A Schema Language for Coordinating Construction and Composition of Partial Behavior Descriptions

Wolfgang Grieskamp Nicolas Kicillof

February 2006

Technical Report
MSR-TR-2006-13

We report on a schema language for coordinating the construction and composition of partial behavior descriptions. The language is a frontend to the semantical and implementation framework of *action machines*, which allows to encode behaviors of software artifacts in a language-agnostic manner, supporting both state-based and interaction-based description styles, as well as *partial* descriptions by means of symbolic representations. Our approach is currently being incorporated into an advanced model-based specification and testing environment at Microsoft Research.

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

1 Introduction

At Microsoft Research, we have in recent years developed a tool environment for model-based testing called Spec Explorer [3]. This environment allows users to model object-oriented, reactive software, analyze the specification with model-checking techniques and derive model-based tests. Feedback from Spec Explorer users at Microsoft shows that the tool's approach is feasible; however, enhancements are needed. Users have frequently requested *notational independence* and support for *model composition*.

While the Spec Explorer tool is based on a particular textual modeling notation called Spec# that describes behavior as rules of an abstract state machine, users want to be able to write models in a variety of notations and styles – they want to use textual notations as well as diagrams, and they want to use both state-based and interaction-based modeling. Moreover, users want to be able to combine models resulting from these different notations by model composition. For example, one common usage is the composition of a state machine or a model program with a scenario that describes a test purpose to be extracted from that model. Other applications are the combination of models of individual features, reflecting the different responsibilities in the process, and merging a primary model with others that represent aspects, reflecting crosscutting concerns. As an overall requirement, each individual model, as well as the composed model, should be amenable to analysis and testing.

Based on this feedback, we started the development of the next version of our modeling environment, which emphasizes *notation-agnostic model composition* as a central concept for organizing the modeling and model-analysis processes. In our new framework, models can arise from a variety of description techniques, and may describe full system behavior as well as partial aspects of it. Composition operators allow for the combination of models, and capture concepts like synchronized and interleaved parallel composition, substitution (similar to aspects in aspect-oriented programming), alternating refinement (a notion of conformance) and a set of regular-expression-like operators. Moreover, we have transformations on models for deriving test suites by unfolding their behavior. These transformations are suitable for both online (on-the-fly) and off-line testing.

The semantic foundation of this work is provided by *action machines* (AM) [6, 5], a uniform encoding of partial behaviors, which can capture a variety of description techniques, including state-machine and scenario-oriented ones. Action machines are essentially labeled transition systems (LTS) where labels represent actions in the behavior, and states are complex structures, i.e. full data states. The power of this approach comes from that both labels and states can be *partial*, i.e. contain symbolic parts. In compositions, the symbolic parts are unified and stepwise refined.

In this paper we introduce CORD, a language providing a frontend for action machines. CORD is intended as an intermediate language resulting from compilation processes: typically, when mapping a modeling problem into the action machine framework, compilation produces some .NET code plus an augmenting CORD script to coordinate and configure the code for modeling, checking and/or testing tasks. Though an intermediate language, CORD is declarative enough to illustrate action machine concepts, while its programming-language-like syntax is also human-writable.

This paper is organized as follows. We give a glance of the use of CORD for model-

ing and model-based testing. We then give an overview of the language core, behavior expressions, and a sketch of their semantics. We then discuss implementation aspects and end with the related work and conclusions.

2 Basics: A Sample

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

Below is a *model program* describing the behavior of the publish-subscribe pattern, we have omitted for simplicity the explicit registration of a subscriber with the publisher:

```
class Publisher {
  Set<Subscriber> subscribers;
  Publisher(){
    subscribers = new Set<Subscriber>();
  }
  void Publish(object data) {
    foreach (Subscriber sub in subscribers){
      sub.mbox.Add(data);
    }
  }
}
class Subscriber {
  Seq<object> mbox = new Seq<object>();
  Subscriber(Publisher pub){
    pub.subscribers.Add(this);
  }
  void Handle(object data){
    Assume.IsTrue(mbox.Count > 0);
    Assume.IsTrue(mbox.Head.Equals(data));
    mbox.RemoveAt(0);
  }
}
```

Model programs are the primary technology used in the Spec Explorer model-based testing tool and are related to abstract state machines (ASM) [8] and extended finite state machines (EFSM). ASMs are given by a set of *guarded-update rules*. Rules are fired in states in which the guard is true, producing new states via attached updates. In contrast to EFSMs, ASMs are not necessarily finite, since the data state they work on may be unbounded (though the core of the ASM theory maintains that each update step grows the data state only by a bounded amount).

Model programs represent guarded update rules by methods. We call each method representing a rule an *action*. This action is enabled (the guard is true) if all `Assume.IsTrue` statements in the method's body succeed when the method is executed in a given state with given parameters.

The C# code above by itself is just a normal program; in order to turn it into a model program and eventually an action machine, some configuration information needs to be supplied. This is one of the roles of CORD. A CORD script defined relative to the above program looks as follows:

```
config PubSub {
  action Publisher();
  action void Publisher.Publish(object data);
  action Subscriber(Publisher pub);
  action observable void Subscriber.Handle(object data);
  domain Publisher = instances(Publisher);
  domain Subscriber = instances(Subscriber);
  domain object = {"foo","bar"};
}
machine PubSubMachine() : PubSub {
  Program[PubSub]
}
```

This script does the following. First, a *configuration* picks those methods to be considered actions of the described behavior: the constructors for publishers and subscribers, plus all the methods in these classes. The configuration also specifies the *domains* to be used for action parameters. In this case, the domains are given on a per-type basis. The domain of a type provides defaults for every action parameter with that type. Here, for all occurrences of publisher and subscriber parameters, we intend to take as parameters *all instances* that have been created for that type in the state where the action is invoked (henceforth this parameter domain is a dynamically growing set), as expressed by the **instances** expression. For the object type used as published data, a fixed set of string values was chosen.

Then, the script declares an (action) *machine* named `PubSubMachine`. This machine is defined by a special operator, `Program[PubSub]`, which turns a (model) program into an AM via the configuration provided as a parameter. The program operator will take all the actions in the given configuration and interpret them as guarded-update rules, with parameter domains provided by the configuration.

With this script, the AM explorer in our tool can *explore* the machine `PubSubMachine`. A snippet of the exploration graph as displayed by the tool is shown in Fig. 1, depicting one cycle of publishing for one publisher and two subscribers (the exploration graph omits the creation phase). Circles represent so-called *control* states and diamonds are *observation* states. A control state is a state for which all outgoing transitions are labeled with controllable actions, whereas an observation state is one for which at least one outgoing transition is labeled with an observable action. The default for all actions is controllable; however, we have declared the `Handle` action as observable in the vocabulary `PubSub`.

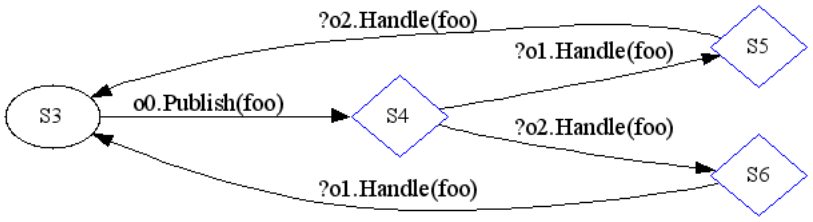

Figure 1: PubSub Snippet

Controllable actions represent inputs provided to a system, whereas observable actions are its outputs. The graph shows which inputs (method invocations from the outside) a confirming behavior needs to accept, and which outputs (observations of method invocations happening in the system) should be accepted by the model. Thus, a system conforms to this behavior snippet if it accepts the invocation of method `Publish` in state `S6`, and then (`S7`) one can observe the handling of the published data by one of the subscribers followed by the handling by the other one (`S8`, `S9`).

Obviously, the exploration graph is much bigger than the shown excerpt. In fact it is infinite, as one can create an arbitrary number of instances of publishers and subscribers. CORD supports various ways to prune the exploration. For example, one can declare *bounds* on the number of instances and *constraints* that prune exploration at certain states:

```
config BoundedPubSub : PubSub {
  bound Publisher = 1; bound Subscriber = 2;
  constraint {.
    foreach (Subscriber sub in
                State.Instances(typeof(Subscriber)))
      Assume.IsTrue(sub.mbox.Count < 3);
  .}
}
machine BoundedPubSubMachine() : BoundedPubSub {
  Program[BoundedPubSub]
}
```

This extends configuration `PubSub` by setting bounds on the number of publisher and subscriber instances, and a *global constraint* to filter out all states with more than two messages in a subscriber's mailbox. The constraint is given in embedded C# code; however, it makes use of a special library function `State.Instances` provided by our exploration runtime to obtain the set of living instances of a given type in the current state (like the **instances** construct in domain declarations). A constraint can call `Assume.IsTrue` to prune exploration, or `Assert.IsTrue` to flag assertion violations and therefore express state invariants.

While the machine `BoundedPubSubMachine` has now a finite behavior, it still contains redundancy, which might not be desired (e.g. in a model-based test setting). For example, this machine's behavior creates subscribers at arbitrary points, and explores all interleavings of those creations with publishing and handling. In order to limit this state explosion, CORD provides various means of *composing* machines and constructing them from scenario-style definitions:

```
machine TestPurpose() : BoundedPubSub {
  Publisher p; Subscriber s1,s2;
  new Publisher()/p;
  new Subscriber(p)/s1; new Subscriber(p)/s2;
  p.Publish(_);
  _.Handle(_)*
}
machine RestrictedPubSub() : BoundedPubSub {
  BoundedPubSubMachine() || TestPurpose()
}
```

The machine `TestPurpose` describes a partial behavior in a regular-expressin style. In general, the body of a machine definition is given as a *behavior expression*. Behavior expressions use operators known from regular expressions, others borrowed from Process Algebra, and some more. Their building blocks are references to other machines (like in the definition `RestrictedPubSub`), to *actions* (like in the definition of `TestPurpose`), and special operators (like `Program[BoundedPubSub]` used in the definition of machine `BoundedPubSubMachine`).

References to actions can have arguments (variables, wild cards or primitive values); and behaviors can declare local variables, like `p` above, or receive them as parameters. An action reference like **new** `Publisher()/p` in a behavior expression does not mean that the body of the method is executed, it only describes the "event" of invoking the method. Consequently, if the action has a return value, the actual value returned is unknown. Unknown values are represented symbolically. In the example, the symbolic return value is bounded to the variable `p`.

We can explore machine `TestPurpose` by itself, yielding the exploration graph in Fig. 2 (note the use of symbolic values like `v0`).

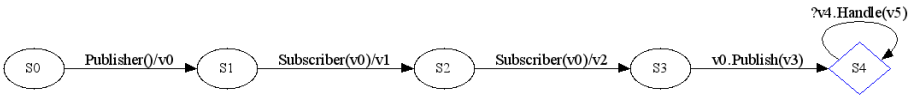

Figure 2: Exploring `TestPurpose`

The exploration of machine `RestrictedPubSub` yields the exploration graph in Fig. 3. Here, the partial behavior described by machine `TestPurpose` is blended through the parallel composition operator `||` with the behavior described by `BoundedPubSubMachine` (injecting parameter domains in the actual model semantics contained in the model program).

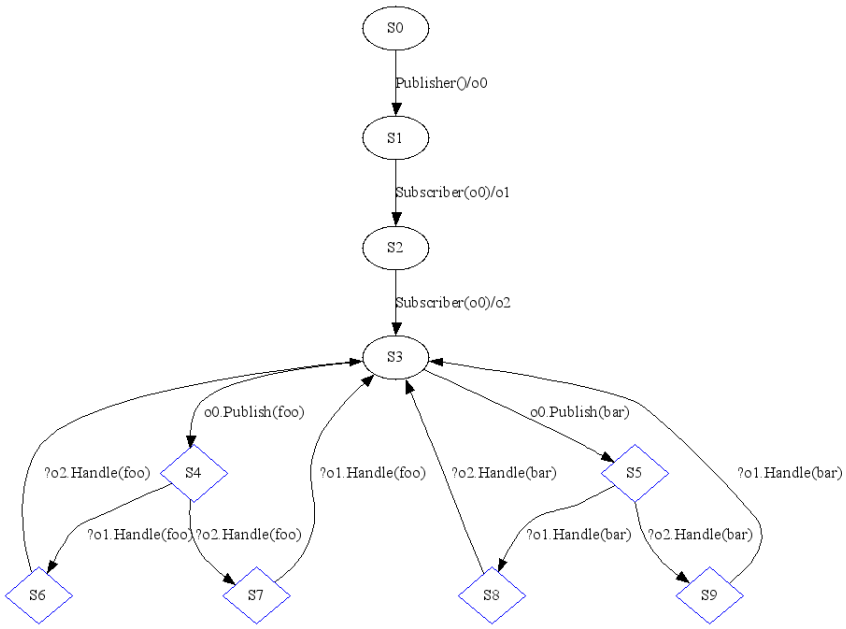

Figure 3: Exploring `BoundedPubSub || TestPurpose`

3 Going Deeper: Behaviors

Fig. 4 shows the syntax of behavior expressions in CORD. The intuition behind most constructs should be clear to the informed reader, though their semantical foundation in the presence of partial behavior raises questions (whose full treatment is beyond the scope of this paper):

In this section, we will discuss these constructs and sketch their meaning. We start with a synopsis of the underlying semantic framework: action machines.

3.1 Semantics of Action Machines

Each CORD behavior denotes an *action machine*. The theory of action machines together with its implementation is developing. The interested reader is referred to an early publication [6] and a forthcoming one [5].

Action machines are an instance of LTSs where labels represent some activity (*action*) of the described artifact and states are rich data models. Something that makes action machines unique is that both labels and states can be *partial*, i.e. contain symbolic parts.

Action machines are based on some representation of *terms* over a given signature and a set of *symbols* (logical variables). Over terms, a universe of *constraints* is available, which can be just conjuncts of equality constraints or full predicate calculus formulas, depending on the instantiation. Constraints come with (not necessarily com-

Let a be action names, m machines names, c configuration names, t translation names, o extension operator names, s substitution names, d variable declarations (in C# syntax), v variable names, l literals, and *stm* a statement in a .NET host language:

e	::=	$\_ \mid v \mid l$	*expressions*
b	::=	$[!][\mathbf{new} \mid e.]a(e_1,\ldots,e_n)[/e]$	*(negated) action reference*
	$\mid$	$m(e_1,\ldots,e_n)[/e]$	*machine reference*
	$\mid$	$\_$	*any action*
	$\mid$	$\ldots$	*any action repetition*
	$\mid$	$b_1;\ b_2$	*concatenation*
	$\mid$	$b_1?$	*option*
	$\mid$	b_1*	*zero-or-more repetition*
	$\mid$	b_1+	*one-or-more repetition*
	$\mid$	$b_1 \mid b_2$	*alternative*
	$\mid$	$b_1 \& b_2$	*permutation*
	$\mid$	$b_1 \mid\mid b_2$	*synchronized parallel*
	$\mid$	$b_1 \mid\mid\mid b_2$	*interleaved parallel*
	$\mid$	$b_1 \mid ? \mid b_2$	*sync/interleaved parallel*
	$\mid$	$b_1 \sim b_2$	*alternating simulation*
	$\mid$	$c_1[/c_2][b_1]$	*restriction*
	$\mid$	$[\sim]t[\,b\,]$	*translation*
	$\mid$	$[\#]s[\,b\,]$	*substitution*
	$\mid$	$\{\,d_1;\ \ldots;\ d_n;\ b\,\}$	*block*
	$\mid$	$\{.stm.\}{:}b_1$	*head constraint*
	$\mid$	$b_1{:}\{.stm.\}$	*tail constraint*
	$\mid$	$o[[c_1,\ldots,c_n]][(b_1,\ldots,b_n)]$	*extension operator*

Figure 4: Syntax of Behaviors

plete) decision procedure which allows to check for satisfiability w.r.t. an underlying model, and to check for subsumption (implication between constraints).

A *state* of an action machine (and its LTS) is a pair of an *environment* and a *control point*. An environment represents the global, potentially shared data state in which AMs operate, and can be represented as an assignment of locations (program variables) to terms and a constraint. Control points are specific to a particular kind of action machine, whereas environments are uniform.

The *labels* of the transitions of action machines are terms, which in general can represent arbitrary things, but in the instance used in this paper represent the *event* of a method being invoked, including the method name, its arguments, and its return value.

An action machine is given by a tuple (C,A,I,T), where C is a set of control points, $A \subseteq C$ is the subset of those control points which are accepting, I is an initialization relation, and T is a transition relation. The states of the machine are denoted as $\sigma \cdot c$, where σ is an environment and c is a control point. Initialization transitions $\sigma \longrightarrow \sigma' \cdot c \in I$ initialize an action machine in a given environment σ, leading into an initial machine state. Regular transitions $\sigma \cdot c \xrightarrow{t} \sigma' \cdot c'$ lead from a machine state to another,

and are labeled with an action term t. An accepting control point indicates a legal end state of a machine run.

A useful idiom for action machine compositions is transition synchronization. Here, we unify the environments and actions labels, and combine the control points in a way particular to the composition operator. As an example, consider transitions $\sigma \cdot c_1 \xrightarrow{t_1} \sigma_1' \cdot c_1'$ and $\sigma \cdot c_2 \xrightarrow{t_2} \sigma_2' \cdot c_2'$ of two action machines, starting from the same environment σ. In a parallel, synchronized composition, these machines transition together as $\sigma \cdot (c_1, c_2) \xrightarrow{t_1 \sqcap t_2} (\sigma_1 \sqcap \sigma_2) \cdot (c_1', c_2')$, provided that the labels and target environments do unify (informally indicated by the use of the $\sqcap$ operator). If labels or environments do not unify, the synchronized transition is not possible.

Another general idiom appears in sequential concatenation of behaviors. Here, we take the accepting states of the first machine (as determined by accepting control points) to identify when to transition into the second machine. The second machine is initialized in an environment resulting from the last transition of the first machine, henceforth the need for initialization transitions.

The transition relation of composed action machines can be mostly computed inductively based on the inductive definitions of the relations of the sub-machines – i.e. we do not need to compute the entire transition relation of a machine to compose it with another machine, which wouldn't be possible in the general case since behaviors might be infinite.

In the sequel, when we talk of the *meaning* of a CORD behavior, we implicitly mean the underlying action machine representing this behavior.

3.2 Signatures and Typing Conditions

CORD behaviors have a typing discipline which is based on the notion of *signatures*. A signature is a set of actions, and is declared by the configuration construct we have already introduced.

Every CORD behavior has an *offered signature*, and stands in the context of an *allowed signature*. The general typing rule is that the offered signature must be a subsignature of the allowed signature, where signature inclusion is based on action-set inclusion.

A machine declaration associates a machine with a signature. Let **machine** $m(v_1, \ldots, v_n)[/v] : c\ \{b\}$ be a machine declaration. The signature of m (and also the allowed signature for the behavior b) is the signature of the configuration c.

3.3 Primitive Behaviors

An action reference, $[!][\textbf{new} \mid e.]a(e_1, \ldots, e_n)[/e]$, offers the singleton signature containing just a if no negation ! is provided. When negated, it offers the whole allowed signature of the behavior context (including a, since in the presence of parameters to action invocations a negation might not necessarily exclude all instances of a). The meaning of the negated action reference is to deliver one transition per action in the allowed signature, with the transition for action a constrained not to match the given parameters, while parameters for all other actions are unbound.

A machine reference behavior, $m(e_1,\ldots,e_n)[/e]$, offers the signature of the referred machine. Semantically, this can be seen as unfolding. Note that we do not support recursive machine references.

The "any action" behavior, _, offers the whole allowed signature. Its meaning is a behavior which has a transition for each action in the demanded signature, with parameters unbound. The "any action repetition" behavior, ..., is just a shortcut for $\_*$.

3.4 Regular Expressions

CORD provides a set of behavior operators based on regular expressions: b_1; b_2 is the sequential concatenation of the given behaviors, b? is b where its initial states are made accepting, $b*$ means repeating b zero or more times, $b+$ means repeating b one or more times, and $b_1|\,b_2$ is alternative (either b_1 or b_2). The permutation operator, $b_1\&\,b_2$, is equivalent to $b_1; b_2|b_2; b_1$. The offered signature of all these behaviors is the union of the signatures of its sub-behaviors.

3.5 Parallel Compositions

CORD provides three operators for parallel composition. In the *synchronized parallel composition*, written as $b_1||b_2$, all transitions of the composed behaviors must be synchronized; transitions which can not synchronize are excluded. The offered signature of this behavior is the *intersection* of the signatures of the sub-behaviors.

In the *interleaved parallel composition*, written as $b_1|||b_2$, all interleavings of the transitions of the sub-behaviors are produced; the offered signature of this behavior is the *union* of the signatures of the sub-behaviors.

The *synchronized/interleaved parallel composition*, written as $b_1|?\,|b_2$, is a mixture of these both constructs. Actions in the intersection of the signatures of the sub-behaviors must synchronize, the remaining actions are interleaved. The offered signature is again the union of the signatures of the sub-behaviors.

3.6 Alternating Simulation

Alternating simulation [1] is a powerful operator for describing behavior refinement, and is related to the concept of *interface automata* [4]. We used the notion informally in Sec. 2 to describe conformance of behaviors in model-based testing.

Alternating simulation is written as $b_1 \sim b_2$ in CORD. The typing conditions are as follows. Recall that actions are either observable or controllable. The set of controllable actions in the offered signature of b_1 must be a subset of the controllable actions in the offered signature of b_2, and vice-versa: the set of observable actions in the offered signature of b_2 must be a subset of the observable actions in the offered signature b_1. The resulting signature of the composition is the union of the controllable actions of b_1 and the observable actions of b_2.

The meaning of alternating simulation as a composition operator is as follows. In each state, every transition of b_1 labeled with a controllable action must be simulated by a transition in b_2, and every transition in b_2 labeled with an observable action must be simulated by a transition in b_1. Simulation hereby means that the simulating machine

must make transition with a matching label, which is at least as partial as that of the other machine, and results in an environment which is at least as partial as that of the other machine (this condition is expressed by subsumption in our semantic framework).

If a simulation as described above is not possible, a distinct error state is produced in the composed behavior. If a simulation is not demanded – for example, b_2 has more controllable transitions than b_1 – these non-demanded transitions are dropped in the result.

Thus, besides providing a powerful notion of conformance, alternating simulation actually reduces the composed sub-behaviors. In the Interface Automata literature[4], this is motivated by the application where b_1 represents an "environment" in which the behavior b_2 (an "interface") is deployed. The composition cuts off everything from b_2 not used by the "environment", and in turn cuts off anything from b_1 not produced by the "interface".

3.7 Restriction

The behavior $c_1[/c_2][b]$ hides transitions of b not labeled with actions from the configuration c_1; it behaves like a hiding operator in process algebra. The offered signature of the composed behavior is the intersection of the offered signature of b and the signature of c_1. If c_2 is given, the allowed signature for the behavior b will be that of c_2, otherwise it will default to the allowed signature of this behavior's context.

The meaning is that all transitions with actions from b which are not in the signature c are contracted in the resulting behavior into a transition which is labeled with a visible action. Consider a trace $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots \xrightarrow{t_n} s_n$. Suppose t_i, $i < n$, is constructed from hidden actions, and t_n from a visible action. In the resulting behavior, this trace will be represented as $s_0 \xrightarrow{t_n} s_n$.

The construction of the restriction behavior is the only operator we present in this paper which can not be defined solely inductively. The reason is that, when producing the transitions of this behavior, we need to "look ahead" into the transitions of the underlying behavior b to find the next transition with a visible action. In fact, this lookahead might diverge, since there might be an infinite trace of transitions with hidden actions. The pragmatic solution our tool provides for this problem is to have a bound for this lookahead which can be globally configured; if the bound is exceeded, we produce an error state.

3.8 Translation

Translation allows to map a behavior into another one with a different signature, and is written as $[\sim]t[\,b\,]$, where t is the name of a *translation declaration*, a particular construct of CORD discussed below. The modifier $\sim$ indicates whether the translation t should be applied from left to right or from right to left. A translation declaration has a "left" and a "right" signature; if $\sim$ is not given, then the left signature of t becomes the allowed signature for b , while the offered signature of the overall behavior is given by the right signature of t; otherwise, it is the opposite way.

Below is a sample of a translation declaration:

```
// C#
class A { void foo(int x); }
class B { static void bar(B b, int x, int y); }
// CORD
config CA { action void A.foo(int x; }
config CB { action void B.bar(B b, int x, int y); }
translation Trans : CA <-> CB {
  domain A <-> B;
  action { A a; int x; int y;
         a.foo(x) <-> B.bar(a,x,y)
         {. Assume.IsTrue(y == x+1); .};
}
```

Translations are declared based on two configurations (here `CA` and `CB`) which determine the left and right signature of the translation. They are defined by a two sets of rewriting rules, one over type domains and one over actions.

A type domain rewriting rule translate values. This mapping can be calculated automatically (as in this sample) or provided by embedded code (not shown in the sample). The automatic translation of type domains roughly works as follows. Suppose a is a value of type A. The translation of a into the domain of B is represented by a term $trans_B(a)$, where we consider $trans_B$ as an uninterpreted function. Vice-versa, the translation of a b into A is represented by $trans_A(b)$. These translation functions are related by the axiom $trans_A(trans_B(x)) \equiv x$ (and vice-versa), for all x.

Action translations map action labels from one signature into the other. Each action rule is parameterized over a set of variables (in the sample `a`, `x`, `y`), which are typed in the left signature of the translation. When using these variables on the right side of the rule, these variables are mapped according to the type domain rules. Henceforth, `a` has type `A` on the left side of the rule, and type `B` on the right side.

The action rules are applied during translation of a behavior as follows. For each transition label, every matching action rule is instantiated to produce a new label and a new transition with that label, applying possible type domain translations as sketched above, and evaluating constraints as imposed by application conditions in embedded code. If for a given transition no rule matches, this transition will be dropped; therefore translations can also filter behavior. If for a given transition more than one rule may match, then one transition will be produced in the translation result for each matching rule.

For example, consider a transition $s \xrightarrow{t} s_1$, where $t = a.foo(x)$, and x is a symbolic integer value. Applying the above translation `Trans`, this transition is translated as $s \xrightarrow{t'} s_2$, where $t' = B.bar(trans_B(a),x,y)$, and the difference between s_1 and s_2 is the addition of the constraint $y \equiv x+1$ to the environment of s_2. Consider now this transition is synchronized with a transition of another machine with label $B.bar(b,2,3)$. The synchronization will produce an environment with equivalences $trans_A(a) \equiv b, x \equiv 2, y \equiv 3$. With the axiom $\forall x \bullet trans_B(trans_A(x)) = x$, we can infer the additional equivalence $trans_B(b) \equiv a$, allowing us to map back and forth between the translated domains.

3.9 Head/Tail Constraints

Head/tail constraints allow to inject assumptions and assertions into behaviors, written in embedded code (C# or other .NET languages).

A head constraint is given as {. *stm* .}:*b*. The meaning is that in every initial state of the behavior *b*, *stm* will be executed. A tail constraint is given as *b*:{. *stm* .}. Here the meaning is that *stm* will be executed in every accepting state of behavior *b*.

Embedded statements can access all local variables of the behavior, as well as global data which is available in the context of the .NET (model) program augmented by CORD. Typically, they use constructs like `Assume.IsTrue` to add assumptions, and `Assert.IsTrue` to add assertions, as we have already seen. In general, a path of transitions which leading to a state in which `Assume.IsTrue(false)` holds is pruned from exploration, whereas a path which leads into a state in which `Assert.IsTrue(false)` is marked as an error path and cuts exploration.

A typical application of tail constraints is to express negative scenarios. Suppose the following CORD machine (using the publish-subscribe example from Sec. 2, where the logic is assumed to be extended by explicit registration of subscribers):

```
machine NoHandleBeforeRegister() : PubSub {
  Subscriber s; !_.Register(s) *;
  s.Handle(_) : {. Assert.IsTrue(false) .}
}
```

This machine states that any path which leads into handling but does not contain a registration of a given subscriber is an error. Using such a machine in synchronized parallel composition with a model or program amounts precisely to the setting usually found in model-checking of temporal properties.

3.10 Substitution

The purpose of substitution is to replace every occurrence of certain steps in a given behavior with another whole behavior. The gluing is provided by a *pattern* that identifies those steps to be replaced. The application of a substitution to a behavior is written as [#]*s*[*b*], where *s* is the name of a *substitution declaration*, a construct of CORD discussed below. The presence of the # modifier indicates that the result of the substitution must behave according to a *co-routine semantics*, otherwise the *routine semantics* is applied. This difference is also explained below.

This is an example of a substitution declaration in CORD:

```
config Concrete {
  action FileOpenDialog();
  action void FileOpenDialog.Show();
  action string FileOpenDialog.GetSelectedPath();
  action static File FileSystem.Open(string path);
}
machine FileOpenMachine()/File f : Concrete {
  string p; FileOpenDialog o;
```

```
  new FileOpenDialog()/o; o.Show();
  o.GetSelectedPath()/p; FileSystem.Open(p)/f;
}
config Abstract {
  action File FileManager.OpenFile();
  action void FileManager.WriteFile(File f,
                                    string data);
  action void FileManager.CloseFile(File f);
}
substitution Subs : Abstract -> Concrete {
   File f; object d;
   _.OpenFile()/f -> { FileOpenMachine()/f }
   _.WriteFile(f, d) -> { f.Write(data) }
   _.CloseFile(f) -> { f.Flush(); file.Close() }
}
```

When this substitution is applied to a behavior (that must adhere to the allowed signature `Abstract`) the result will have as its allowed signature `Concrete` and every transition labeled with a term that matches the left-hand side of a rule (a line with a `->`) will have been replaced with the corresponding right-hand-side behavior. Necessary equality constraints are added to the environment in order to achieve the effect of parameter passing from the original machine to the inserted behavior and back.

In the co-routine semantics, the right-hand-side behavior is not initialized each time it is inserted, but execution continues from the (accepting) control point reached in the previous insertion, thus maintaining its own internal state. In the proposed example, co-routine semantics could be used to reflect the fact that the `FileOpenDialog` preserves the last opened locations between invocations.

Of the multiple applications of substitution we are particulary interested in two. The first one is *hierarchical model refinement*, where steps in a (more abstract) model are replaced with specific (more detailed) behaviors. For example, a model might contain transitions labeled Initialization and Termination, which are later instantiated with machines describing actual initialization and termination sequences. The second application is Aspect Oriented Modeling. In this setting, the substitution is used as the weaving mechanism, and the result is similar to that of *around advices* in Aspect Oriented Programming[10], where the original join point is replaced by a different behavior provided by the aspect writer. Quantification is rich, allowing for predicates over the environment and action label equivalences. We plan to further extend this operation (through the use of *triggered scenarios* [14, 15, 2]) in order to allow complete behavior specifications to identify sets of gluing points (*pointcuts*).

3.11 Extension Operators

CORD provides so-called *extension operators* which allow to plug-in new ways to construct machine from external and additional composition operators. The syntax for the application of an extension operator is $o[[c_1,\ldots,c_n]][(b_1,\ldots,b_n)]$. The c_i's and the b_i's are any number of configurations/behaviors which are passed on to the operator.

Extension operators are plugged into the system in a way which we leave open here. Essentially, one can describe the arity of the operator (i.e. which number of configurations and/or behaviors it takes), and attach code to this operator to compute the offered signature during type checking and to realize the intended semantics during runtime.

We have seen one extension operator already in use, namely `Program[PubSub]` in Sec. 2. This operator takes the configuration in `PubSub` and uses it to construct a guarded-update machine from the .NET program augmented by CORD.

There are various other extension operators which come with the current implementation of CORD, among them in particular operators which expand a behavior into a test-suite using various traversal techniques.

4 Implementation

The implementation of CORD and action machines is based on XRT [7] (Exploring Runtime), a software model-checker and virtual execution framework for .NET. XRT provides symbolic state representation and exploration of full .NET code. On top of XRT, we have defined an abstraction for action machines via a small set of interfaces. This abstraction takes environments as provided by XRT's data state model, and adds the concept of action machines and action machine states as interfaces. Relations for transitions are described by enumerations delivered by the machine and state interfaces. Each of the CORD behaviors we have discussed is represented by an according implementation of these interfaces. The compilation of behavior expressions is therefore just a straightforward denotational mapping from the abstract syntax tree into an action machine. As is common for compiler generators, embedded C# code is dealt with by extraction into a separate C# document without deeper analysis; this document is then compiled using the standard C# compiler and loaded into the running program using reflection.

Technically, our current implementation makes one significant simplification regarding realization of synchronization. Recall, in the sketch of the semantics above, the synchronization of transitions $\sigma \cdot c_1 \xrightarrow{t_1} \sigma_1' \cdot c_1'$ and $\sigma \cdot c_2 \xrightarrow{t_2} \sigma_2' \cdot c_2'$ as $\sigma \cdot (c_1, c_2) \xrightarrow{t_1 \sqcap t_2} \sigma_1 \sqcap \sigma_2 \cdot (c_1', c_2')$. Actually, instead of computing the join of two environments, we use the environment resulting from the first transition σ_1' as the starting environment for the second transition. This is safe if the transitions do commute (in most of our applications they do, even in the presence of state sharing via the environment). While this treatment is not by design, but a consequence of XRT limitations, which we intend to overcome soon; it might turn out to be a useful optimization switch in future versions of the implementation.

5 Related Work

In order to better understand the contributions of CORD, compared to existing research, it is worth recalling some of its features that make it specially suitable for model-based

specification and testing of running software artifacts. Firstly, it features a rich notion of state and actions (with symbolic parts). Secondly, it serves as a coordination language for both state-based and interaction-based description styles. Moreover primitive behavior can be described from scratch in CORD but it can also be given in the form of any running component, provided its behavior can be exposed as an action machine. Thirdly, it features a neat distinction between controllable and observable actions. Finally, it features a carefully crafted alternating simulation operator as a first class citizen thus simplifying conformance testing tasks.

Thus, though Process-Algebraic and regular-expression-based mechanisms for composing behavior are ubiquitous in the Computer Science literature (e.g., event correlation and monitoring [12, 13, 16]; interaction-based scenario languages [14, 9], etc.), features mentioned are not found in any single approach. For instance, languages with the richest composition operators usually have poor action or state structures (e.g., [14, 13]) to actually predicate on. Event correlation approaches like GEM use a regular-expression-like syntax to denote event patterns. Unlike CORD, these languages are either focused on monitoring distributed system or, as in the case of PAR, aimed at writing event-pattern reactive programs that recognize temporal patterns of events and respond by generating output notifications in publish-subscribe architectures.

High level Message Sequence Charts [9] structure pieces of behavior by means of graphs. TMCS ([14]) extend MSCs with partial and conditional scenarios, together with process-algebra-like operators to compose these building blocks. Conditional scenarios in this setting are primarily meant to refine non-deterministic based specifications by constraining and retaining desirable sequences. Several other extensions have been proposed to MSCs in order to overcome specific deficits in the notation, sometimes adding operators to combine diagrams in different ways. As an additional example, [11] introduces a join operator to combine overlapping scenarios, which addresses one of the many forms of coordination that we intend to cover.

6 Conclusions

We have presented CORD, a novel language for coordination of construction and composition of partial behaviors. CORD is based upon and reflects the semantical and implementation framework of action machines, and provides a powerful, language-agnostic approach to model-checking, model-based testing, and other analysis activities based on models.

The combination of state-based and interaction-based description on a *technical* level is naturally achieved in our framework. A key feature which makes CORD and action machines unique and flexible is the systematic support for partial behavior descriptions, encoded by symbolic states and transition labels, which captures many of the usual technical concerns. Indeed, we *do not* see our approach as a methodological answer to the problem of combining state-based and interaction-based formalisms, which is indicated by the fact that we consider CORD as an intermediate language, not a language for end-users. Rather, our approach provides a technical platform which allows to conduct further experiments in this direction.

Immediate practical application of the work presented in this paper is apparent in

its use for model-based testing. As explained in the introduction and in Sec. 2, MBT users would like to combine state-based models, like model programs, with scenarios in order to extract test purposes; they would like to compose models of individual features; and they would like to relate various different notational styles. Our approach offers direct technical answers to these demands. Our experiments indicate that, in these application areas, the presence of partial states and labels does not introduce a scalability problem.

6.1 Acknowledgement

We would like to thank Victor Braberman, UBA, for very useful comments on an early draft of this paper and for contributing an analysis of related work. Thanks go also to colleagues at Microsoft Research – Colin Campbell, Nikolai Tillmann, Wolfram Schulte, and Margus Veanes – for many inspirations and fruitful discussions.

References

[1] 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.

[2] V. Braberman, N. Kicillof, and A. Olivero. A scenario-matching approach to the description and model checking of real-time properties. *IEEE Transactions on Software Engineering*, 31(12):1028–1041, December 2005.

[3] 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.

[4] 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.

[5] W. Grieskamp, N. Kicillof, and N. Tillmann. Action machines: a framework for encoding and composing partial behaviors. Technical Report MSR-TR-2006-11, Microsoft Research, February 2006.

[6] 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.

[7] 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.

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

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

[10] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[11] I. Krueger. Modeling and synthesis with MSC extensions for broadcasting, overlapping, preemptive, and triggered collaborations. In *Workshop on Scenarios and State Machines at ICSE 2003*, 2003.

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

[13] 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.

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

[15] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 109–118, New York, NY, USA, 2002. ACM Press.

[16] 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.