but some intuitive arguments. For the *parallel machine*, if a synchronized transition is possible in a stronger environment, it will also be possible in a weaker one. For the *sequence machine*, if we can glue the environments at a transition from one machine to the next in a stronger environment, then this should also be possible in a weaker one. Finally, for *alternating simulation*, if a master can do a transition in a stronger environment and the slave can simulate this transition, then the same should also be possible in a weaker environment.

5 Implementation and Application

The implementation of action machines is based on XRT [13] (Exploring Runtime), a software model-checker and virtual execution framework for .NET providing symbolic state representation and exploration of .NET programs. On top of XRT, we have defined an abstraction for action machines by a small set of interfaces, and provided implementations of these interfaces for action machine instances presented in this paper and several more.

We have devised and implemented a frontend notation for accessing action machine functionality, called CORD [11]. The CORD language is intended as an intermediate language resulting from compilation processes: typically, when mapping a modeling problem into the action machine framework, a compilation produces some .NET code (e.g. via C#) plus an augmenting CORD script that coordinates and configures the code for the modeling, checking and/or testing problem. CORD defines a discipline for describing action machines by behavior expressions. Behavior expressions include the constructions we have formalized in this paper, plus many more. Among those are the following: *Given machines* which allow to describe a single step machine for a given action with specified or symbolic parameters, all the operators of *regular expressions*, operators from *process algebra* (in addition to parallel synchronized/interleaved composition, hiding and translation), and operators specific for model-based testing, like an operator which takes a behavior, finitizes it, and unfolds it into a set of acyclic traversals realizing transition coverage.

The next version of the Spec Explorer tool has been developed based on action machines and CORD. This tool allows to model behaviors in a variety of notations, including model programs, textual scenarios, and UML diagrams. All these notations are mapped via CORD to action machines. For example, UML activity charts are translated into a compound machine where the atomic blocks are the single-step machine, and sequential composition, parallel interleaved composition, and repetition are applied.

The new Spec Explorer tool and its features have been presented to an internal Microsoft audience, and its design evolution has been monitored by selected users of the old Spec Explorer tool. It is currently in the early adaption phase, and preliminary results of its application to product development look promising.

6 Related Work

The construction of action machines, which combines concepts of process algebra (labeled transition systems), abstract state machines (rich shared, global data states), and finite automata (accepting states) is novel. Though process-algebraic and regular-expression-based mechanisms for composing behavior are ubiquitous in the Computer Science literature (e.g., event correlation and monitoring [20, 21, 26]; interaction-based scenario languages [22, 16], etc.), the combination of features is not found in any single approach. For instance, languages with rich composition operators usually have poor action or state structures (e.g., [22, 21]). On the other hand, approaches with rich data structure, like ASM, B, or Z, have poor support for behavioral composition.

Some approaches [25, 8, 19] combine a formal specification language (e.g. Z, Object-Z) to realize rich data state, with a process algebraic notation (e.g. CSP, Timed CSP, timed automata) to describe behavioral and control aspects. Although there are some similarities with our work, there also fundamental differences. Process algebraic approaches usually do not support sequential composition of entire behaviors. Action machines extend labeled transition systems, one of the canonical operational models for process algebras, with the necessary notions to deal with sequential composition in the presence of rich data state: namely accepting states, which determine when to transition from one to the next machine; and initialization transitions, which allow propagating the environment (data part) to the next machine. Another key difference to those approaches is the support of shared data state between machines, which comes apparent in the joining of environments in parallel synchronized transitions. While our approach can capture isolated state as a special case (which may be contained in "private" parts of the environment, in which case the joining problem reduces to the special location containing the action label), the potential for sharing is a necessary feature for our applications. One example of this is expressing a temporal property over the state of one machine by means of another machine. Another example is state-dependent parameter generation in model-based testing: here one machine should provide parameter generators for the other machine, where the parameter generators are defined in dependency of the data state of the other machine.

A further difference with the mentioned work is the particular attention we pay, in the context of a precise natural operational semantics, to the incompleteness of decision procedures on the symbolic term domain. We have found this to be essential when dealing with practical systems which involve symbolic computation, since in practice no symbolic decision procedure is complete. A novelty of our approach is the use of *may* and *must* transitions to formalize alternating simulation in the presence of such incompleteness. (The basic notion of alternating simulation has been introduced and analyzed in the context of interface automata [2, 7, 6].)

In the application area of model-based testing, the use of model composition for scenario control is not new. The TGV tool [9], for example, uses finite automata for describing test purposes to slice models given in IOLTS. The TorX tool [24] has a customized description language which serves similar purposes in the "ioco" context. Our approach is more general, by allowing composition of arbitrary models for the purpose of model slicing.

The use of a symbolic state representation for exploration has also been proposed

before. In [18] such a framework is introduced in the context of the Java Pathfinder model checker. Our implementation infrastructure based on 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. Most important for the context of this paper is the joining of symbolic states which provides the glue for synchronized transitions, which has not been attempted before as far as we know.

7 Conclusions

We have presented a framework to support model composition and exploration, addressing practical problems derived from the feedback of applying model-based testing technology in the daily production process at Microsoft. Our approach, implemented on top of the Exploring Runtime XRT, is the core of the next generation of the Spec Explorer tool.

While model-based testing is the primary motivation for this work, applications go further. For example, model-checking problems are naturally described with action machines using parallel composition of a machine with another machine that describes a negated scenario. The slice of this composition will produce the counter-example. The granularity of transitions and actions in this setting can be chosen as required. However, while we have indications that exploration of partial, symbolic models for MBT scales, mainly because transitions represent full method invocations, the model-checking case, where transitions might represent statements, imposes stronger challenges on symbolic exploration.

In this paper, we only discuss the semantics of the core compositions of action machines. A richer set of compositions is discussed informally, together with the action machine coordination language CORD, in [11]. Many constructs presented there require further mathematical analysis. We are also looking at a denotational foundation of action machines using trace semantics or similar as an augmentation of the natural semantics. One particular interesting question here are the algebraic properties of our composition operators.

Acknowledgment We would like to thank Victor Braberman for very useful comments on an early draft. Thanks go also to colleagues at Microsoft Research – Colin Campbell, Wolfram Schulte, and Margus Veanes – for useful discussions.

References

[1] Spec# tool, March 2005. http://research.microsoft.com/specsharp.

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

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

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

[5] B. Davey and H. Priestly, editors. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[6] 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.

[7] 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.

[8] J. S. Dong, R. Duke, and P. Hao. Integrating object-z with timed automata. In *ICECCS*, pages 488–497. IEEE Computer Society, 2005.

[9] J. 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] 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.

[11] W. Grieskamp and N. Kicillof. A schema language for coordinating construction and composition of partial behaviors. In *Proceedings of the 28th International Conference on Software Engineering & Co-Located Workshops – 5th International Workshop on Scenarios and State Machines*. ACM, May 2006.

[12] W. Grieskamp, N. Tillmann, C. Campbell, W. Schulte, and M. Veanes. Action machines – towards a framework for model composition, exploration and conformance testing based on symbolic computation. In *QSIC 2005: Quality Software International Conference*. IEEE, September 2005.

[13] W. Grieskamp, N. Tillmann, and W. Schulte. XRT - Exploring Runtime for .NET - Architecture and Applications. In *SoftMC 2005: Workshop on Software Model Checking*, Electronic Notes in Theoretical Computer Science, July 2005.

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

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

[16] ITU-T. Recommendation Z.120. Message Sequence Charts. Technical Report Z-120, International Telecommunication Union – Standardization Sector, Genève, 2000.

[17] G. Kahn. Natural semantics. In *Symposium on Theoretical Computer Science (STACS'97)*, volume 247 of *Lecture Notes in Computer Science*, 1987.

[18] S. Khurshid, C. S. Pasareanu, and W. 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.

[19] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, 1998. IEEE Computer Society Press.

[20] M. Mansouri-Samani and M. Sloman. Gem: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering Journal*, 4(2):96–108, 1997.

[21] C. Sánchez, H. B. Sipma, M. Slanina, and Z. Manna. Final semantics for event-pattern reactive programs. In J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. J. M. M. Rutten, editors, *CALCO*, volume 3629 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2005.

[22] B. Sengupta and R. Cleaveland. Triggered message sequence charts. *SIGSOFT Softw. Eng. Notes*, 27(6):167–176, 2002.

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

[24] 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.

[25] J. Woodcock and A. Cavalcanti. The semantics of circus. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.

[26] D. Zhu and A. S. Sethi. Sel, a new event pattern specification language for event correlation. In *Proc. of the IEEE Intl. Conf. ICCCN '01*, pages 586–589, 2001.