

Behavioral Composition in Symbolic Domains

Wolfgang Grieskamp
Microsoft Research
wrwg@microsoft.com

Nicolas Kicillof^{*}
University of Buenos Aires
nicok@dc.uba.ar

Colin Campbell
Microsoft Research
colin@microsoft.com

ABSTRACT

We report preliminary results toward a framework for composing behavioral models. In our framework, *models* arise from a variety of description techniques such as state machines or scenarios given by textual and diagrammatic notations. Such models may describe full system behavior or *aspects* of system behavior representing a given concern. Our models may be *composed* with each other and *transformed* by various operators, yielding models that are themselves composable and transformable. We can use the models, either individually or in composition, in a variety of analysis processes such as model checking, refinement checking and model-based testing. What enables the flexibility of composition is the *symbolic representation* of values and state. Our approach is currently being incorporated into an advanced model-based specification and testing environment at Microsoft Research.

1. INTRODUCTION

A practical industrial modeling process for software depends even more on the application of the principle of separation of concerns [7] than the coding stage does. The reason is that we usually find a larger variety of stakeholders in the modeling phase than during coding, with more viewpoints on the developed artifact and hence different concerns. Thus, effective specification is necessarily focused on chosen levels of abstraction, usually more than one. Moreover, in modeling we need to capture not only the final product but also the steps that lead to this product, from requirements' analysis to system design, component design and test.

At Microsoft Research, we have in recent years developed a tool environment for model-based testing called Spec Explorer [4]. This environment allows users to model object-oriented, reactive software, analyze the specification with model-checking techniques and derive model-based tests. The feedback of Spec Explorer users at Microsoft shows that

^{*}Currently visiting Microsoft Research

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 transition 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-machine modeling and scenario-oriented (interaction-based) description techniques. Moreover, users want to be able to combine the models resulting from these different notations by model composition. For example, one common usage is the composition of a state machine model with a scenario that describes a test purpose to be extracted from the state machine. 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 from users, 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 process. 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 model intersection, 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 the model's behavior. These transformations are suitable for both online (on-the-fly) and off-line testing.

Our implementation of this framework uses XRT [11] (eXploring Runtime), a software model-checker and simulation framework for .NET, which supports symbolic state representation. The behavioral constraints that act as the “glue” for composition are sketched in preliminary form in [10] as *action machines*, along with several basic composition operators. In this paper, we refine the concepts described in [10], and present new composition operators which have not been published before.

We see the *contributions* of our work in the context of aspect-oriented modeling (AOM) as follows. Our approach indicates that there is no fundamental difference in the for-

malisms required to represent “primary” models and “aspect” models; the later are just more partial descriptions of system behavior than the former. Since we can interpret a model starting from an arbitrary symbolic state, we can analyze and validate partial models not only in composition, but also stand-alone; which by the separation of concern principle contributes to a better understanding of the overall problem domain. Moreover, our first experiments indicate that the major notation families for behavioral description, namely state-machine-based and scenario-based (interaction-based) description techniques, can be both used for modeling aspects. This is achieved in our framework by mapping both notational styles into the uniform representation of action machines

2. ACTION MACHINES

Models of object-oriented reactive programs are uniformly represented in our framework by *action machines* [10]. Composition operators take action machines and deliver new actions machines.

2.1 Basics

A mathematical foundation of action machines is given by a variation of labeled-transition systems in [10]. Here, we present a high-level sketch.

An action machine is defined by an initial state from which the behavior of the machine can be transitively unfolded. Each state provides an enumeration of *steps*, describing those state transitions which the machine can make from the given state. States may be marked as *accepting*, i.e. they represent the end of a full run of the machine. States are *symbolic* and consist of an *explicit* part and a *constraint*. They can be as rich as the full underlying .NET programming model (i.e. including heap, threads, and so on), as supported by XRT [11].

A step consists of an action label and a target state. In our application, an action label typically describes the “occurrence” of a method invocation in the modeled program, such as $o.m(x_1, \dots, x_n)/y$; where m is a static or instance method, o is the receiver object (if m is instance-based), x_i s are input parameters, and y is the result of the method invocation. All of o , x or y can be *symbolic* values, i.e. terms in the underlying symbolic computation domain which contain free logical variables.

We can sometimes unify the states resulting from two steps of different action machines. This is the major instrument for defining composition. Two states unify if (a) the explicit state components unify pointwise in their overlapping portions, and (b) after applying the substitution resulting from that unification, the conjunction of the constraints is satisfiable. The result of the unification consists of the explicit state parts with overlapping parts identified, and the conjunction of the symbolic state parts.

Consider, as an example, two states $S_1 = (\{v_1 \mapsto 1, v_2 \mapsto x\}, x > 0)$ and $S_2 = (\{v_1 \mapsto 1, v_2 \mapsto x'\}, x' \leq 1)$, where v_i are integer state variables with a given assignment and x, x' are logical variables. The unification of those states is given as $S_1 \wedge S_2 = (\{v_1 \mapsto 1, v_2 \mapsto x\}, x > 0 \wedge x' \leq 1 \wedge x = x')$, which is equivalent to $(\{v_1 \mapsto 1, v_2 \mapsto 1\}, x = 1 \wedge x' = 1)$.

A typical composition of action machines is the intersection. Two action machines can step together in an intersection if (a) the action labels of the step unify, and (b) the target states unify. Consider, for example, an action machine M_1 which steps from state $S_0 = (\{v_1 \mapsto 1, v_2 \mapsto 0\}, true)$,

via action $a(x)$, to the state S_1 described above. Obviously, this machine performs the update $v_2 := x$ and adds the constraint $x > 0$. Now consider another machine M_2 which steps from S_0 to S_2 with action $a(x')$; this machine performs the update $v_2 \mapsto x'$ and adds the constraint $x' \leq 1$. Both machines step together in the intersection with $a(1)$ to $S_1 \wedge S_2$. Note that the symbolic parameters to a (x and x') are indirectly bounded to 1 by the unified target state.

We should note that full unification of states in the underlying implementation of action machines can be avoided in all compositions we have investigated so far, which is relevant in order to make our approach feasible for the full richness of states we consider, namely .NET program states. The trick here is to sequentially thread one state through the calculation of a composed step, taking the resulting target state of one step as an input to another step in a composition. This is equivalent to state unification, provided the order in which steps are computed is not relevant, i.e. they commute. In the implementation, we do not actually check for commutativity, but use a deterministic order of step calculation.

An important tool for analyzing action machines is *subsumption exploration*. This is an algorithm that unfolds the behavior of an action machine stopping at states which are subsumed by previously visited states. It is similar to an explicit state model-checking algorithm [5], except that the termination criterion is not state identity but subsumption. A symbolic state subsumes another state if (a) the explicit state components unify pointwise in their overlapping portions, and (b) after applying the substitution resulting from that unification, the constraint of the subsumed state implies the constraint of the subsuming state. In terms of viewing symbolic states as the set of concrete states they characterize, subsumption means that the subsumed state represents a subset of concrete states represented by the subsuming state. An action machine satisfies the condition that in a subsuming state at least those steps are available which are available in the subsumed state. We use subsumption exploration for checking properties on action machines (like that an intersection is not empty, or that a refinement does not contain error states), but also for graphically displaying action machines to the user. The visualization is a state graph where transitions of subsumed states are linked to the subsuming states, representing fixed points as cycles.

2.2 Atomic Action Machines

Atomic action machines can result from various modeling notations, such as diagram notations like state charts and message sequence charts (MSCs), and textual notations like abstract state machines [12], the Spec# notation [1], temporal logic notations, and so on. At the current experimental stage of our framework, we define atomic action machines using C#, either as *actions with preconditions* that can be unwound into state machines, or as scenario programs whose state is control state. We also provide a regular-expression like language to define and combine machines.

In the first notation, we mark normal C# methods as actions using custom attributes. The applicability of these action methods is guarded by assumption statements in their body; if all assumptions are true in a given state, the action constitutes a step in that state, with state updates as performed by the body of the method. This is very similar to the use of Spec# in Spec Explorer. However, in addi-

tion to Spec Explorer capabilities, we can keep parameters of these actions (methods) symbolic. If an action with symbolic parameters is explored for an action machine, and the control flow of the method body depends on the parameters, we actually explore all possible control flow decisions, which results in a step of the action machine for each of the control flow paths under the assumption of the path condition.

In the second currently supported base notation, we describe action machines by scenario programs; the control flow of these programs determines the action machine’s state transitions. To that end, calls to methods marked as actions are “abstracted” in that these actions are not actually invoked but execution is suspended and a step of the action machine is delivered. The input parameters of the step’s action are determined by the input parameters in the scenario program; the result is represented by a free logical variable. Thus action machines resulting from scenario programs allow us to describe control flow patterns of method invocations from an “outside” point of view without ever actually calling those methods, which are kept uninterpreted; this is very similar in spirit to scenario (interaction-based) notations like message sequence charts.

Basic machines like the described ones, but also other ones which would stem from notations like state charts or MSCs [13], can begin operating in different initial states. At one extreme, they start from a given fixed state, where global state is explicitly initialized with concrete values. At the other extreme, they start from an initial state where global state is linked to unconstrained symbolic values. A full range of combinations lie between these extremes. The first case typically represents the situation where we run the machine from the beginning of a system’s lifespan. The second case represents the case where we run the machine at “any” given state. It is particularly useful if the machine describes an aspect which can be invoked in any given state of a system evolution, and needs to be analyzed for all of possible instantiations.

2.3 Action Machine Compositions

In this section, we list the most significant composition operations on action machines we have investigated and implemented so far, and sketch their application.

2.3.1 Intersection

In the intersection of two action machines, written $M_1 \otimes M_2$, a step is performed from a given state only if both machines can perform a step with unifiable actions and unifiable target states. Thus, intersection effectively constraints the behavior which can be exposed by both machines. In our experimental applications, the intersection is useful in restricting a system model to a given purpose for model-based testing; typically, the system model will be given as a state machine, and the test purpose as a scenario. Intersection can also be used for model checking; in that case we would build the negation of a property we want to check (or a “negative” scenario [16, 2]) and intersect that with a model; the property is satisfied if the intersection is empty.

2.3.2 Concatenation

In the concatenation of two action machines, $M_1; M_2$, the accepting states of the first machine are continued with the behavior of the second machine. In our experimental applications, we typically use concatenation to describe various “phases” of a system, e.g., an initialization phase, an op-

eration phase, and a shutdown phase. For the model-based testing context, often phases like initialization and shutdown are described in a scenario style, whereas phases like operation are described by state machines. This is because, in a testing setting, initialization and shutdown phases are usually introduced only in order to reach a certain state of the system, regardless of all the possible ways in which this state can be reached; whereas in the operation phase full system behavior needs to be modeled.

A useful construction that we have defined and implemented together with the composition operators is the *universal machine*, represented as \dots . This action machine exhibits all possible behaviors, i.e. an arbitrary number of steps of all possible actions.

Concatenation and the universal machine give rise to a derived operator, namely *precedence*, written $M_1 \rightarrow M_2$, which is defined as $M_1; \dots; M_2$.

2.3.3 Repetition

In the repetition of an action machine, M^* , the initial state of M is considered an accepting state (in order to admit the empty behavior), and each accepting state of the original M will be recursively continued with M^* , maintaining its accepting state condition. Thus repetition describes an arbitrary number of concatenations of the M machine. Repetition is typically used to describe cyclic systems, where M is the behavior of one cycle. We also support *non-empty repetition*, M^+ .

2.3.4 Choice

A choice, written $M_1 \oplus M_2$, exhibits the behavior of either one operand or the other. This is used to compose, for example, alternative features which can be activated in a given state.

Note that the behavior of an action machine can contain internal nondeterminism, which is important for realizing the full power of the choice operator. The choice may construct behaviors which start with a common prefix and only diverge after a number of steps.

Choice and concatenation give rise to a derived operator, namely *alternation*, written as $M_1 \& M_2$, and defined as $(M_1; M_2) \oplus (M_2; M_1)$. Alternation of several machines yields all permutations. This operator allows us, for example, to abstract the order in which certain features are activated. We also support the *optional* operator, $M?$, which accepts both the behavior of M and the empty behavior.

2.3.5 Interleaving

Interleaving, written $M_1 \parallel M_2$, can perform all consistent interleavings of the steps of both constituting machines. *Consistent* hereby means that if a step of one machine disables consecutive steps of the other, then these steps will not show in the result. Interleaving allows us to define parallel composition of features of system behavior, which may execute independently, possibly with varying results according to the resulting order between actions.

An example application of interleaving in a real-world setting would be the update procedure of an operating system, which runs as a background process, but might at some point need user intervention such as closing running applications. Different interleavings of this task with the ones the user is performing, might yield different results; and thus need to be explored, tested and checked individually. This is enabled by the interleaving operator, which generates a distinct path

for each possible merger.

2.3.6 Translation

Translation, written as $M_1 \uparrow \sigma$, translates the actions of steps in machine M_1 according to rules described by σ . σ is essentially a model morphism, which maps types and actions into other types and actions. The exact definition of σ is currently subject of ongoing exploration. The minimal functionality we provide is allowing to add, erase and reorder parameters of action invocations, and to remap the invoked methods themselves. This is described by rewriting rules $\alpha \Rightarrow \beta$ composing σ , where the sets of variables appearing in α and β do not need to be the same. The meaning is that in each step whose action matches α , the step of the resulting machine will perform β after applying all substitutions resulting from the match. For example, $o.f(a, b)/c \Rightarrow a.g(o, d)/b$ is such a rewriting rule, which remaps the method f to the method g , reordering parameters o , a , and b , erasing c , and introducing a fresh variable d .

Another problem which should be addressed by translation but is currently still under investigation is that of type domain morphisms, namely replacing one type by another type and translating the instances of that types into each other. This is particularly needed for object-identity remappings in model-based testing, as described in [4].

2.3.7 Substitution

The substitution, written $M_1 \leftarrow_{\rho} M_2$, allows to replace steps in M_1 with the behavior of M_2 according to the gluing defined by ρ . ρ consists of a rewriting rule $\alpha \Rightarrow \beta$ and a set of “termination” actions $\gamma_1, \gamma_2, \dots, \gamma_n$. The meaning is as follows. Initially, the resulting machine behaves as M_1 while M_2 is suspended in its initial state. Whenever M_1 can make a step α and M_2 can make the step β resulting in a unified target state, then M_1 is suspended, and M_2 is executed. When M_2 reaches an accepting state via any of the termination actions γ_i , M_1 is resumed and M_2 suspended. This behavior can be seen as if M_1 would “call” into the co-routine M_2 , where communication of input and output parameters is realized via the variables shared between the actions in α , β and γ_i . M_2 acts as a co-routine since it maintains its own internal state between invocations from M_1 .

Substitution is related to the notion of weaving in Aspect Oriented Programming (AOP). Although the current quantification mechanism is limited to enumeration, we are planning on extending this operator (through the use of *triggered scenarios* [15, 16]) to allow complete behavior specifications to identify sets of gluing points (pointcuts). As in AOP’s “around advices”, the original step α of M_1 is hidden in the behavior of the resulting composed machine, and substituted by steps of M_2 . However, α may appear in some of the steps of M_2 , such that the resulting machine appears to the outside as that behavior has been added “around” α , effectively mimicking the “proceed” statement in AspectJ[14].

The design of the substitution machine is still subject of ongoing research in our framework. In our current experimental applications, we use substitution in particular for hierarchical feature composition. Suppose for example one model that references an action representing a `FileOpen` dialog, and another model realizing the detailed behavior of the dialog itself. We can first check and explore both models individually as separate units, and later glue them together

using substitution to further inspect them in composition. Cases like this are particularly interesting for our co-routine-like approach, because the `FileOpen` dialog preserves the last opened location between invocations.

2.3.8 Alternating Refinement

The alternating refinement machine, $M_1 \rightsquigarrow M_2$, serves two purposes: first it checks whether M_2 is an alternating refinement of M_1 , second it represents the behavior of M_2 “in the environment” M_1 . The notion of alternating refinement has been first defined in [3] and further investigated in [6], in the context of “interface automata”. It provides the notion of conformance between a model, and an implementation or another model in Spec Explorer [4].

The basic idea is that the actions in M_1 and M_2 are partitioned into *controllable* and *observable* ones; we can think of the former as inputs and of the latter as outputs of the M_2 machine. We use the notions of controllable and observable instead of inputs and outputs to emphasize the viewpoint of the environment (for example, a “tester”) M_1 onto the machine M_2 .

The machine $M_1 \rightsquigarrow M_2$ checks that, in each state, M_2 can do at least the controllable steps available in M_1 , while M_1 can do at least the observable steps available in M_2 . If this condition is not satisfied for a given state, the refinement machine steps into an error state. Otherwise, the behavior of the resulting machine will be pruned down to the controllable steps of M_1 and the observable ones of M_2 , representing M_2 ’s behavior in the context of the environment M_1 .

2.3.9 Test Suite Unfolding

We view the construction of a test suite from an action machine as a transformation which yields another action machine. That action machine has a tree-like behavior, where each sub-tree starting at the initial state represents one test case, and satisfies the following property: each state in the sub-tree has either exactly one outgoing controllable step, or any number of outgoing observable steps. We call the first kind of states *active*, since they represent the situation where a tester (a human or a program) actively makes a decision; and the second kind of state, *passive*, since the tester waits for a reaction from the system-under-test. Thereby, the signature of the action machine is extended by two pseudo actions to represent transitions between active and passive states: one controllable action called *StartWait* which transitions from an active state into a passive state, representing the decision of the tester to now observe outputs of the system, and one observable action *Timeout(timespan)* which transitions from a passive state into an active one, representing that the tester stops waiting for output.

Test-suite unfolding can be done in various ways. Exhaustive unfolding is intended to capture all behavior of the action machine, which must be finite to that end. Random unfolding makes intelligent random choices on which paths to unfold. Other unfolding techniques use different pruning techniques, which have been developed for Spec Explorer and other Model-Based Testing tools over recent years. All techniques can be used for online (on-the-fly) testing as well as for off-line testing; in the first case, we test the implementation as unfolding proceeds, whereas in the second case we persist a test suite which represents the unfolding.

3. CONCLUSION AND DISCUSSION

We have presented a language-agnostic framework for be-

havioral model composition which provides a rich set of composition operators, including some that resemble concepts from Aspect Oriented Software Development (AOSD). The key technique which enables the power and simplicity of this framework stems from employing symbolic representations for values and states. These allow us to describe partial models of behavior, which can serve as aspects in a more general sense than usually found in AOSD, and can be explored, checked and tested by themselves.

To the best of our knowledge, the use of symbolic representations for gluing the composition of behavioral models has not been used as explicitly as in our framework before. While the composition operators we describe are mostly folklore, coming from process algebra, regular languages, and other sources, they have not been applied for model composition in a symbolic domain.

Aspect Oriented Modeling (AOM) approaches (e.g. [9, 8]) aim at tackling some of the problems we deal with, namely composing several parts of a design that can be created independently. But they usually only deal with cases where a primary or base model expresses the system's core functionality to which some specific modifications are introduced by aspects. Our framework has a wider field of application to any setting in which individual (not necessarily crosscutting) features or concerns need to be combined in one of a variety of manners.

Since the employment of symbolic representation and computation comes not for free in terms of efficiency, we would not claim that our approach can be also applied to the AOP domain. However, we believe it is suitable at least for AOM, and it can be used there for model-checking and model-based testing. Though we haven't yet performed larger case studies, symbolic state in related XRT applications scales well. Ultimately, more experiments are needed to understand the bottlenecks.

As regards to the set of our composition operators and their meaning, further work is required and case studies have to be undertaken to fix and extend the operator set for praxis. As stated above, one major plan is improving the "quantifications" capabilities of our substitution operator, extending them to temporal predicates on the full execution history. We are currently trying to identify the properties of a general symbolic triggering operator on action machines, allowing the definition of rules which match not only a single action, but a sequence of actions.

We are also working on a suitable textual representation of compositions of action machines for our users. A further line of work is the combination of action machines with Microsoft's Domain-Specific Languages (DSL) tools which ship with Visual Studio Team System, in order to enable a framework for the definition of domain-specific diagrammatic behavior notations which map to basic action machines and thus can benefit from composition as well as exploration, checking and testing capabilities.

4. ADDITIONAL AUTHORS

Additional authors: Pritam Roy (University of California, Santa Cruz; currently visiting Microsoft Research), Wolfram Schulte (Microsoft Research), Nikolai Tillmann (Microsoft Research) and Margus Veanes (Microsoft Research)

5. REFERENCES

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

- [2] A. Alfonso, V. A. Braberman, N. Kicillof, and A. Olivero. Visual timed event scenarios. In *ICSE: Proceedings 26th International Conference on Software Engineering*, pages 168–177. IEEE Computer Society, 2004.
- [3] 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.
- [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] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [6] 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.
- [7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [8] G. Georg, R. France, and I. Ray. Composing aspect models. In F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kandé, D. Stein, P. Tarr, and A. Zakaria, editors, *The 4th AOSD Modeling With UML Workshop*, 2003.
- [9] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Commun. ACM*, 44(10):87–93, 2001.
- [10] 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.
- [11] 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.
- [12] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [13] ITU-T. Recommendation Z.120. Message Sequence Charts. Technical Report Z-120, International Telecommunication Union – Standardization Sector, Genève, 2000.
- [14] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.
- [15] B. Sengupta and R. Cleaveland. Triggered message sequence charts. *SIGSOFT Softw. Eng. Notes*, 27(6):167–176, 2002.
- [16] 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.