Composing Model Programs for Analysis

Margus Veanes[a], Jonathan Jacky[b]

[a] *Microsoft Research, Redmond, WA 98052*
[b] *University of Washington, Seattle, WA, 98195*

Abstract

Model programs are high-level behavioral specifications used for software testing and design analysis. Composition of model programs is a versatile technique that, at one end of the spectrum, enables one to build up larger models from smaller ones, and at the other end of the spectrum allows one to restrict larger models to specific scenarios. In this paper we provide a formal foundation for composition of model programs and investigate its use in various situations that arise in model program analysis.

Key words: model program, state machine, labeled transition system, scenario control, model-based testing, model analysis, model validation

1. Introduction

A model program is a kind of executable specification, usually intended as a test oracle or test case generator. A model program describes a labeled transition system (LTS), and plays a role similar to other state-based notations used in model-based testing. An important motivation for model programs has been acceptance by industrial software developers and test engineers. Therefore, a key distinguishing feature is that model programs are coded in the test engineer's usual programming language, often the same language as in the implementation. Model programs can use a library of familiar data types including sets, sequences, and maps (that is, dictionaries) to describe program state.

Composition is a useful and versatile operation that combines two or more model programs to obtain another model program called the product. There is already extensive industrial experience with composing model programs, including a very large project that tested hundreds of protocols [1], and another that tested a web-based communication product [2].

The contributions of this paper are to consolidate and unify earlier results about composition in a common formalism, to extend definitions allowing nondeterminism and not assuming that the model programs are explorable, to show the use of composition in the context of symbolic analysis, and to compare with

Email addresses: `margus@microsoft.com` (Margus Veanes), `jon@washington.edu` (Jonathan Jacky) related work by others. This paper is not an industrial case study, but we do discuss a small example to show how composition is used in an industrial testing tool.

Model programs represent behavior (ongoing activities). A *trace* is a sample of behavior consisting of a sequence of *actions*. An *action* is a unit of behavior, viewed at some level of abstraction. Actions have names and arguments. The names of all of a system's actions are its *vocabulary*. For example, when modeling a network protocol, the vocabulary comprises all the message types, the actions' arguments are the fields in the messages, the actions are the individual messages (including fields), and a trace is a sequence of messages (as might be observed by a network monitor). Traces are central; the purpose of a model program is to generate traces. In order to do this, a model program usually must contain some stored information called its *state*. The model program state is the source of values for the action arguments, and also determines which actions are enabled at any time. When modeling a network protocol, the state might include the set of open connections and all the messages in flight. A model program has an *initial state* where all traces begin, and a set of *accepting states* which are the only states where a trace is allowed to end. Usually these states represent conditions where there is no unfinished work in progress. When modeling a network protocol, the initial state and the accepting states might be those where no connections are open and no messages are in flight.

Recall that composition combines two or more model programs to obtain another model program called the product. The effect of composition is to synchronize shared actions (that appear in more than one of the composed model programs) and to interleave unshared actions (that only appear in one). Composition has several useful applications. For example, it can be used to validate a model program: to show that it accurately represents the intended behaviors. Composition can be used to check whether a system can execute a particular scenario. Any temporal property than can be expressed as a finite state machine can be checked in this way. For another example, almost any automatic test generation method will generate too many tests unless there is something to prevent it. Composition can provide scenario control to focus on issues of interest and eliminate redundant test cases. Composition can also assist symbolic analysis of model programs by pruning the search space during proof search.

The formalization of the parallel composition of model programs builds on the classical theory of LTSs [3]. Our goal is therefore *not* to define yet another notion of composition but to show how the composition of model programs can be defined in a way that preserves the underlying LTS semantics. It is important to note here that the composition of model programs is *syntactic*. It is effectively a program transformation that is most interesting when it is formally grounded in an existing semantics and has useful algebraic properties. This fills an important semantic gap and makes compositional modeling more practical in tools like Spec Explorer and NModel.

In Section 2 we define the background theory $\mathcal{T}$ and we define model programs formally. In Section 3 we define composition of model programs. In

Section 4 we discuss some pragmatics of implementing and using composition in a software analysis and testing tool. In Section 5 we show how to use composition to check temporal properties. In Section 6 we show how to use composition for scenario control in model-based testing. In Section 7 we illustrate the use of composition as a way to reduce symmetries in symbolic analysis. Section 8 discusses related work.

2. Model programs

In this section we define the background theory and define model programs formally. Model programs are defined here over a fixed background theory $\mathcal{T}$ that includes linear arithmetic, Booleans, tuples, and sets. The universe is multi-sorted, with all values having a fixed *sort*. The background is adequate for the purposes of this paper, although it disallows for example nested sets (sets of sets) that are sometimes used in high-level models. We impose this restriction to provide a self-contained axiomatization for a large class of model programs, that is suitable for both symbolic analysis as well as explicit state analysis techniques. Many common data types, such as maps, can be defined in terms of $\mathcal{T}$ and we do so when this is needed. Recent advances in Satisfiability Modulo Theories make it possible to analyze expressive fragments of $\mathcal{T}$.

We use an expression language that we also refer to as $\mathcal{T}$. Well-formed expressions or terms of $\mathcal{T}$ are shown in Figure 1. We do not add explicit sort annotations to terms but always assume that they are well-sorted.

The background axioms of $\mathcal{T}$ includes the axioms of linear arithmetic, the axioms for tuples, the axioms for the set operations as shown in Figure 1, and the extensionality axiom for sets. The expression $\mathit{Ite}(\varphi, t_1, t_2)$ equals t_1 if φ is true, and it equals t_2, otherwise. For each sort, there is a specific *Default* value in the background. In particular, for Booleans the value is *false*, for set sorts the value is $\emptyset$, for integers the value is 0 and for tuples the value is the tuple of defaults of the respective tuple elements. The function *TheElementOf* maps every singleton set to the element in that set and maps every other set to *Default*. A set comprehension term $\{t[x] \mid_x \varphi[x]\}$ denotes the set of all $t[a]$ such that $\varphi[a]$ holds. Note that the use of set comprehensions as terms is justified by the *extensionality axiom* for sets: $\forall v\, w\, (\forall y (y \in v \leftrightarrow y \in w) \rightarrow v = w)$. We write $FV(t)$ for the set of free variables in t.

There is also a specific *action sort* $\mathbb{A}$, values of this sort are called *actions*. Two actions are equal if and only if they have the same action symbol and their corresponding arguments are equal. An action with action symbol f is called an f*-action*. If the arity n of an action symbol f is positive, we assume that f is associated with a unique variable f_i, for all i, $0 \leq i < n$, called the i-th *parameter variable* of f.

An *assignment* is a pair $x := t$ where x is a variable and t is a term (both having the same sort). An *update rule* is a finite set of assignments where all the assigned variables are distinct.

$$
\begin{array}{lll}
T^{\sigma} & ::= & x^{\sigma} \mid \mathit{Default}^{\sigma} \mid \mathit{Ite}(T^{\mathbb{B}}, T^{\sigma}, T^{\sigma}) \mid \mathit{TheElementOf}(T^{\mathbb{S}(\sigma)}) \mid \\
 & & \pi_i(T^{\sigma_0 \times \cdots \times \sigma_{i-1} \times \sigma \times \cdots \times \sigma_k}) \\
T^{\sigma_0 \times \sigma_1 \times \cdots \times \sigma_k} & ::= & \langle T^{\sigma_0}, T^{\sigma_1}, \ldots, T^{\sigma_k} \rangle \\
T^{\mathbb{Z}} & ::= & k \mid T^{\mathbb{Z}} + T^{\mathbb{Z}} \mid k * T^{\mathbb{Z}} \\
T^{\mathbb{B}} & ::= & \mathit{true} \mid \mathit{false} \mid \neg T^{\mathbb{B}} \mid T^{\mathbb{B}} \wedge T^{\mathbb{B}} \mid T^{\mathbb{B}} \vee T^{\mathbb{B}} \mid \forall x\, T^{\mathbb{B}} \mid \exists x\, T^{\mathbb{B}} \mid \\
 & & T^{\sigma} = T^{\sigma} \mid T^{\mathbb{S}(\sigma)} \subseteq T^{\mathbb{S}(\sigma)} \mid T^{\sigma} \in T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{Z}} \leq T^{\mathbb{Z}} \\
T^{\mathbb{S}(\sigma)} & ::= & \{T^{\sigma} \mid_{\bar{x}} T^{\mathbb{B}}\} \mid \emptyset^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \cup T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \cap T^{\mathbb{S}(\sigma)} \mid \\
 & & T^{\mathbb{S}(\sigma)} \setminus T^{\mathbb{S}(\sigma)} \\
T^{\mathbb{A}} & ::= & f^{(\sigma_0, \ldots, \sigma_{n-1})}(T^{\sigma_0}, \ldots, T^{\sigma_{n-1}})
\end{array}
$$

Figure 1: Well-formed expressions in $\mathcal{T}$. Sorts are shown explicitly here. An expression of sort σ is written T^{σ}. The sorts $\mathbb{Z}$ and $\mathbb{B}$ are for integers and Booleans, respectively, k stands for any integer constant, x^{σ} is a variable of sort σ. The sorts $\mathbb{Z}$ and $\mathbb{B}$ are *basic*, so is the *tuple sort* $\sigma_0 \times \cdots \times \sigma_k$, provided that each σ_i is basic. The *set sort* $\mathbb{S}(\sigma)$ is not basic and requires σ to be basic. All quantified variables are required to have basic sorts. The sort $\mathbb{A}$ is called the *action sort*, $f^{(\sigma_0, \ldots, \sigma_{n-1})}$ stands for an *action symbol* with fixed arity n and argument sorts $\sigma_0, \ldots, \sigma_{n-1}$, where each argument sort is a set sort or a basic sort. The sort $\mathbb{A}$ is *not* basic. The only atomic relation that can be used for $T^{\mathbb{A}}$ is equality. $\mathit{Default}^{\mathbb{A}}$ is a nullary action symbol. Boolean expressions are also called *formulas* in the context of $\mathcal{T}$. In the paper, sort annotations are mostly omitted but are always assumed.

Definition 1. A *model program* is a tuple $P = (\Sigma, \Gamma, \varphi^0, R)$, where

- Σ is a finite set of variables called *state variables*;
- Γ is a finite set of action symbols called *vocabulary*;
- φ^0, $FV(\varphi^0) \subseteq \Sigma$, is a formula called the *initial state condition*;
- R is a collection $\{R_f\}_{f \in \Gamma}$ of *action rules* $R_f = (\gamma, U, X)$, where, let $V = \Sigma \cup X \cup \{f_i\}_{i < \mathrm{arity}(f)}$,
  - γ is a formula called the *guard of* f, $FV(\gamma) \subseteq V$;
  - U is an update rule $\{x := t_x\}_{x \in \Sigma_f}$ for some $\Sigma_f \subseteq \Sigma$, $FV(t_x) \subseteq V$,
  - X is a set of variables, disjoint from Σ, called *choice variables of* f, each $\chi \in X$ is associated with a formula $\exists x \varphi[x]$, called the *range condition* of χ, denoted by $\chi^{\exists x \varphi[x]}$, $FV(\exists x \varphi[x]) \subseteq V \setminus \{\chi\}$.

We indicate the component of a model program P by adding the subscript P to it, e.g., Σ_P is the set of state variables of P.

We often say *action* to also mean an action rule or an action symbol, if the intent is clear from the context. When an action does not use choice variables, we abbreviate the action rule $(\gamma, U, \emptyset)$ by (γ, U), and when the update rule is also empty we abbreviate it further by its guard γ.

2.1. Accepting states

A model program P may also be associated with an *accepting state condition* φ_P^{acc} such that $FV(\varphi_P^{acc}) \subseteq \Sigma_P$. In Definition 1 this component has been omitted, and is instead assumed to be represented by a *unique* action symbol *acc* with arity 0 and the associated action rule φ_P^{acc}. Intuitively, an accepting state is "labeled by *acc*". One may assume, without loss of generality, that $acc = \mathit{Default}^{\mathbb{A}}$.

2.2. Choice variables

They are "hidden" parameter variables, the range condition of a choice variable determines the valid range for its values. For parameter variables, the range conditions are typically part of the guard. If the choice variable has a set sort, it is assumed to be a map (see below), and the range condition must hold for the elements in the range of that map.

2.3. Maps

We assume here a standard representation of maps as function graphs. A *map* $m = \{k_i \mapsto v_i\}_{i<\kappa}$ is represented as a set of *key-value* pairs $\{\langle k_i, v_i\rangle\}_{i<\kappa}$. Updating a map m with a key-value pair $\langle k, v\rangle$ produces a new map that is the same as m except that k maps to v.

$$\mathit{Update}(m,k,v) \overset{\text{def}}{=} \{e \mid e \in m \wedge \pi_0(e) \neq k\} \cup \mathit{Ite}(v = \mathit{Default}, \emptyset, \{\langle k, v\rangle\})$$

Given a set u of tuples, we write $\pi_i(u)$ for $\{\pi_i(x) \mid x \in u\}$. The definition of $\mathit{Update}(m,u)$, where u is a set of key-value pairs (where no key occurs twice, but where some value may be *Default*) is analogous:

$$\mathit{Update}(m,u) \overset{\text{def}}{=} \{e \mid e \in m \wedge \pi_0(e) \notin \pi_0(u)\} \cup \{e \mid e \in u \wedge \pi_1(u) \neq \mathit{Default}\}$$

Lookup of a value based on a key is defined as follows.

$$\mathit{Lookup}(m,k) \overset{\text{def}}{=} \mathit{TheElementOf}(\{v \mid \langle k, v\rangle \in m\})$$

We also write $m(k)$ as a shorthand for $\mathit{Lookup}(m,k)$. Note that maps are extensional, since keys that are mapped to *Default* are removed from the map, i.e., given two maps m_1 and m_2:

$$m_1 = m_2 \Leftrightarrow \forall k(m_1(k) = m_2(k))$$

Note that, if the default value is not removed from the range, then, for example $m_1 = \{\langle 1, \mathit{Default}\rangle\} \neq \emptyset = m_2$ but $\forall k(m_1(k) = m_2(k))$. A map m is *finite* if $m(k) = \mathit{Default}$ for all but finitely many k. When maps are used as state variables they are typically finite.

Example 1. The following model program, called *Credits*, is written in AsmL. It specifies how a client and a server need to use message identifiers, based on a sliding window protocol. Here we illustrate the components of the *Credits* model program according to Definition 1.

```
var window as Set of Integer = {0}
var maxId as Integer = 0
var requests as Map of Integer to Integer = {->}

[Action]
Req(m as Integer, c as Integer)
  require m in window and c > 0
  requests := Add(requests,m,c)
  window := window difference {m}

[Action]
Res(m as Integer, c as Integer)
   require m in requests and requests(m) >= c and c >= 0
   window := window union {maxId + i | i in {1..c}}
   requests := RemoveAt(requests,m)
   maxId := maxId + c
```

The three state variables are indicated with the keyword `var`. Note that the sort of *requests* is $\mathbb{S}(\mathbb{Z} \times \mathbb{Z})$.

The two actions *Req* and *Res* are indicated with the `[Action]` attribute on the corresponding method definition. The parameter variables of the *Req*-action are $Req_0 = m$ and $Req_1 = c$ (assuming standard conventions for naming and scoping of formal parameters of methods), similarly for the *Res*-action.

The initial state condition is given by the initial assignment of values to the state variables, i.e., $window = \{0\} \wedge maxId = 0 \wedge requests = \emptyset$.

The *Req*-action has the following action rule. The guard of the *Req*-action is $Req_0 \in window \wedge Req_1 > 0$, and the update rule is

$$\{requests := Update(requests, Req_0, Req_1),\ window := window \setminus \{Req_0\}\}.$$

Similarly, the guard of the *Res*-action is

$$requests(Res_0) \neq Default \wedge requests(Res_0) \geq Res_1 \wedge Res_1 \geq 0,$$

and the update rule of the *Res*-action consists of three assignments:

$$\begin{array}{rcl} window & := & window \cup \{maxId + i \mid 1 \leq i \wedge i \leq Res_1\} \\ requests & := & Update(requests, Res_0, Default) \\ maxId & := & maxId + Res_1 \end{array}$$

Neither action uses choice variables.[1] ⊠

2.4. Representing ASMs as model programs

Standard *ASM update rules* [4] can be translated into update rules of model programs. A detailed translation from standard ASMs to model programs is

[1]In AsmL choice variables are introduced by using the *choose*-construct.

given in [5]. We are omitting the details of this translation here and only provide the high-level intuition.

The translation from an ASM update rule U uses a function $\mathbf{u}(U,g)$, defined by induction over the structure of U, that, for each dynamic function g in a signature $\Sigma^{\mathrm{dynamic}}$ of dynamic ASM functions, creates a term in $\mathcal{T}$ that represents the set of updates applied to g. Typically, this term is a comprehension term. In the corresponding model program, g is a map-valued state variable. The model program has an additional Boolean state variable *inconsistent*, the purpose of this variable is to represent states resulting from an inconsistent state update.

$$\begin{aligned} \mathit{IsInconsistent}(U,g) &\stackrel{\mathrm{def}}{=} \exists\, x\, y\, z(\langle x,y\rangle \in \mathbf{u}(U,g) \wedge \langle x,z\rangle \in \mathbf{u}(U,g) \wedge y \neq z) \\ \mathit{IsInconsistent}(U) &\stackrel{\mathrm{def}}{=} \bigvee_{g\in\Sigma^{\mathrm{dynamic}}} \mathit{IsInconsistent}(U,g) \end{aligned}$$

In general, the translation of an ASM update rule U yields the corresponding model program update rule

$$\begin{aligned} \mathit{inconsistent} &:= \mathit{IsInconsistent}(U) \\ g &:= \mathit{Ite}(\mathit{IsInconsistent}(U), g, \mathit{Update}(g, \mathbf{u}(U,g))) \quad (g \in \Sigma^{\mathrm{dynamic}}). \end{aligned}$$

The translation introduces choice variables, in case the ASM update rule U uses choose-statements. The translation motivates why the background $\mathcal{T}$ uses sets and why dynamic functions are represented as function graphs, namely, to encode the update semantics of ASMs. A further motivation for this representation comes from the partial update semantics of AsmL, where total as well as partial updates [6] are allowed, and where dynamic functions are represented by maps. Moreover, the representation in $\mathcal{T}$ provides a basis for several symbolic analysis techniques, see Section 8. The model program representation of ASMs also brings action based ASM models closer to Event-B models [7], that are discussed in Section 8, and may provide some insights into combining both modeling approaches.

2.5. Semantics of model programs

A model program describes a *labeled transition system* or *LTS*, that is a tuple $(\mathbf{S}, \mathbf{S}_0, L, R)$, where $\mathbf{S}$ is a set of *states*, $\mathbf{S}_0 \subseteq \mathbf{S}$ is a set of *initial states*, L is a set of labels and $R \subseteq \mathbf{S} \times L \times \mathbf{S}$ is a *transition relation*.

A (Σ-)*state* is a mapping of variables (in Σ) to values. Given a state S and an expression E, E^S is the *evaluation of* E *in* S. Given a state S and a formula φ, $S \models \varphi$ means that φ is true in S. *Since* $\mathcal{T}$ *is assumed to be the background we usually omit it, and assume that each state also has an implicit part that satisfies* $\mathcal{T}$*, e.g. that* $+$ *means addition and* $\cup$ *means set union.* In the following definitions we assume a fixed model program $P = (\Sigma, \Gamma, \varphi^0, R)$. We assume here that all actions of P have assignments for all state variables (by adding trivial assignments $x := x$ for all state variables x that are not assigned).

Definition 2. Let $a = f(b_0, \ldots, b_{n-1})$ be an action with rule $(\gamma, U, \{x_i^{\exists x \varphi_i}\}_{i<m})$; a is *enabled* in a state S if there exists a state

$$S_1 = S \cup \{f_i \mapsto b_i\}_{i<n} \cup \{x_i \mapsto c_i\}_{i<m} \quad \text{(for some } c_i\text{)},$$

such that $S_1 \models \gamma$ and $S_1 \models (\exists\, x\, \varphi_i[x]) \Rightarrow \varphi_i[x_i]$ (for all $i < m$), in which case a *causes a transition* from S to $S_2 = \{x \mapsto t^{S_1}\}_{x:=t \in U}$.

Note that if a range condition $\exists x \varphi_i$ is false then any value for x_i is valid. The rationale behind this is that range conditions may reflect conditions in nested contexts, which may be false. In case of a top-level choice, the range condition is typically part of the guard γ.

Example 2. Consider the following model program in AsmL.

```
var z as Integer
[Action]
f(k as Integer)
  require true
  if k > 3
    choose y | y in {4..k}
      z := y
```

The guard of f is *true*, so the action is enabled in all states. The translation in Section 2.4 yields the update rule $z := \mathit{Ite}(f_0 > 3, y, z)$ for f, where y is a choice variable with the range condition $\exists x\, 4 \leq x \leq f_0$. Note that the range condition is false for the action $f(2)$. ⊠

Given a finite sequence of transitions $(\tau_i)_{i<k}$, where $\tau_i = (S_i, a_i, S_{i+1})$ and a_i causes a transition from S_i to S_{i+1} for $i < k$, we write $S_0 \xrightarrow{\alpha} S_k$, where $\alpha = (a_i)_{i<k}$, and we write $S_0 \xrightarrow{\tau} S_k$, where $\tau = (\tau_i)_{i<k}$.

Definition 3. $[\![P]\!]$ is the LTS $(\mathbf{S}, \mathbf{S}_0, L, R)$, where $\mathbf{S}_0$ is the set of Σ-states where φ^0 is true, L, R and $\mathbf{S}$ are the least sets such that, $\mathbf{S}_0 \subseteq \mathbf{S}$, and if $S \in \mathbf{S}$ and a is an action such that $S \xrightarrow{a} S'$ then $a \in L$, $S' \in \mathbf{S}$ and $(S, a, S') \in R$.

We say that a state is *reachable* in P if it belongs to the set of states of $[\![P]\!]$.

Definition 4. A *run* of P is a sequence of transitions $(S_i, a_i, S_{i+1})_{i<\kappa}$ in $[\![P]\!]$, for some $\kappa \leq \omega$, where S_0 is an initial state of $[\![P]\!]$. The sequence $(a_i)_{i<\kappa}$ is called an (*action*) *trace* of P. The set of runs and traces of P is denoted by $\mathcal{R}(P)$ and $\mathcal{L}(P)$, respectively.

Intuitively, an action trace is an abstraction of a run. For most parts of the paper we are interested in traces. Runs are important in the context of symbolic reasoning about model programs. When a model program has a unique initial state and is deterministic (for each state S and action a, there is at most one transition $S \xrightarrow{a} S'$) then runs and traces are obviously in a one-to-one correspondence and can be treated synonymously. This is the case in the context of NModel [8].

3. Model program composition

Under composition, model programs with the same action signature synchronize their steps for the actions. The guards of the actions in the composition are the conjunctions of the guards of the component model programs. The update rules are compositions of the update rules of the component model programs. Given two update rules U_1 and U_2 with assigned variables V_1 and V_2, respectively, let

$$\begin{aligned} U_1 \otimes U_2 \ \stackrel{\text{def}}{=}\ & \{v := t \mid v := t \in U_1, v \in V_1 \setminus V_2\} \cup \\ & \{v := t \mid v := t \in U_2, v \in V_2 \setminus V_1\} \cup \\ & \{v := \mathit{Ite}(t_1 = t_2, t_1, \mathit{Default}) \mid v := t_1 \in U_1, v := t_2 \in U_2, \\ & \qquad v \in V_1 \cap V_2\} \end{aligned}$$

In the case when a shared state variable is assigned two distinct values we resort to the default value. A more general approach is to combine the values in a consistent manner, or if no such combination is possible, to enter an error state in the style discussed in Section 2.4. For our primary applications such extensions are not relevant. Let P and Q be fixed model programs.

$$\begin{aligned} P &= (\Sigma_P, \Gamma_P, \varphi^0_P, (\gamma_{f,P}, U_{f,P}, X_{f,P})_{f\in\Gamma}) \\ Q &= (\Sigma_Q, \Gamma_Q, \varphi^0_Q, (\gamma_{f,Q}, U_{f,Q}, X_{f,Q})_{f\in\Gamma}) \end{aligned}$$

Definition 5. Assume $\Gamma = \Gamma_P = \Gamma_Q$, the *product* $P \otimes Q$ is the following model program where $U_f = U_{f,P} \otimes U_{f,Q}$, and $X_f = X_{f,P} \cup X_{f,Q}$.

$$P \otimes Q \stackrel{\text{def}}{=} (\Sigma_P \cup \Sigma_Q, \Gamma, \varphi^0_P \wedge \varphi^0_Q, (\gamma_{f,P} \wedge \gamma_{f,Q}, U_f, X_f)_{f\in\Gamma}).$$

The following facts follow immediately.

$$\begin{aligned} [\![P \otimes Q]\!] &= [\![Q \otimes P]\!] \\ [\![(P \otimes Q) \otimes P']\!] &= [\![P \otimes (Q \otimes P')]\!] \end{aligned}$$

Note that a state of the product $P \otimes Q$ is accepting iff it is accepting in both P and Q, i.e., if $\mathit{acc} \in \Gamma$ then $\varphi^{\mathit{acc}}_{P\otimes Q}$ is $\varphi^{\mathit{acc}}_P \wedge \varphi^{\mathit{acc}}_Q$.

3.1. Unshared actions

The *enabling* (*disabling*) *extension of* P for a set of action symbols F not in Γ_P is denoted by P^{+F} (P^{-F}). If $f \in F$, then in P^{+F} (P^{-F}) the action rule of f is *true* (*false*). We extend the definition of product to the case where P and Q may have unshared actions symbols as follows.

$$P \otimes Q \ \stackrel{\text{def}}{=}\ [\![P^{+\Gamma_Q\setminus\Gamma_P} \otimes Q^{+\Gamma_P\setminus\Gamma_Q}]\!]$$

Intuitively, unshared actions are *interleaved*. Note the following consequence of this definition in relation to accepting states. If for example P has no accepting state condition, i.e., $\mathit{acc} \in \Gamma_Q \setminus \Gamma_P$, then by default all states of P are treated as accepting states in the product. If there is a need to consider all states of P as non-accepting states then one can use the model program $P^{-\{\mathit{acc}\}}$.

3.2. Avoiding emergent behavior

When product composition is used in an unrestricted manner then the product is a new model program which may have traces that occur in neither of the components of the product. In the context of complex systems this is in general referred to as *emergent behavior* [9]. In the context of model program composition, emergent behavior may occur when the components share state variables. The following theorem characterizes the composition of model programs with disjoint state variables. We define *parallel composition*

$$P \parallel Q \overset{\text{def}}{=} P \otimes Q \qquad (\text{assuming } \Gamma_P = \Gamma_Q \text{ and } \Sigma_P \cap \Sigma_Q = \emptyset)$$

Given an action sequence $(a_i)_{i<\kappa}$ for some $\kappa \leq \omega$, and a runs

$$\tau_M = (S_{i,M}, a_i, S_{i+1,M})_{i<\kappa} \text{ of } M, \text{ for } M = P, Q,$$

let

$$\tau_P \parallel \tau_Q \overset{\text{def}}{=} (S_{i,P} \cup S_{i,Q}, a_i, S_{i+1,P} \cup S_{i+1,Q})_{i<\kappa}.$$

This definition is lifted to sets of runs in the usual way, let $\alpha(\tau)$ stand for the action trace of a run τ:

$$\mathcal{R}(P) \parallel \mathcal{R}(Q) \overset{\text{def}}{=} \{\tau_P \parallel \tau_Q \mid \tau_P \in \mathcal{R}(P),\ \tau_Q \in \mathcal{R}(Q),\ \alpha(\tau_P) = \alpha(\tau_Q)\}$$

Theorem 1. *$\mathcal{R}(P \parallel Q) = \mathcal{R}(P) \parallel \mathcal{R}(Q)$.*

Proof. Assume $\Gamma = \Gamma_P = \Gamma_Q$ and $\Sigma_P \cap \Sigma_Q = \emptyset$. Since $\Sigma_P \cap \Sigma_Q = \emptyset$, we know that for all $f \in \Gamma$, the action rule of f in $P \otimes Q$ is

$$(\gamma_{f,P} \wedge \gamma_{f,Q}, U_{f,P} \cup U_{f,Q}, X_{f,P} \cup X_{f,Q})$$

It follows easily from the above definitions that $(S_i, a_i, S_{i+1})_{i<\kappa}$ is a run of $P \otimes Q$ if and only if $(S_{i,M}, a_i, S_{i+1,M})_{i<\kappa}$ is a run of M, for $M = P$ and $M = Q$, where $S_i = S_{i,P} \cup S_{i,Q}$. The statement follows. ⊠

Corollary 1. *$\mathcal{L}(P \parallel Q) = \mathcal{L}(P) \cap \mathcal{L}(Q)$.*

The main reason why the corollary is relevant is that it makes it possible to apply compositional reasoning over the traces in the following sense. If all traces of P satisfy a property φ and all traces of Q satisfy a property ψ then all traces of $P \parallel Q$ satisfy both properties φ and ψ.

Example 3. A typical use of composition is for *scenario control.* Suppose we want to consider all the traces of actions of the *Credits* model program where the *Req*-action and the *Res*-action alternate. In other words, we want to restrict $\mathcal{L}(\mathit{Credits})$ with the regular expression $(\mathit{Req}\,\mathit{Res})^*$. Composing *Credits* with following model program $\mathit{FSM}((\mathit{Req}\,\mathit{Res})^*)$ will do the trick.

$$(\{s\}, \{\mathit{Req}, \mathit{Res}, \mathit{acc}\}, s = 0, \{(s = 0, s := 1)_{\mathit{Req}}, (s = 1, s := 0)_{\mathit{Res}}, (s = 0)_{\mathit{acc}}\}$$

$FSM((Req\,Res)^*)$ is essentially the finite automaton:

$$\begin{tikzcd} {} \arrow[r] & \boxed{0} \arrow[r, bend left, "Req"] & 1 \arrow[l, bend left, "Res"] \end{tikzcd}$$

Note that in the product $Credits \otimes FSM((Req\,Res)^*)$ a state is accepting when no *Res*-action is enabled. In a testing scenario this could mean that there is no pending response in the system under test, so the system is stable and can be reset, and a new test case can be started, where a test case is a trace of *Req* and *Res*-actions in the product that ends in an accepting state. ⊠

The construct shown in Example 3 is an example of a frequently used technique to restrict behaviors of a model program P in the context of testing. In general, given a regular expression ρ over Γ, the corresponding model program $FSM(\rho)$ is composed with P.

3.3. Trace restriction

For scenario control, it is sometimes useful to refer to the state variables of a model program in order to write a scenario for it. In other words, there is a contract model program P and there is a scenario model program Q that may read the state variables of P but it may not change the values of those variables. Let $WriteSet(Q)$ be the set of all state variables of Q that are assigned by some action of Q. We define *restriction composition*

$$P \upharpoonright Q \overset{\text{def}}{=} P \otimes Q \qquad (\text{assuming } \Gamma_Q \subseteq \Gamma_P \text{ and } WriteSet(Q) \cap \Sigma_P = \emptyset)$$

Theorem 2. $\mathcal{L}(P \upharpoonright Q) \subseteq \mathcal{L}(P)$.

Proof. Let $F = \Gamma_P \setminus \Gamma_Q$. We know that $P \otimes Q = P \otimes Q^{+F}$. For all $f \in \Gamma_P$, the action rule of f in $P \otimes Q$ is

$$(\gamma_{f,P} \wedge \gamma_{f,Q}, U_{f,P} \cup U_{f,Q}, X_{f,P} \cup X_{f,Q}),$$

since f does not assign to shared state variables. We may assume that each state variable $x \in \Sigma_P$ is assigned in $U_{f,P}$. Assume $(S_i, a_i, S_{i+1})_{i<\kappa}$ is a run of $P \otimes Q$. Let $S_{i,P}$ be the restriction of S_i to the variables Σ_P, for all i. There are two cases.

1. If a_i is an f-action for some $f \in F$ then $\gamma_{f,Q} = true$, $U_{f,Q} = \emptyset$ and $X_{f,Q} = \emptyset$, so $(S_{i,P}, a_i, S_{i+1,P})$ is a transition in $[\![P]\!]$, provided $S_{i,P}$ is reachable.
2. If a_i is an f-action for some $f \in \Gamma_Q$, then there is an extension S_i' of S_i such that $S_i' \models \gamma_{f,P} \wedge \gamma_{f,Q}$ and $S_{i+1} = \{x \mapsto t^{S_i'}\}_{x:=t \in U_{f,P} \cup U_{f,Q}}$. Note that S_i' is also an extension of $S_{i,P}$ and $S_{i+1,P} = \{x \mapsto t^{S_i'}\}_{x:=t \in U_{f,P}}$, because no $x \in \Sigma_P$ is assigned in $U_{f,Q}$. So $(S_{i,P}, a_i, S_{i+1,P})$ is a transition in $[\![P]\!]$, provided $S_{i,P}$ is reachable.

Since $S_{0,P}$ is an initial state of P, it follows that $(S_{i,P}, a_i, S_{i+1,P})_{i<\kappa}$ is a run of P. ⊠

In this case composition of P and Q does not introduce traces that were not traces of P. A typical use of restriction composition is for *guard strengthening* that is illustrated in Example 4. It is not possible to achieve the same effect easily with parallel composition, without duplicating state variables.

Example 4. Consider the following model program, called *MinReq*.

$$\mathit{MinReq} = (\{\mathit{window}\}, \{\mathit{Req}\}, \mathit{true}, \{(\mathit{Req}_0 = \mathit{Min}(\mathit{window}))_{\mathit{Req}}\})$$

The definition of *Min*, in terms of $\mathcal{T}$, is

$$\mathit{Min}(X) \overset{\text{def}}{=} \mathit{TheElementOf}(\{y \mid y \in X \wedge \forall z(z \in X \Rightarrow y \leq z)\}).$$

MinReq requires the first argument of *Req*-action to be the minimal element in *window*. In $\mathit{MinReq}^{+\{\mathit{Res}\}}$ the action rule of *Res* is *true*. The restriction composition $\mathit{Credits} \upharpoonright \mathit{MinReq}$ (with *Credits* from Example 1) strengthens the guard of the *Req*-action with the condition $\mathit{Req}_0 = \mathit{Min}(\mathit{window})$. ⊠

4. Implementation and use

In this section we discuss some pragmatics of implementing and using composition in a tool for software analysis and testing. In particular, we discuss the NModel toolkit [8, 10].

In NModel we distinguish two kinds of model programs: *contract model programs* and *scenario machines*.

A *contract model program* is a complete specification (the "contract") of the system it models. It can generate every trace that the system is allowed to execute, and cannot generate any trace that the system is forbidden to execute.

In NModel, a contract model program is usually written in C#. Its state variables are the variables of the C# program; its initial state is their initial values. Some of its methods are the actions of the model program: the methods labeled with the `[Action]` attribute, similar to Example 1. For each action method, there is an *enabling condition*: a Boolean method that returns true in states where the action is enabled (the enabling condition also depends on the action arguments), similar to the `require` statements in Example 1. The accepting states are defined by another Boolean method.

A *scenario machine* is a model program that does not comprise a complete specification, but only describes a collection of related traces — perhaps just one. A scenario machine might describe the test cases (traces) in a test suite designed to cover a particular slice of functionality. Test engineers create scenario machines to express the use cases they intend to test. Usually some scenario machines can be transcribed directly from examples in specifications or requirements documents. Others are suggested by the test engineers' judgments. In the NModel framework, scenario machines are usually finite state machines (FSMs), expressed in a simple text format that describes the graph of the FSM, similar to Example 3.

Exploration. In NModel, the primary technique for visualizing and analyzing model programs is called *exploration.* Exploration generates a finite state machine (FSM) from a model program. Exploration is, in effect, finite state model checking [11]. The states (nodes) of the FSM are model program states and the transitions (edges) of the FSM are labeled by actions.

Starting at the initial state, the exploration algorithm executes enabled actions, adding actions and states to the FSM as it goes. Exploration terminates when all states have been explored or (more typically) some other stopping condition has been reached. We say that a model program P is infinite if $[\![P]\!]$ is infinite. In general, P is infinite (or too large to generate or store), so exploration can be configured to stop after a given number of transitions have been executed. It is also possible to define a Boolean function on state variables called a *state filter* to exclude states from exploration, for example if a data structure exceeds a given size. The exploration algorithm is *lazy*; it generates each state and transition only when needed. Exploration can be stopped at any time; the generated FSM may be incomplete, but it is correct (in the sense of being a reachable subset or under-approximation of $[\![P]\!]$) as far as it goes.

Composition. Recall that composition is an operation that combines two (or more) model programs to obtain another model program called their *product.* Recall also that the effect of composition is to synchronize shared actions (that appear in the vocabularies of more than one of the composed model programs) and to interleave unshared actions (that only appear in one).

The composition algorithm is, in effect, parallel exploration of all the composed programs. Starting from the initial states in all programs, synchronize on shared actions: at each state add to the product only those shared actions that are enabled in the corresponding states in all programs. Interleave unshared actions: at each state add to the product any unshared actions that are enabled in the corresponding states of any program (recall also Section 3.1). A state is an accepting state in the product if it corresponds to an accepting state in all of the programs.

If all actions are shared, the product is usually smaller than the composed machines, because the product can only contain synchronized actions. If there are unshared actions, the product might be larger, due to interleaving.

Like exploration, the composition algorithm is lazy. This enables a finite scenario machine Q to be explored in parallel with an infinite contract model program P. Their product can be readily computed. It is typical to compose a contract model program with a scenario machine, as illustrated in Example 3. Note also that $P \upharpoonright Q$ is not necessarily finite, but may require further restrictions, e.g., by providing finite ranges for the action parameters with an additional scenario machine.

The following sections discuss an example, a simple client/server that uses TCP/IP sockets. This is not an industrial case study (as in [1] and [2]), but a small working example created for purposes of exposition. The actions in the model program represent calls to the socket API to open and close sockets, make connections, and send and receive messages. A single model program

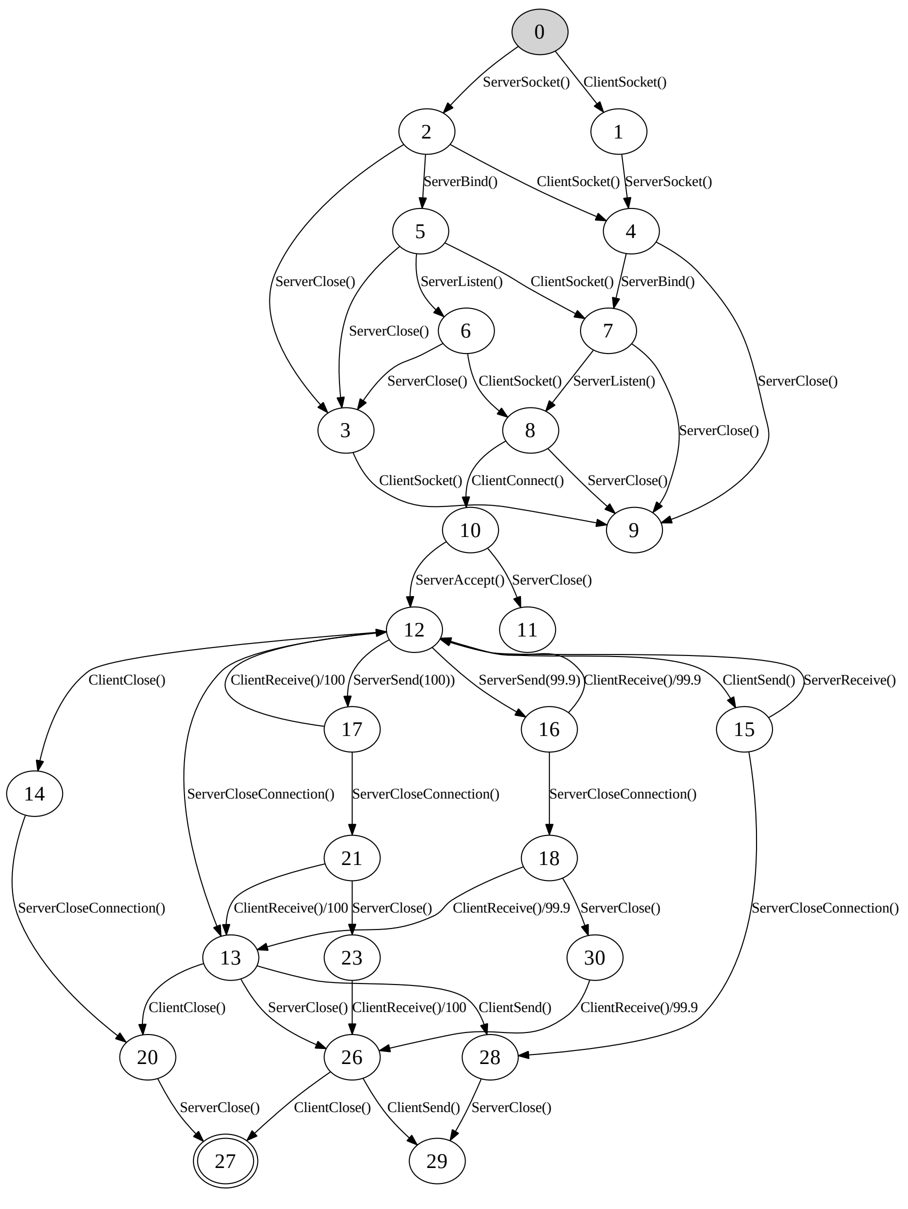

Figure 2: FSM generated by exploring client/server contract model program

represents the client, the server, and the network including messages in flight. This example has been made finite by allowing only one open socket connection, and by limiting message contents to only a few values. Figure 2 shows the FSM generated by exploring this model program. The initial state is gray (top); the accepting state has a double border (bottom). We shall compare the size of this graph (the number of nodes and transitions) and its shape to figures 4 and 5 (below). For our purposes in this paper, the node and transition labels in this figure are not important. (The full contract model program itself is too large to include in the paper; it is included with the NModel software.)

5. Property checking

It is necessary to *validate* a model program: to show that it accurately represents the intended behaviors. Validation usually includes checking whether the model program can execute some traces that are known to be allowed, and cannot execute others that are known to be forbidden.

We can use composition to check whether a model program can execute a particular trace: express the trace as a scenario machine, and compose the two. If the product of the composition reaches an accepting state, the model program can execute that trace; otherwise, it cannot. In effect, the contract model program here acts as an oracle that judges whether the behavior of the scenario machine is allowed.

The left side of figure 3 shows the graph of a scenario machine to be checked by the client/server model program. In this scenario the client and server exchange messages before executing an Accept action. This behavior is forbidden. The right side of the figure shows the product of that forbidden scenario with the contract model program (fig. 2). Each node in the product is labeled by a pair of numbers, the numbers of corresponding states in the scenario machine (left) and the contract model program (compare to fig. 2). The product is a single trace that does not reach an accepting state. The trace stops in a non-accepting state after the client Connect action, which is the last action that is enabled in the corresponding states in both machines. This shows that the contract model program cannot execute this scenario. That is because the scenario describes a forbidden behavior here. In general, when composition reveals that a contract model program cannot execute a plausible scenario, it is advisable to also investigate whether it is the contract model program which is at fault (fails to express the intended behaviors).

This method can be generalized to check any temporal property than can be expressed by a finite state machine.

6. Scenario control

Model programs are often intended as test case generators for *model-based testing.* Almost any automatic test generation method will generate too many tests unless there is something to prevent it. Composition can provide *scenario control* to focus on issues of interest and eliminate redundant test cases.

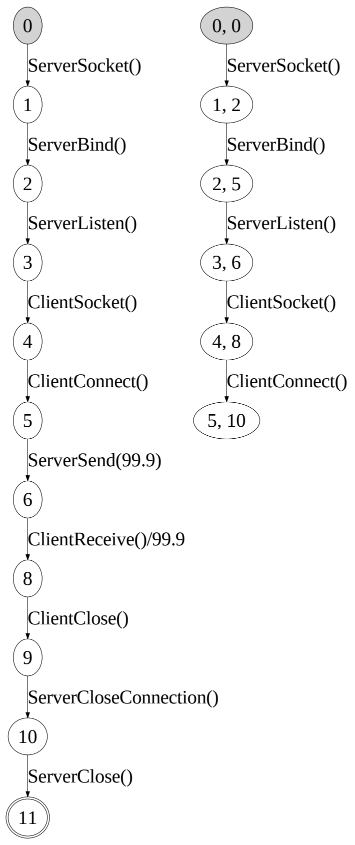

Figure 3: Left: scenario machine, forbidden scenario. Right: product of forbidden scenario machine with contract model program.

One way to generate test cases from a model program is to generate its FSM by exploration, then traverse the FSM. Each path through the FSM from the initial state to an accepting state is a test case. A Postman Tour covers all of the edges in the graph of the FSM, visiting every state and executing every action. Usually several, or many, paths (test cases) are needed to complete the Postman Tour. Figure 4 shows the test suite generated by a Postman Tour of the FSM of the client/server model program shown in Figure 2.

This test suite contains many similar (redundant) test cases because there are many different (but uninteresting) paths through the setup and shutdown portions of the graph, where the client and server are opening and closing their sockets. Each path describes a different interleaving of these client and server actions. The Postman Tour covers all possible interleaving orders. Some of these actions occur during those parts of protocol execution when the client and server are not even connected, so the order of these actions cannot matter;

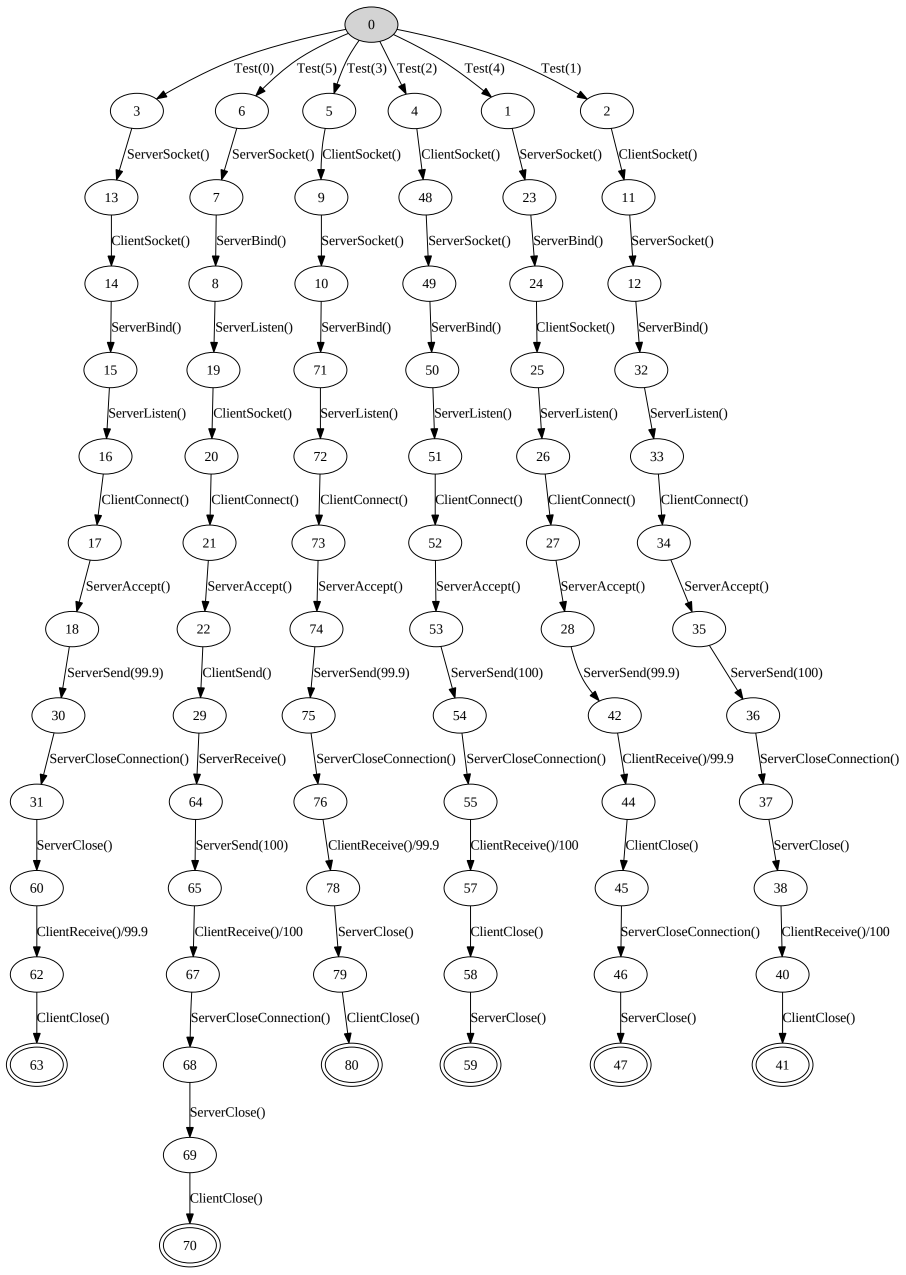

Figure 4: Test suite obtained by traversing the FSM of the contract model program

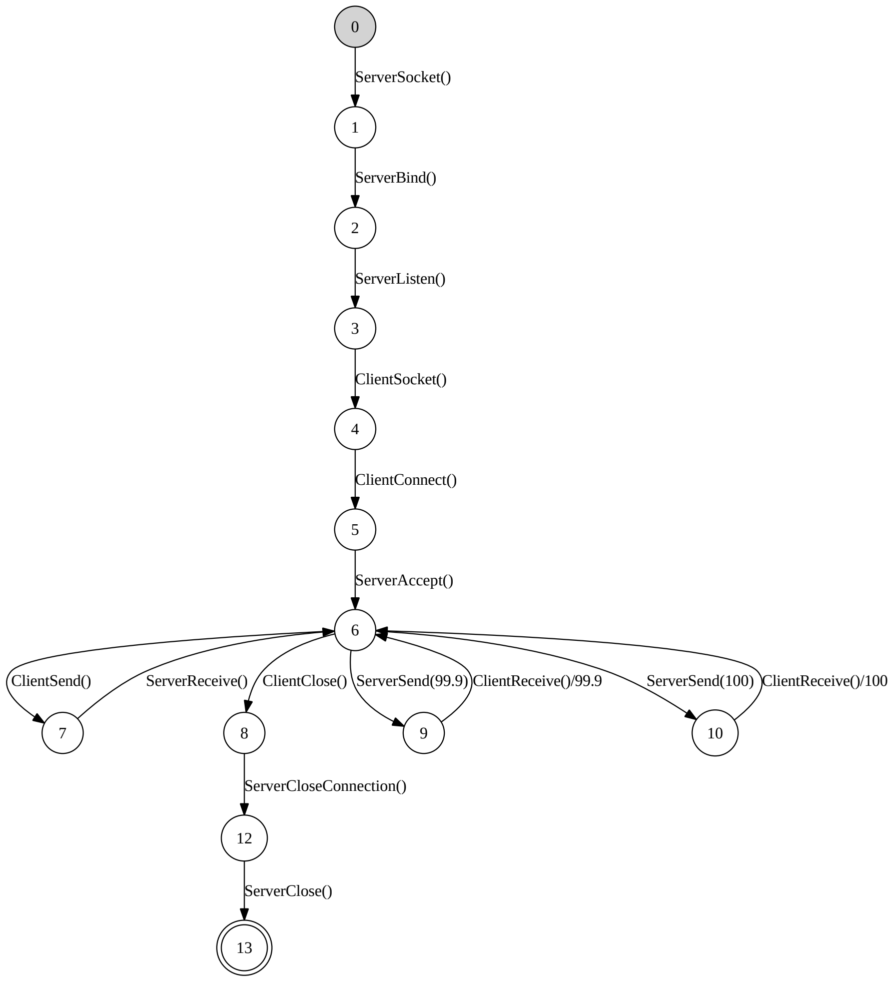

Figure 5: FSM of client/server contract model program composed with test case machine

any single path would provide adequate coverage. In fact, these startup and shutdown portions are executed only in order to reach the interesting part of the graph, where client and server exchange messages.

We compose the contract model program with a test scenario machine to eliminate redundant paths through startup and shutdown. The test scenario machine (not shown here) describes a single path through startup and shutdown. The startup and shutdown actions are shared so the contract model program must synchronize with them. Therefore, only the single path from the scenario machine appears in the product. However, the client and server *send* and *receive*-actions are not present in the scenario machine. They are unshared, so they may interleave freely in the product, limited only by the enabling conditions in the contract model program.

The graph of the product of the composition of the contract model program with the test scenario machine appears in Figure 5 (compare to figs. 2 and 4). There is just one path through startup and shutdown, but several loops through the interesting part of graph where client and server exchange messages. This product machine can be traversed by a single path, because the paths through the several send and receive actions all loop back to the same state. This path corresponds to a single test case.

7. Symbolic reachability checking

Correctness assumptions about a model program can also be expressed through *state invariants*. A state where an invariant is violated is *unsafe*. A part of the model validation process is *safety analysis*, which aims at identifying unsafe states that are reachable from some initial state. The techniques discussed above can also be used for safety analysis. Here we look at a different approach that uses theorem proving. The main advantage of this approach is that the model program does not have to be explorable in the way discussed in Section 4, but may for example include unbounded ranges for action parameters or an initial state condition that allows an unbounded number of initial states. The main disadvantage is that the theorem prover needs to support a rich background theory that may cause the proof search to be very expensive. We show how composition can be used to assist the proof search.

Bounded reachability checking. Bounding the number of steps from the initial state leads to the *bounded reachability checking* problem of model programs: given a model program, a step bound k, and a condition φ, is φ reachable in P from some initial state of P within k steps?

One can construct a formula $BRC(P, \varphi, k)$ in $\mathcal{T}$ from given P, k, and φ such that $BRC(P, \varphi, k)$ is satisfiable in $\mathcal{T}$ if an only if φ is reachable in P from some initial state of P within k steps [5]. Moreover, given $S \models BRC(P, \varphi, k)$, one can extract an initial state S_0 and an action trace α of length $l \leq k$ from S, such that $S_0 \xrightarrow{\alpha} S'$ where $S' \models \varphi$. The formula $BRC(P, \varphi, k)$ can be analyzed using the *satisfiability modulo theories* approach [5, 12, 13] that has been implemented using the SMT solver Z3 [14].

Using composition. In some cases, checking satisfiability of $BRC(P, \varphi, k)$ can be very expensive. One of the core problems is detecting symmetries that arise in proof search due to similarities in the structure of formulas. The below $Bag(n)$ model program is a distilled example that illustrates the symmetry detection problem, that came up as a subproblem in the context of a scheduling problem. There are a number of indexed counters that can be decremented using the action D that takes the index of the counter as an argument and decrements the corresponding counter. In this particular case there are two counters $C(0)$ and $C(1)$, where C is a map, both having the initial value $n > 0$, or one can view C as a bag (multi-set) containing n 0's and n 1's. D deletes one element from the bag.

$$\begin{aligned} Bag(n) \quad = \quad & (\{C^{\mathbb{S}(\mathbb{Z}\times\mathbb{Z})}\}, \{D\}, C = \{\langle 0, n\rangle, \langle 1, n\rangle\}, \\ & \{(C(D_0) > 0, C := \mathit{Update}(C, D_0, C(D_0) - 1))_D\}) \end{aligned}$$

An equivalent AsmL version of the model programs looks like:

```
var C as Bag of Integer = {0 -> n, 1 -> n}
[Action]
D(x as Integer)
  require x in C
  remove x from C
```

We are interested in finding a sequence of actions that exhausts all the counters (empties the bag), i.e., the reachability condition φ is $C = \emptyset$ (recall the definition of *Update* and recall that $\mathit{Default}^{\mathbb{Z}} = 0$). The order of applying the actions $D(0)$ and $D(1)$ is clearly immaterial.

If the step bound k is smaller than $2n$ then $BRC(Bag(n), \varphi, k)$ is clearly unsatisfiable. The execution time of the theorem prover grows exponentially in k in this case (see Table 1). We can use the knowledge that the order of decrementing the different counters is irrelevant and fix such an order by composing $Bag(n)$ with *Order*.

$$\mathit{Order} = (\{x^{\mathbb{Z}}\}, \{D\}, \mathit{true}, (x \leq D_0, x := D_0)_D)$$

The model program *Order* imposes a linear order on the actions where action $D(a)$ has to precede action $D(b)$ if $a < b$. Thus, if $a < b < c$ then traces of *Order* must match the pattern $D(a)^* D(b)^* D(c)^*$.

This use of composition is directly related to partial order reduction [15], that can be achieved by strengthening of the guards of the transitions in the context of symbolic model checking [11]. Note that in the *Bag* example, the actions $\{a, b\} = \{D(0), D(1)\}$ are *independent* in the following sense [16, 17]: 1) for all states S, if a is enabled in S and $S \xrightarrow{a} S_1$, then b is enabled in S iff b is enabled in S_1, and 2) if a and b are both enabled in S then there is a unique state S_1 such that $S \xrightarrow{a,b} S_1$ and $S \xrightarrow{b,a} S_1$. Independent actions can neither disable nor enable each other, and commute when enabled. This notion of independence

Table 1: Satisfiability checking of $BRC(M, C = \emptyset, k)$ with Z3 (version 0.1) for various M and k. Execution time is shown in seconds.

Model program$M$	**Step bound**k	**Verdict**	**Time**
$Bag(5)$	10	Sat	0.14
$Bag(5) \parallel Order$	10	Sat	0.14
$Bag(5)$	9	Unsat	1.5
$Bag(5) \parallel Order$	9	Unsat	0.16
$Bag(8)$	16	Sat	2.2
$Bag(8) \parallel Order$	16	Sat	1.4
$Bag(8)$	15	Unsat	152
$Bag(8) \parallel Order$	15	Unsat	1

provides the starting point for various partial order reduction techniques and, combined with Theorem 1, justifies the use of composition in this particular case. An interesting open problem is how to automate the technique in the context of SMT.

8. Related work

This paper is based on and unifies earlier work related to composition from [8, 12, 18, 19]. In particular, Corollary 1 was originally stated in [18], for a variation of the definition of model programs. The *Credits* example used here is studied in [8, 19] in the context of facet oriented protocol modeling. The use of composition of model programs is the cornerstone of many additional analysis techniques based on explicit state model checking, that are discussed from a practical perspective in [8] and are implemented in the NModel toolkit [10] (also discussed in Sections 4 – 6 here). NModel supports arbitrarily nested data structures, e.g., sets containing maps containing sets, etc, that goes beyond $\mathcal{T}$. During exploration, NModel supports explicit state based symmetry reduction techniques that use graph isomorphism checking [20, 21]. In NModel a model program is scoped by a namespace. Within that namespace, classes can be given a `[Feature]` attribute that declares that class as a feature or submodel program of the full model program. This mechanism can be used to construct separate facet model programs that share state variables, for restriction composition. The main composition operator in NModel is parallel composition, that assumes that the composed model programs do not share state variables. The *FSM* construct is supported in NModel by entering a textual representation of a nondeterministic finite automaton or NFA (e.g. in a text file), that is converted to a finite state model program representing a lazy determinization of the NFA based on the Rabin-Scott algorithm, see e.g. [22, Theorem 2.1]. The *FSM* construct is related to a more general coordination language approach for scenario control [23].

Model programs have a long history in the context of model-based testing with Spec Explorer [24, 25, 26], where composition was supported in a limited form through *scenario actions* that came to exist due to a popular demand. The new version of Spec Explorer is owned by the Windows organization, and is used for model-based testing of protocols [1]. Model programs are similar to action machines [27], the main difference is how composition is handled, composition of action machines is based on inference rules and symbolic computation that incorporates the notion of computable approximations of subsumption checking between symbolic states, using three-valued logic. Model programs and their compositionality properties also are related to parameterized extended finite state machines [28], symbolic transition systems [29], and attributed automata [30]. Some of those relationships are discussed in [18]. More about model-based testing applications and further motivation for the composition of model programs can be found in [31, 32, 25].

Model programs are similar to Event-B models [7]. Much like model programs being an extension of ASMs with actions, Event-B is an extension of the B-method [33] with events (corresponding to actions in model programs) that describe atomic behaviors. Each event is associated with a guard and an assignment, that causes a state transition when the guard is true is a given state. Unlike Event-B that is a whole specification language, model programs are language agnostic, in particular there is a mapping from AsmL as well as C# into model programs, and models from different modeling languages may be mixed, e.g. by composing textually represented FSM model programs with C# model programs in NModel [8]. When translating from AsmL or ASMs that use *choose*-statements, the resulting model programs are nondeterministic (use choice variables), in Event-B there is a similar *any*-statement for expressing nondeterminism. It is unclear as to what extent Event-B supports *forall*-statements that are commonly used in AsmL and are in model programs translated into comprehensions in the background $\mathcal{T}$. Choose-statements may also be nested within forall-statements in ASMs which translate into non-basic choice functions or Skolem functions in model programs [5]. Composition of Event-B models was introduced in [34] and is further discussed in [35]. Event composition and event fusion, introduced in [34], are similar to composition of model programs, where event composition assumes disjoint state variables, whereas event fusion allows shared state variables. It is unclear from the presentation in [34], how events can be parameterized, and how the parameter values are combined during composition. Note that in the context of model programs, the action parameters are unified through the action parameter variables that are essentially shared read-only variables in the composed model program. The primary motivation for composition of model programs has been testing and scenario control, whereas in Event-B the primary motivation for composition is to support feature oriented system refinement during modeling. B-models can be analyzed with the ProB tool [36], in form of model animation and model checking. In particular, ProB has built-in LTL model checking support. In NModel, LTL is not directly supported, the user would need to write an explicit automaton corresponding to the LTL formula and compose it with the contract model. In addition to

B, in the most recent version of ProB, there is also support for CSP-M and Z, see [36]. A useful extension to ProB would be to support model programs in a way that would allow for example ASM-style models to be composed with B-style models, that would also help to close the gap between the two communities. Recent work in the context of model programs has investigated various symbolic methods for analysis, that depend on the formalization of model programs with respect to the background theory $\mathcal{T}$, such as symbolic model checking [12, 13], symbolic conformance checking [37] and symbolic ioco or alternating refinement checking [38]. In these contexts, composition of model programs can be used for scenario oriented analysis, as illustrated in Section 7. Due to the close relation between model programs and Event-B models, some of the complexity results and techniques might also be relevant to the B-community as well as other modeling formalisms that rely on sets and maps and where refinement relations are used, such as RAISE, Z, TLA+, see [39].

In general, model programs over $\mathcal{T}$ are a non-trivial extension of explicit LTSs. They also differ from formalisms that use symbolic or standard programmatic descriptions of LTSs through extended finite state machines, by supporting unbounded comprehensions. In particular, the bounded model checking problem is highly undecidable, Σ_1^1-complete, for general model programs [5] (even for a single step) whereas the problem is decidable for standard sequential programs.

The main application of model programs has been for analysis and testing of software systems. In particular, for passive testing or runtime monitoring, a model program can be used as an oracle that observes the traces of a system under test and reports a failure when an action occurs that is not enabled in the model. In the context of testing of reactive systems with model programs [32], the action symbols are separated into controllable and observable ones, e.g., in Example 1, *Req* could be controllable and *Res* observable. In that context the semantics of a model program as an input-output LTS [3, 40] is fundamental in order to use ioco [41, 42], or alternating refinement [43, 44], as a foundation of the conformance relation.

Symbolic analysis of model programs is recent and ongoing work. The use of composition as a way to reduce symmetries in proof search is first noted in [12] in the context of symbolic reachability checking of model programs. The symbolic bounded reachability problem of model programs is studied in [5, 12, 13]. The problem is a generalization of symbolic bounded model checking [45, 46] to model programs. Theorem proving modulo $\mathcal{T}$ can also be used to check whether the traces of one model program form a subset of the traces of another model program [37]. We use the state of the art SMT solver Z3 [14] for our experiments on satisfiability problems in $\mathcal{T}$. The reduction to Z3 takes advantage of built-in support for *Ite* terms, sets, tuples, and an extensional theory of arrays that supports a direct encoding of maps.

Acknowledgments

We thank the anonymous referees whose comments considerably helped to improve this paper.

References

[1] W. Grieskamp, D. MacDonald, N. Kicillof, A. Nandan, K. Stobie, F. Wurden., Model-based quality assurance of Windows protocol documentation, in: First International Conference on Software Testing, Verification and Validation, ICST, Lillehammer, Norway, 2008.

[2] J. Ernits, R. Roo, J. Jacky, M. Veanes, Model-based testing of web applications using NModel, in: TESTCOM/FATES 2009, LNCS, Springer, 2009.

[3] R. Keller, Formal verification of parallel programs, Communications of the ACM (1976) 371–384.

[4] Y. Gurevich, Specification and Validation Methods, Oxford University Press, 1995, Ch. Evolving Algebras 1993: Lipari Guide, pp. 9–36.

[5] M. Veanes, N. Bjørner, Y. Gurevich, W. Schulte, Symbolic bounded model checking of abstract state machines, Int J Software Informatics 3 (2–3) (2009) 1–22.

[6] Y. Gurevich, N. Tillmann, Partial updates, Theoretical Computer Science 336 (2–3) (2005) 311–342.

[7] J.-R. Abrial, S. Hallerstede, Refinement, decomposition and instantiation of discrete models: Application to Event-B, Fundamenta Informaticae 77 (1-2) (2007) 1–28.

[8] J. Jacky, M. Veanes, C. Campbell, W. Schulte, Model-based Software Testing and Analysis with C#, Cambridge University Press, 2008.

[9] M. Resnick, Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds, MIT Press, 1997.

[10] NModel, http://www.codeplex.com/NModel, public version released May 2008.

[11] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, MIT Press, 1999.

[12] M. Veanes, N. Bjørner, A. Raschke, An SMT approach to bounded reachability analysis of model programs, in: K. Suzuki, T. Higashino, K. Yasumoto, K. El-Kakih (Eds.), FORTE 2008, Vol. 5048 of LNCS, Springer, 2008, pp. 53–68.

[13] M. Veanes, A. Saabas, On bounded reachability of programs with set comprehensions, in: LPAR'08, Vol. 5330 of LNAI, Springer, 2008, pp. 305–317.

[14] L. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08), Vol. 4963 of LNCS, Springer, 2008, pp. 337–340.

[15] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem, Vol. 1032 of LNCS, Springer, 1996.

[16] S. Katz, D. Peled, Defining conditional independence using collapses, Theoretical Computer Science 101 (1992) 337–359.

[17] C. Flanagan, P. Godefroid, Dynamic partial-order reduction for model checking software, in: J. Palsberg, M. Abadi (Eds.), POPL 2005, ACM, 2005, pp. 110–121.

[18] M. Veanes, C. Campbell, W. Schulte, Composition of model programs, in: J. Derrick, J. Vain (Eds.), FORTE 2007, Vol. 4574 of LNCS, Springer, 2007, pp. 128–142.

[19] M. Veanes, W. Schulte, Protocol modeling with model program composition, in: K. Suzuki, T. Higashino, K. Yasumoto, K. El-Kakih (Eds.), FORTE 2008, Vol. 5048 of LNCS, Springer, 2008, pp. 324–339.

[20] M. Veanes, J. Ernits, C. Campbell, State isomorphism in model programs with abstract data structures, in: J. Derrick, J. Vain (Eds.), FORTE 2007, Vol. 4574 of LNCS, Springer, 2007, pp. 112–127.

[21] J. Ernits, Two state space reduction techniques for explicit state model checking, Ph.D. thesis, Tallinn University of Technology (2007).

[22] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 1979.

[23] W. Grieskamp, N. Kicillof, A schema language for coordinating construction and composition of partial behavior descriptions, in: SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, Shanghai, China, May 27, 2006, ACM, 2006, pp. 59–66.

[24] W. Grieskamp, Y. Gurevich, W. Schulte, M. Veanes, Generating finite state machines from abstract state machines, SIGSOFT Softw. Eng. Notes 27 (4) (2002) 112–122.

[25] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, L. Nachmanson, Model-based testing of object-oriented reactive systems with Spec Explorer, in: R. Hierons, J. Bowen, M. Harman (Eds.), Formal Methods and Testing, Vol. 4949 of LNCS, Springer, 2008, pp. 39–76.

[26] M. Utting, B. Legeard, Practical Model-Based Testing - A tools approach, Elsevier Science, 2006.

[27] W. Grieskamp, N. Kicillof, N. Tillmann, Action machines: a framework for encoding and composing partial behaviors, IJSEKE 16 (5) (2006) 705–726.

[28] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines – a survey, Proceedings of the IEEE 84 (8) (1996) 1090–1123.

[29] L. Frantzen, J. Tretmans, T. Willemse, A symbolic framework for model-based testing, in: K. Havelund, M. Núñez, G. Rosu, B. Wolff (Eds.), FATES/RV 2006, no. 4262 in LNCS, Springer, 2006, pp. 40–54.

[30] M. Meriste, J. Penjam, Attributed models of executable specifications, in: M. V. Hermenegildo, S. D. Swierstra (Eds.), PLILP'95, Vol. 982 of LNCS, Springer, 1995, pp. 459–460, full version available as Research Report CS80/95, Department of Computer Science, Institute of Cybernetics, Tallinn 1995.

[31] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, M. Veanes, Testing concurrent object-oriented systems with Spec Explorer, in: J. Fitzgerald, I. J. Hayes, A. Tarlecki (Eds.), FM 2005: Formal Methods, Vol. 3582 of LNCS, Springer, 2005, pp. 542–547.

[32] M. Veanes, C. Campbell, W. Schulte, N. Tillmann, Online testing with model programs, in: ESEC/FSE-13, ACM, 2005, pp. 273–282.

[33] J.-R. Abrial, The B-Book: Assigning programs to meanings, Cambridge University Press, 1996.

[34] M. Poppleton, The composition of Event-B models, in: E. Börger, M. J. Butler, J. P. Bowen, P. Boca (Eds.), Int. Conference on ASM, B and Z (ABZ'08), Vol. 5238 of LNCS, Springer, 2008, pp. 209–222.

[35] M. Butler, Decomposition structures for Event-B, in: M. Leuschel, H. Wehrheim (Eds.), Integrated Formal Methods (IFM'09), Vol. 5423 of LNCS, Springer, 2009, pp. 20–38.

[36] ProB, http://www.stups.uni-duesseldorf.de/prob/.

[37] M. Veanes, N. Bjørner, Symbolic bounded conformance checking of model programs, in: A. Pnueli, I. Virbitskaite, A. Voronkov (Eds.), Perspectives of System Informatics (PSI'09), LNCS, Springer, 2009.

[38] M. Veanes, N. Bjørner, Input-output model programs, in: M. Leucker, C. Morgan (Eds.), ICTAC 2009, Vol. 5684 of LNCS, Springer, 2009, pp. 322–335.

[39] D. Bjørner, M. Henson (Eds.), Logics of Specification Languages, Springer, 2008.

[40] N. Lynch, M. Tuttle, Hierarchical correctness proofs for distributed algorithms, in: 6th annual ACM Symposium on Principles of distributed computing, ACM, 1987, pp. 137–151.

[41] E. Brinksma, J. Tretmans, Testing Transition Systems: An Annotated Bibliography, in: Summer School MOVEP'2k, Vol. 2067 of LNCS, Springer, 2001, pp. 187–193.

[42] J. Tretmans, Model based testing with labelled transition systems, in: R. Hierons, J. Bowen, M. Harman (Eds.), Formal Methods and Testing, Vol. 4949 of LNCS, Springer, 2008, pp. 1–38.

[43] R. Alur, T. A. Henzinger, O. Kupferman, M. Vardi, Alternating refinement relations, in: D. Sangiorgi, R. de Simone (Eds.), CONCUR 1998, Vol. 1466 of LNCS, Springer, 1998, pp. 163–178.

[44] L. de Alfaro, Game models for open systems, in: Verification: Theory and Practice, Vol. 2772 of LNCS, Springer, 2004, pp. 269 – 289.

[45] A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: R. Cleaveland (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'99), Vol. 1579 of LNCS, Springer, 1999, pp. 193–207.

[46] L. de Moura, H. Rueß, M. Sorea, Lazy theorem proving for bounded model checking over infinite domains, in: A. Voronkov (Ed.), CADE 2002, Vol. 2392 of LNCS, Springer, 2002, pp. 438–455.