Composition of Model Programs

Margus Veanes[1], Colin Campbell[2⋆], and Wolfram Schulte[1]

[1] Microsoft Research, Redmond, WA
{margus,schulte}@microsoft.com
[2] Modeled Computation LLC, Seattle, WA
colin@modeled-computation.com

Abstract. Model programs are a useful formalism for software testing and design analysis. They are used in industrial tools, such as SpecExplorer, as a compact, expressive and precise way to specify complex behavior. One of the challenges with model programs has been the difficulty to separate contract modeling from scenario modeling. It has not been clear how to separate those concerns in a clean way. In this paper we introduce composition of model programs, motivate why it is useful to be able to compose model programs, and what composition of model programs formally means.

1 Introduction

Model programs are a useful formalism for software testing and design analysis. They are used in industrial tools like SpecExplorer [1] as a compact, expressive and precise way to specify complex behavior. Model programs are unwound into transition systems that can be used in model-based testing, for runtime conformance checking of a system under test, and for design validation [4, 15–17].

In practice we have observed two distinct uses of model programs. The first use is as a *software contract* that encodes the expected behavior of the system under test. Here, the model program acts as an oracle that predicts system behavior in each possible context. The unwinding of such a contract model is typically infinite, since for many systems, such as those that allocate new objects at runtime, there are infinitely many possible states.

The second use is to define the *scenarios* to be tested or analyzed. Here, the purpose of the model program is to produce (when unwound) states and transitions of interest for a particular test or type of analysis. For example, one might want to direct a test to consider only certain interleavings of actions instead of all possible interleavings. Another example would be a model that specifies a finite set of input data to be used as system inputs.

Current practice tends to combine these two roles within a single model program, even though it is recognized that cleanly separating these concerns would be much better engineering practice. In addition, we have observed that as contract models grow, it would be helpful if they could be divided into submodels of manageable size. Up to now we have lacked the formal machinery to accomplish this.

⋆ The work in this paper was done at Microsoft Research.

At issue is the separation of design aspects into distinct but related model programs. If model programs are related exclusively by common action labels, then the desired system behavior is the intersection of possible traces for each aspect. In classical automata theory, the technique of achieving intersection of traces is product composition. We extend this technique here to define *parallel composition* of model programs.

Not all composition is parallel; sometimes it is useful to think in terms of phases of system operation. A typical example occurs when there is an initialization phase, followed by an operational phase with many possible behaviors, followed by a shutdown phase. We define the *serial composition* of model programs, which is analogous to serial composition of finite automata for language concatenation.

The main contribution of the paper is the formalization of the parallel composition of model programs in a way that builds on the classical theory of LTSs [12]. 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.

Achieving this goal required us to "rethink" the way actions are treated. Spec Explorer uses a mixture of a Mealy view and an LTS view that causes a complication in the definition of conformance. In this paper we adopt a consistent LTS-based view of action traces. This enables a direct application of the formal LTS based teory of testing using ioco [3] when the direction (input or output) of actions is specified. A key aspect of the composition of model programs is that actions are represented by terms that may include variables and values, and the notion of an action vocabulary is defined using only the function symbol part of the action. When actions are synchonized, values are shared through *unification* and may transfer data from one model program to another.

Model program composition is the cornerstone of the *NModel* framework that provides a modeling library for model programs written in C#. NModel is in the process of becoming an open source project and is the software support for the forthcoming textbook [11] that discusses the use of model programs as a practical modeling technique. While this paper provides the foundations of model program composition, the textbook shows practical techniques and applications, with an emphasis on composition as a method of layering system behavior into independent features.

The techniques for parallel and serial composition of model programs, as we will see below, have characteristics that make them appealing for use in the domain of software testing and design validation. We begin with an example. Then in sections 3 and 4 we give a formalization.

1.1 Example

Consider three model programs M_1, M_2 and M_3 that specify, respectively, a GUI-based application, a dialog box used in that application and a test scenario. The state spaces of the model programs are disjoint but their action signatures have nonempty intersections.

In the presentation that follows we unwind control state but not data state to produce control graphs in the spirit of Extended Finite State Machines (EFSMs) [13]. Figures 1-3 show M_1, M_2 and M_3 using this view.

The model program of the GUI-based application is shown in Figure 1. It has three control states, p_1, p_2 and p_3. Control state p_1 is both the initial state (indicated by the incoming arrow) and an accepting state (indicated by the double circle). The arcs between control states are labeled by guarded update rules called actions. These actions contain enabling conditions (prefixed by requires) and updates in curly braces. The actions include parameters which are substituted by ground values during unwinding.

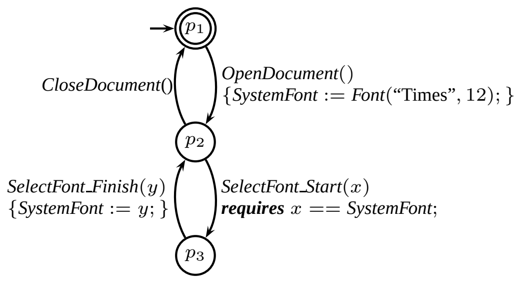

p_1
CloseDocument()
OpenDocument$()$
$\{$*SystemFont* $:=$ *Font*(“Times”, $12); \}$
p_2
SelectFont_Finish(y)
$\{$*SystemFont* $:= y; \}$
SelectFont_Start(x)
requires $x ==$ *SystemFont*;
p_3

Fig. 1. Application model M_1.

The data state of M_1 contains one state variable, *SystemFont*.

Runs of a model program begin in the initial control state and end in an accepting control state. Every step of the run must satisfy the enabling condition of the action that produced it.

Note that this model program uses an *LTS view* instead of a *Mealy view* for the action that sets the system font. In an LTS view, inputs and outputs appear as separate transitions, possibly breaking a single logical action into two parts. *SelectFont_Start* takes an input, namely the current system font given by the data state variable *SystemFont*. The parameter of *SelectFont_Finish* denotes the output. Since the *SelectFont_Finish* action has no enabling condition, any font value could be selected.

Model program M_2 that describes a font-choosing dialog box is shown in Figure 2.

The action signature of M_2 consists of *SelectFont_Start*, *SelectFont_Finish*, *OK*, *Cancel*, *SetFontName* and *SetFontSize*. Notice that this vocabulary has two actions in common with M_1, the application model, as well as four actions that are not shared.

Once started, the dialog box allows the user to set the font size and the font name in any order and as many times as desired. Depending on whether the user presses *OK* or *Cancel* either the newly selected font or the prior font is included in the exit label.

Model program M_3 gives a scenario of interest for testing. It is shown as Figure 3.

The scenario model shows two use cases for the font dialog. There are only two possible traces for this machine.

As is typical with scenario models, M_3 contains no updates to data state. We also use *SetFontSize*(10) as a shorthand for *SetFontSize*(x) ***requires*** $x == 10$. We use the

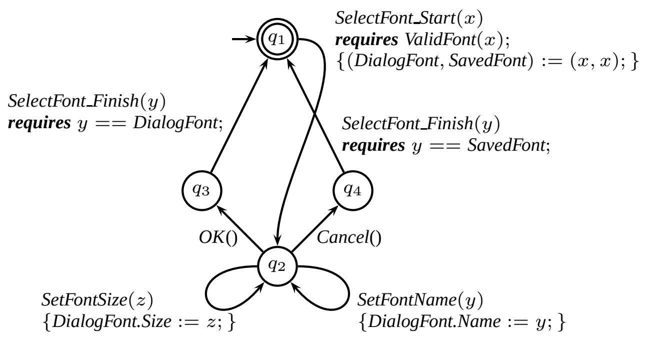

$SelectFont\_Start(x)$
requires $ValidFont(x)$;
$\{(DialogFont, SavedFont) := (x, x);\}$
$SelectFont\_Finish(y)$
requires $y == DialogFont$;
$SelectFont\_Finish(y)$
requires $y == SavedFont$;
q_1
q_2
q_3
q_4
$OK()$
$Cancel()$
$SetFontSize(z)$
$\{DialogFont.Size := z;\}$
$SetFontName(y)$
$\{DialogFont.Name := y;\}$

Fig. 2. Font chooser dialog model M_2.

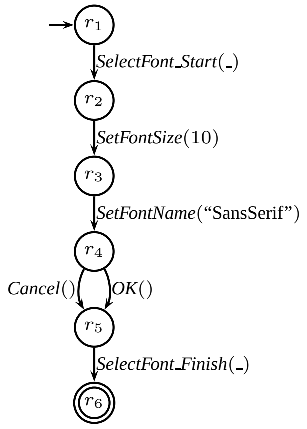


Fig. 3. Scenario model M_3 showing two ways to use the font dialog.

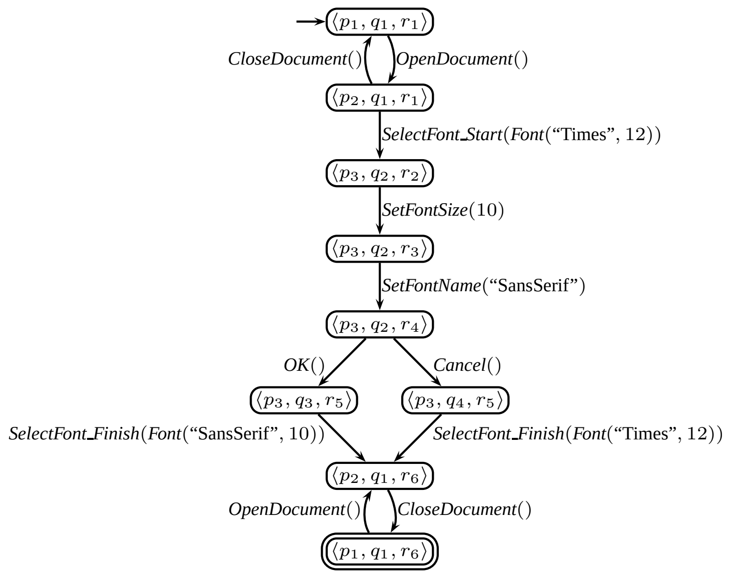
underscore symbol (“_”) to indicate an unconstrained parameter that is not used in any precondition or update.

Figure 4 shows the parallel composition of M_1, M_2 and M_3. The diagram omits the state update rules for brevity.

Under parallel composition, model programs will synchronize steps for shared actions and interleave actions not found in their common signature. The control states of the composed model program are a subset of the cross product of the control states of the component models.

The enabling conditions of the transitions are the conjunction of the enabling conditions of the component models. The data updates are the union of the data updates of the component programs. There can be no conflicting updates because the data signatures must be disjoint.

An accepting state under parallel composition occurs when all of component control states are accepting states. This accounts for the fact that the font may only be selected



$\langle p_1, q_1, r_1\rangle$
CloseDocument()
OpenDocument()
$\langle p_2, q_1, r_1\rangle$
SelectFont_Start(*Font*("Times", 12))
$\langle p_3, q_2, r_2\rangle$
SetFontSize(10)
$\langle p_3, q_2, r_3\rangle$
SetFontName("SansSerif")
$\langle p_3, q_2, r_4\rangle$
OK()
Cancel()
$\langle p_3, q_3, r_5\rangle$
$\langle p_3, q_4, r_5\rangle$
SelectFont_Finish(*Font*("SansSerif", 10))
SelectFont_Finish(*Font*("Times", 12))
$\langle p_2, q_1, r_6\rangle$
OpenDocument()
CloseDocument()
$\langle p_1, q_1, r_6\rangle$

Fig. 4. Parallel composition M_4 of the application model M_1, the font chooser dialog model M_2, and the scenario model M_3. Update rules associated with labels are not shown.

exactly one time in the composed model program– the scenario model M_3 does not loop, and its initial state is not an accepting state.

2 Basic definitions

Let Σ be a fixed signature of function symbols. Some function symbols in Σ, denoted by Σ^{dynamic}, may change their interpretation and are called *state variables*. The remaining set of symbols, denoted by Σ^{static}, have a fixed interpretation with respect to a given *background theory* $\mathcal{B}$. $\mathcal{B}$ is identified with its models that are called *states*. It is assumed that all states share the same universe $\mathcal{V}$ of values. Without loss of generality one may identify a state with a particular interpretation (value assignment) to all the state variables. Note that *logic variables* are distinct from state variables. Logic variables are needed below to be able to construct nonground action terms.

Example 1. Consider the application model M_1 in Figure 1. *SystemFont* is a nullary state variable here. $\mathcal{V}$ is fixed and includes at least strings, integers, and fonts. A font can be constructed using the static binary function *Font*. M_1 has a single nullary state variable *SystemFont*.

Terms are defined inductively over Σ and a set of logic variables disjoint from Σ. An *equation* is an atomic formula $t_1 == t_2$ where t_1 and t_2 are terms and '$==$' is the formal equality symbol. Formulas are built up inductively from atomic formulas using logical connectives and quantifiers.[3] A term or a formula e may contain free logic variables *FV*(e); e is *ground* or *closed* if *FV*(e) is empty. A *substitution* is a finite (possibly empty) map from logic variables to terms. Given a substitution θ and an expression e, $e\theta$ denotes the replacement of x in e by $\theta(x)$ for each x in *FV*(e). We say that θ is *grounding for* e if $e\theta$ is ground. Given a closed formula φ and a state S, $S \models \varphi$ is used to denote that S *satisfies* φ, or φ *holds* or *is true* in S.[4] A closed formula is *consistent* if it is true in some state. We write t^S for the interpretation of a ground term t in S. When an n-ary function symbol f is *self-interpreting* or a *free constructor* it means that $f(t_1, \ldots, t_n)^S = g(u_1, \ldots, u_m)^S$ if and only if f and g are the same function symbol (and thus $n = m$) and $t_i^S = u_i^S$ for all i.

Example 2. Consider the signature of M_1 again and let $t = \mathit{Font}(x, y)$; t is a term with $\mathit{FV}(t) = \{x, y\}$. The substitution $\theta = \{x \mapsto \text{“Times”}, y \mapsto 10\}$ is grounding for t and $t\theta$ is the ground term $\mathit{Font}(\text{“Times”}, 10)$ denoting the corresponding font, where *Font* is a free constructor. Let S be a state where the value of *SystemFont* is the Times font of size 12. Then $S \models \neg \mathit{SystemFont} == \mathit{Font}(\text{“Times”}, 10)$ because *Font* is self-interpreting and $10 \neq 12$.

A *location* is a pair $\langle f, (v_1, \ldots, v_n)\rangle$ where f is an n-ary function symbol in Σ^{dynamic} and $(v_1, \ldots, v_n)$ is a sequence of values. An *update* is an ordered pair denoted by $l \mapsto v$, where l is a location and v a value. A set U of updates is *consistent* if there are no two distinct updates $l \mapsto v_1$ and $l \mapsto v_2$ in U. Given a state S and a consistent set U of updates, $S \uplus U$ is the state where, for all $f \in \Sigma^{\text{dynamic}}$ of arity $n \geq 0$ and values $v_1, \ldots, v_n$,

$$f^{S \uplus U}(v_1, \ldots, v_n) = \begin{cases} w, & \text{if } \langle f, (v_1, \ldots, v_n)\rangle \mapsto w \in U; \\ f^S(v_1, \ldots, v_n), & \text{otherwise.} \end{cases}$$

In other words, $S \uplus U$ is the state after applying the updates U to S.

For the purposes of this paper it is enough to assume that all state variables are nullary, in which case the notions of locations and state variables can be unified.

A *program* P over Σ when applied to (or executed in) a state S, produces a set of updates. Often P also depends on formal parameters $\mathit{FV}(P) = x_1, \ldots, x_n$ for some $n \geq 0$. Thus, P denotes a function $[\![P]\!] : \mathit{State} \times \mathcal{V}^n \to \mathit{UpdateSet}$. It is convenient to extend the notion of expressions to include programs so that we can talk about free variables in programs and apply substitutions to them. Given a grounding substitution θ for P and a data state S, we write $[\![P\theta]\!](S)$ or $[\![P]\!](S, \theta)$ for $[\![P]\!](S, x_1\theta^S, \ldots, x_n\theta^S)$.

Example 3. Returning to M_1 in Figure 1, we have that the transition from p_3 to p_2 is associated with the assignment (i.e. a basic program) $\mathit{SystemFont} := y$, say P, with a single formal parameter y. Given a substitution $\theta = \{y \mapsto t\}$ where t is ground, and any state S, $[\![P\theta]\!](S) = \{\mathit{SystemFont} \mapsto t^S\}$.

[3] In general we may also have relation symbols, or Boolean functions, in Σ and form atomic formulas other than equations.

[4] We have in mind standard Tarski semantics for first order logic.

We also use the notion of a *labeled transition system* or *LTS* $(\mathcal{S}, \mathcal{S}_0, \mathcal{L}, \mathcal{T})$ that has a nonempty set $\mathcal{S}$ of *states*, a nonempty subset $\mathcal{S}_0 \subseteq \mathcal{S}$ of *initial states*, a nonempty set $\mathcal{L}$ of *labels* and a transition relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$. Here states and labels are abstract elements but in our use of LTSs the notion of LTS states and first-order states as introduced above will coincide. A *run* is a transition sequence $(S_i, L_i, S_{i+1})_{i<k}$, of some (possibly infinite) length k, and if $k > 0$ then $S_0 \in \mathcal{S}_0$; if k is finite and nonzero then S_k is called the *end-state* of the run. An S*-run* for a given initial state S is a nonempty run as above where $S_0 = S$. An S*-trace* of an S-run as above is the label sequence $(L_i)_{i<k}$ of length k. Intuitively, a trace is the sequence of labels of a run; the states are not part of a trace. A *finite* run or trace has finite length.

3 Model programs

A *guarded program* (over Σ) is a pair $[\varphi]/P$ where φ is a formula and P is a program. Let G be a guarded program $[\varphi]/P$. Intuitively, G denotes the restriction of $[\![P]\!]$ to those states and input parameters where φ holds. Let $\mathit{FV}(G) \stackrel{\text{def}}{=} \mathit{FV}(\varphi) \cup \mathit{FV}(P)$.

Definition 1. Σ^{action} denotes a fixed subset of the free constructors of Σ^{static} called *action symbols*. An *action term* is a term $f(t_1, \ldots, t_n)$ where f is an n-ary action symbol for some $n \geq 0$, and each t_i is either a distinct logic variable or a ground term over $\Sigma^{\text{static}} - \Sigma^{\text{action}}$. Given $\Gamma \subseteq \Sigma^{\text{action}}$ we write $\mathcal{A}(\Gamma)$ for the set of all action terms with action symbols in Γ. By an *action* we mean the interpretation of a ground action term.

Notice that the interpretation of a ground action term is the same in all data states. Notice also that there is essentially no difference between a nullary action symbol and the corresponding action (term).

Example 4. Consider M_1 in Figure 1. There are two nullary action symbols *CloseDocument* and *OpenDocument*, and two unary action symbols *SelectFont_Start* and *SelectFont_Finish*. *Font* is a free constructor, it is not an action symbol. The terms *SelectFont_Start*(*Font*("Times", 10)) and *SelectFont_Start*(x) are action terms; the terms *SelectFont_Start*(*SystemFont*) and *SelectFont_Start*(*Font*("Times", y)) on the other hand are *not* action terms, because in the former *SystemFont* is not in Σ^{static} and in the latter the action parameter *Font*("Times", y) is not a logic variable and not a ground term.

Definition 2. A *model program with explicit control graph* M has the following components.

1. A *signature* Σ.
2. An *action signature* $\Gamma \subseteq \Sigma^{\text{action}}$.
3. A finite nonempty set Q of *control points*.
4. An *initial control point* $q^{\text{init}} \in Q$.
5. A set of *accepting control points* $Q^{\text{acc}} \subseteq Q$.
6. A finite *control graph* $\delta \subseteq Q \times \mathcal{A}(\Gamma) \times Q$. The elements of δ are called *control transitions*.
7. A family $R = \{r_\rho\}_{\rho\in\delta}$ of *guarded programs*, where, for all $\rho = (q, a, p) \in \delta$, $\mathit{FV}(r_\rho) \subseteq \mathit{FV}(a)$; r_ρ is called *the guarded program for* ρ.

8. A closed formula φ^{entry} over Σ called an *entry condition*.

The guard of the guarded program for a control transition ρ is denoted by φ_ρ and the program is denoted by P_ρ. We denote M by the tuple $(\Sigma, \Gamma, Q, q^{\text{init}}, Q^{\text{acc}}, \delta, R, \varphi^{\text{entry}})$.

By a *model program* in this paper we mean a model program with explicit control graph.

A model program can be thought of as a control-flow graph whose edges are annotated by action terms and program segments similar to an EFSM [13].[5]

We use the special program *skip* that produces no updates.

Example 5. The model program M_1 in Figure 1 has the following components. The signature is described in Example 1. The action signature is described in Example 4. The control points are p_1, p_2 and p_3, where p_1 is both the initial control point and the only accepting control point. There are four control transitions in M_1. The guard of a control transition is indicated with the *requires* keyword or omitted if *true*. The program of a control transition is written within braces or omitted if *skip*. This is the Spec# [16] syntax of model programs.

A *state* of M as above is a pair $\langle S, q\rangle$ where S is a Σ-state and $q \in Q$. S is called the *data component* of S or a *data state*, whereas q is called the *control component* of S or a *control state*.[6] An *initial* state is a state whose control component is an initial control point and whose data component satisfies the entry condition. An *accepting* state is a state whose control component is an accepting control point.

Definition 3. The *labeled transition system underlying* M *LTS*(M) has the actions of M as its labels. The (initial) states of *LTS*(M) are the (initial) states for M. There is a transition $(\langle S, q\rangle, b, \langle S', q'\rangle)$ in *LTS*(M), if there is a control transition $\rho = (q, a, q')$ in M and a substitution θ such that:

- $b = a\theta^S$,
- $S \models \varphi_\rho\theta$,
- $[\![P_\rho\theta]\!](S)$ is consistent and $S' = S \uplus [\![P_\rho\theta]\!](S)$.

A transition of *LTS*(M) is called a *step* of M. Given a state S and an action a, we write $\delta(S, a)$ for the set of all states X such that (S, a, X) is a transition of *LTS*(M). Given a state S and a finite sequence $(a_i)_{i<k}$ of actions, we let

$$\hat{\delta}(S, (a_i)_{i<k}) = \bigcup\{\delta(X, a_{k-1}) : X \in \hat{\delta}(S, (a_i)_{i<k-1})\},$$
$$\hat{\delta}(S, ()) = \{S\}.$$

Thus, $\hat{\delta}(S, \alpha)$ is the set of all end-states of all S-runs whose trace is α. An action sequence α is an *accepting* S-trace if $\hat{\delta}(S, \alpha)$ contains an accepting state.

[5] In general, the control graph of a model program may itself be a control program and the set of generated control states may be infinite. We do not use this generalization in this paper.

[6] Formally, let *pc* be a fixed nullary function symbol not in Σ and let $\Sigma' = \Sigma \cup \{pc\}$. Then $\langle S, q\rangle$ stands for a Σ'-state where $pc^{\langle S,q\rangle} = q$ and $f^{\langle S,q\rangle} = f^S$ for all $f \in \Sigma$.

Definition 4. Let M be a model program with initial control state q_0. An S-*run* of M is an $\langle S, q_0\rangle$-run of *LTS*(M). An S-*trace* of M is an $\langle S, q_0\rangle$-trace of *LTS*(M). The set of all S-traces of M is denoted by *Traces*(S, M). An S-*trace* α of M is accepting if it is finite and $\hat{\delta}(\langle S, q_0\rangle, \alpha)$ contains an accepting state.

Example 6. The example shows how traces can depend on the data component of states. A possible accepting trace of M_1 from any initial state is:

OpenDocument,
SelectFont_Start(*Font*("Times", 12)),
SelectFont_Finish(*Font*("SansSerif", 10)),
SelectFont_Start(*Font*("SansSerif", 10)),
SelectFont_Finish(*Font*("Times", 10)),
CloseDocument

The argument to *SelectFont_Start* is the current system font recorded in the data state of M_1. When font selection finishes the new font is recorded in the state, i.e., in the action *SelectFont_Start*(*font*), the *font* argument acts like an input argument and in *SelectFont_Finish*(*font*) the *font* argument acts like an output argument of a font selection procedure.

4 Composition of model programs

The main operator underlying parallel composition of model programs is the product of two model programs. We will also use the following action signature extension operation over model programs.

Definition 5. Let M be a model program as above with action signature Γ. Let Γ' be a set of action symbols. We write $M^{+\Gamma'}$ for the model program whose action signature is extended with Γ' and $M^{+\Gamma'}$ has the following additional extensions for each action symbol $f \in \Gamma' - \Gamma$, let a_f denote a fixed action term $f(\_, \ldots, \_)$ where each occurrence of _ stands for a fresh logic variable,

- for all control states q, δ is extended with the control transition, (q, a_f, q),
- for each new control transition (q, a_f, q), $r_{(q,a_f,q)} = [\textit{true}]/\textit{skip}$.

The intuition is that for each new action symbol any corresponding action is enabled in every state and produces a self-loop in that state. This is also easily seen in the LTS semantics of $M^{+\Gamma'}$. This construct is used mainly to interleave actions that are not shared between two model programs being composed in a product. Notice that an action does not belong to a model program (or the underlying LTS) if its function symbol is not in the action signature of the model program.

Example 7. Consider M_1 in Figure 1 and let Γ_2 be the action signature of the font chooser dialog model M_2 in Figure 2. The only action symbols that M_1 and M_2 have in common are *SelectFont_Start* and *SelectFont_Finish*. Thus $M_1^{+\Gamma_2}$ has for example the new control transitions $(p_i, \textit{SetFontSize}(\_), p_i)$ for $1 \leq i \leq 3$ that are enabled in all states.

4.1 Product composition

We first define the product of two model programs that share the same signature and the same action signature. We then define parallel composition of model programs by using signature extension and product composition.

Due to the restricted form of action terms, two action terms a_1 and a_2 unify if and only if they have the same action symbol of some arity $n \geq 0$, and for all i, $1 \leq i \leq n$, the i'th argument of a_1 and the i'th argument of a_2 either denote the same value or at least one of them is a logic variable. If a_1 and a_2 unify there is trivially a most general unifier $\theta = \mathit{mgu}(a_1, a_2)$, i.e., any action that is both an instance of a_1 and an instance of a_2 is an instance of $a_1\theta$ (or $a_2\theta$).

We assume that logic variables used in two model programs are distinct so that we do not need to worry about variable renaming. Given two guarded programs $r_1 = [\varphi_1]/P_1$ and $r_2 = [\varphi_2]/P_2$ we write $r_1 \parallel r_2$ for the guarded program $[\varphi_1 \wedge \varphi_2]/P_1 \parallel P_2$, where the parallel composition $P_1 \parallel P_2$ produces the union of the updates of P_1 and P_2, i.e. $[\![P_1 \parallel P_2]\!](S, \theta) = [\![P_1]\!](S, \theta) \cup [\![P_2]\!](S, \theta)$.

Definition 6. Let $M_i = (\Sigma, \Gamma, Q_i, q_i^{\text{init}}, Q_i^{\text{acc}}, \delta_i, \{r_\rho^i\}_{\rho \in \delta_i}, \varphi_i^{\text{entry}})$, for $i = 1, 2$, be two model programs. The *product* of M_1 and M_2, denoted by $M_1 \times M_2$, is the model program

$$(\Sigma, \Gamma, Q_1 \times Q_2, \langle q_1^{\text{init}}, q_2^{\text{init}} \rangle, Q_1^{\text{acc}} \times Q_2^{\text{acc}}, \delta, \{r_\rho\}_{\rho \in \delta}), \varphi_1^{\text{entry}} \wedge \varphi_2^{\text{entry}}),$$

where δ and $\{r_\rho\}_{\rho \in \delta}$ are constructed as follows. For all $\rho_1 = (q_1, a_1, p_1) \in \delta_1$ and $\rho_2 = (q_2, a_2, p_2) \in \delta_2$ such that $\theta = \mathit{mgu}(a_1, a_2)$ exists,

- $\rho = (\langle q_1, q_2 \rangle, a_1\theta, \langle p_1, p_2 \rangle) \in \delta$, and
- $r_\rho = r_{\rho_1}\theta \parallel r_{\rho_2}\theta$.

If M_1 and M_2 are model programs with different action signatures Γ_1 and Γ_2 then $M_1 \times M_2 \stackrel{\text{def}}{=} M_1^{+\Gamma_2} \times M_2^{+\Gamma_1}$.

One can show that the product operator is commutative and associative as far as trace semantics of the final model program is concerned. This is made explicit in the following statement.

Proposition 1. *Let M_1, M_2 and M_3 be model programs with the same signature and action signature, and let S be a data state. Then $\mathit{Traces}(S, M_1 \times M_2) = \mathit{Traces}(S, M_2 \times M_1)$ and $\mathit{Traces}(S, M_1 \times (M_2 \times M_3)) = \mathit{Traces}(S, (M_1 \times M_2) \times M_3)$.*

Example 8. The model program M_4 in Figure 4 shows the product $M_1 \times M_2 \times M_3$. Let Γ_i denote the action signature of M_i. In this case $\Gamma_2 = \Gamma_3$ but Γ_1 has the additional actions for opening and closing a document, and does not include the action for changing the font name/size and the *OK* and *Cancel* actions. If we first construct the product $M_2 \times M_3$, we get a specialization M_{23} of the font chooser dialog model M_2 where we first set the font size to be 10 and then set the font name to be SansSerif. The product $M_1 \times M_{23}$, i.e. M_4, corresponds intuitively to a hierarchical refinement of M_1 with a particular use of the font dialog model as described by M_{23}. The actions that are specific to the font selection model are considered as self-loops in M_1, and conversely, closing and opening of a document are considered as self-loops in M_{23}. The final product M_4 is therefore $M_1^{+\Gamma_2} \times M_{23}^{+\Gamma_1}$. As an example of a guarded update program of M_4 consider the control transition

$$\rho = (\langle p_2, q_1, r_1\rangle, \mathit{SelectFont\_Start}(\mathit{Font}(\text{“Times”}, 12)), \langle p_3, q_2, r_2\rangle)$$

If we follow the definitions exactly and do not simplify the formulas and the programs then the guard associated with ρ is

$$\begin{aligned} &\textbf{\textit{requires}}\ \mathit{Font}(\text{“Times”}, 12) == \mathit{SystemFont} \\ &\qquad \wedge\ \mathit{true} \\ &\qquad \wedge\ \mathit{ValidFont}(\mathit{Font}(\text{“Times”}, 12)), \end{aligned}$$

and the program associated with ρ is

$$\mathit{skip} \parallel ((\mathit{DialogFont}, \mathit{SavedFont}) := (\mathit{Font}(\text{“Times”}, 12), \mathit{Font}(\text{“Times”}, 12)) \parallel \mathit{skip})\,.$$

4.2 Parallel composition

When the product composition is used in an unrestricted manner the end result is a new model program, which from the point of view of trace semantics might be unrelated to the original model programs. Essentially, this problem occurs if two model programs can read each others state variables.

Let $\mathit{SV}(e)$ denote the set of all *state variables* that occur in e, where e is either an expression, a program or a model program. Given a Σ_1-state S and a signature $\Sigma_2 \subseteq \Sigma_1$, we write $S{\restriction}\Sigma_2$ for the *reduct* of S to Σ_2. An ASM program is “honest” about its state dependencies in the sense that state variables that are not explicitly mentioned in the program do not influence its behavior and cannot be updated (e.g. there is no implicit stack and the programs cannot change the control state). Formally, we use the following fact:

Lemma 1. *Let S be a data state over Σ, let $\mathit{SV} \subseteq \Sigma^{\text{dynamic}}$, and let P be a program such that $\mathit{SV}(P) \subseteq \mathit{SV}$. Let $\Sigma' = \Sigma^{\text{static}} \cup \mathit{SV}$. Then $[\![P]\!](S) = [\![P]\!](S{\restriction}\Sigma')$.*

Definition 7. Let M_1 and M_2 be model programs with action signatures Γ_1 and Γ_2, respectively. M_1 and M_2 are *composable in parallel* if they have the same signature but disjoint state variables, in which case the *parallel composition* of M_1 and M_2, denoted by $M_1 \parallel M_2$, is defined as the product $M_1 \times M_2$.

The following theorem shows that parallel composition of model programs corresponds to parallel composition of the underlying LTSs. Such composition has the desired language-theoretic property that the traces produced by the composite model program are the intersection of the traces produced independently by the composed model programs.

Theorem 1. *Let M_1 and M_2 be model programs that are composable in parallel and have the same action signature. Then*

$$\mathit{Traces}(S, M_1 \parallel M_2) = \mathit{Traces}(S, M_1) \cap \mathit{Traces}(S, M_2).$$

Proof. Let $M_i = (\Sigma, \Gamma, Q_i, q_i^{\text{init}}, Q_i^{\text{acc}}, \delta_i, \{r_\rho^i\}_{\rho\in\delta_i}, \varphi_i^{\text{entry}})$, for $i = 1, 2$, be two model programs such that $\mathit{SV}(M_1) \cap \mathit{SV}(M_2) = \emptyset$. Let S be a data state. Let $M = M_1 \times M_2$. We only show that $\mathit{Traces}(S, M_1 \times M_2) \subseteq \mathit{Traces}(S, M_1) \cap \mathit{Traces}(S, M_2)$. The other direction is similar by using the same definitions in the opposite direction. Consider a trace $(a_i)_{i<k} \in \mathit{Traces}(S, M_1 \times M_2)$. There is a corresponding S-run

$$(\langle S_i, \langle q_i, p_i\rangle\rangle, a_i, \langle S_{i+1}, \langle q_{i+1}, p_{i+1}\rangle\rangle)_{i<k}$$

where $\langle q_0, p_0\rangle$ is the initial control state of the product model program and $S = S_0$. Fix an arbitrary step i in the run. The following holds by Definition 3: there is a control transition $\rho_i = (\langle q_i, p_i\rangle, t_i, \langle q_{i+1}, p_{i+1}\rangle)$ in M and a substitution θ such that

– $a_i = t_i\theta^{S_i}$,
– $S_i \models \varphi_{\rho_i}\theta$, and
– $[\![P_{\rho_i}\theta]\!](S_i)$ is consistent and $S_{i+1} = S_i \uplus [\![P_{\rho_i}\theta]\!](S_i)$.

By Defininition 6, there are control transitions $\rho_i^1 = (q_i, t_i^1, q_{i+1})$ in M_1 and $\rho_i^2 = (p_i, t_i^2, p_{i+1})$ in M_2 such that

– $\sigma = \mathit{mgu}(t_i^1, t_i^2)$ exists and $t_i = t_i^1\sigma$,
– $\varphi_{\rho_i} = \varphi_{\rho_i^1}\sigma \wedge \varphi_{\rho_i^2}\sigma$, and
– $P_{\rho_i} = P_{\rho_i^1}\sigma \parallel P_{\rho_i^2}\sigma$.

Let $\Sigma_1 = \Sigma - \mathit{SV}(M_2)$ and $\Sigma_2 = \Sigma - \mathit{SV}(M_1)$. Since $\mathit{SV}(M_1)$ and $\mathit{SV}(M_2)$ are disjoint and the guards in M_j may only contain state variables from $\mathit{SV}(M_j)$, it follows that $S_i{\restriction}\Sigma_1 \models \varphi_{\rho_i^1}\sigma\theta$ and $S_i{\restriction}\Sigma_2 \models \varphi_{\rho_i^2}\sigma\theta$. Also, since $[\![P_{\rho_i}\theta]\!](S_i) = U_1 \cup U_2$ is consistent, so are U_1 and U_2, where $U_1 = [\![P_{\rho_i^1}\sigma\theta]\!](S_i)$ and $U_2 = [\![P_{\rho_i^2}\sigma\theta]\!](S_i)$. By using Lemma 1 and the disjointness of $\mathit{SV}(M_1)$ and $\mathit{SV}(M_2)$ we know that $U_1 = [\![P_{\rho_i^1}\sigma\theta]\!](S_i{\restriction}\Sigma_1)$ and $U_2 = [\![P_{\rho_i^2}\sigma\theta]\!](S_i{\restriction}\Sigma_2)$. By using $S_{i+1} = S_i \uplus U_1 \cup U_2$, we get that $S_{i+1}{\restriction}\Sigma_1 = S_i{\restriction}\Sigma_1 \uplus U_1$ and $S_{i+1}{\restriction}\Sigma_2 = S_i{\restriction}\Sigma_2 \uplus U_2$.

Since i was chosen freely, we can construct the run

$$(\langle S_i{\restriction}\Sigma_1, q_i\rangle, a_i, \langle S_{i+1}{\restriction}\Sigma_1, q_{i+1}\rangle)_{i<k}$$

for M_1 and then expand all states in the run to Σ in such a way that the first state is S. We know also that $S \models \varphi_1^{\text{entry}}$ because $S \models \varphi_1^{\text{entry}} \wedge \varphi_2^{\text{entry}}$. It follows that $(a_i)_{i<k} \in \mathit{Traces}(S, M_1)$. Symmetrical argument applies to M_2. $\square$

Example 9. Consider M_1, M_2, M_3 from above. The state variables of each M_i are clearly disjoint; M_1 has the single state variable *SystemFont*, M_2 has the state variables *DialogFont* and *SavedFont*, and M_3 has no state variables. Thus M_4 is a parallel composition of $M_1^{+\Gamma_2}$, $M_2^{+\Gamma_1}$ and $M_3^{+\Gamma_1}$, where Γ_1 and Γ_2 are as in Example 8.

4.3 Serial composition

In scenario control it is often useful to compose two model programs serially (i.e. in a sequence). Intuitively, a serial composition of two model programs M_1 and M_2 means that the control flow may transition from an accepting control point of M_1 to the initial control point of M_2. Serial composition is therefore not well-defined for model programs that share control points. Note that, unlike the parallel case, state variable signatures need not be disjoint in serial composition.

Definition 8. Two model programs M_1 and M_2 are *serially composable* if they have the same action signature and disjoint sets of control points.

The formal definition of serial composition uses a new nullary action symbol τ for the transition from M_1 to M_2. The τ transition corresponds to an internal control transition from any accepting control point of M_1 to the initial control point of M_2 whose guard is the entry condition of M_2.

Definition 9. Let $M_i = (\Sigma, \Gamma, Q_i, q_i^{\text{init}}, Q_i^{\text{acc}}, \delta_i, \{r_\rho^i\}_{\rho\in\delta_i}, \varphi_i^{\text{entry}})$, for $i = 1, 2$, be two serially composable model programs and let τ be a fresh action symbol not in Γ. M_1 *followed by* M_2 using τ, denoted by $M_1;_\tau M_2$, is the model program

$$(\Sigma, \{\tau\} \cup \Gamma, Q_1 \cup Q_2, q_1^{\text{init}}, Q_2^{\text{acc}}, \underbrace{\delta_1 \cup \delta_2 \cup \{(q, \tau, q_2^{\text{init}}) : q \in Q_1^{\text{acc}}\}}_{\delta}, \{r_\rho\}_{\rho\in\delta}, \varphi_1^{\text{entry}}),$$

where $r_\rho = r_\rho^1$, if $\rho \in \delta_1$; $r_\rho = r_\rho^2$, if $\rho \in \delta_2$; $r_\rho = [\varphi_2^{\text{entry}}]/$*skip*, otherwise.

It is easy to see that an S-trace of $M_1;_\tau M_2$ has the form $\alpha\tau\beta$ where α is an accepting S-trace of M_1 and β is an S'-trace of M_2 for some $S' \in \hat{\delta}_{M_1}(S, \alpha)$. Elimination of τ can be done at the expense of introducing nondeterminism. For parallel composition of two model programs, τ-actions in each one are always considered as distinct actions and are interleaved. One could also introduce τ as a special action that is always interleaved in a parallel composition as is done for example in the definition of LTSs [14].

5 Conclusions and related work

There is a tradeoff between how much of the global state should be encoded as control state and how much should be encoded as data state. In pure abstract state machines, states are completely encoded as data states, and there is no separate notion of control state [2, 9]. Model programs defined in [16] adopt this view. While this view is more concise and sufficient for many purposes it forces one to encode the control state as data state, and this may not be natural from the point of view of control flow as understood in traditional programming. Not having the distinction between control and data state makes also the definition of certain forms of composition, such as serial composition, harder to formalize because data states are shared whereas control states are disjoint in serial composition.

The approach that we have taken is similar to extended finite state machines (EFSMs) where a finite part of the state is separated as control state. In general, the control part does not need to be finite in model programs, but may encorporate the local stack of a program. Model programs are similar to parameterized EFSMs [13], except that EFSMs are a generalization of Mealy machines, whereas model programs do not distinguish a priori between inputs and outputs and incorporate the notion of accepting states like classical automata. The distinction between inputs and outputs becomes relevant for defining conformance, but is not relevant for the composition operators discussed in this paper that are used for scenario control and for composing aspects of a system model.

An important change from our prior approach of using model programs as a mixed Mealy and LTS view, taken in SpecExplorer, is the introduction of intermediate control states between the input part and the output part of an action. In other words, the underlying semantics is given by an LTS. This separation is also used with FSM based approaches where it is sometimes more convenient to formulate composition using IOTSs [6]. One of the key reasons for us to separate the inputs from the outputs as separate actions, rather than using a Mealy view, was to be able to have a simple definition of conformance relation that allows output nondeterminism when dealing with reactive systems. This is important for using ioco [3] or refinement of interface automata [5] for formalizing the confomance relation.

Further differences from EFSMs are that accepting states in model programs are used for serial composition and for defining validity of traces, and labels are not abstract elements but structured terms that allow sharing of arbitrary data values through unification. The trace semantics of model programs is based on the unwinding of model programs as labeled transition systems [14] where states are considered to be abstract points.

The separation of control state from data state, while allowing communication with terms that can incorporate data values, is important in the model-based testing applications of model programs, e.g. for scenario control and visualization of model programs. The definitions of parallel and serial composition of model programs are related to similar operations on classical automata (see e.g. [10]). There is a large body of work using FSMs and variations of LTSs that use the classical parallel composition of automata where shared actions are synchronized and other actions are interleaved asynchronously. It is important therefore that the semantics of composed model programs is based on the same notion of composition.

Model programs are also related to symbolic transition systems that have an explicit notion of data and data-dependent control flow [7]. Model program composition as defined in this paper is independent of the mechanism of exploration used. Various approaches, including explicit state exploration as well as exploration with symbolic labels and states, may be applied. For example, *action machines* [8] rely on symbolic techniques. The main difference compared to composition of action machines is that composition of model programs is syntactic, whereas composition of action machines is defined in the style of natural semantics using inference rules and symbolic computation that incorporates the notion of computable approximations of subsumption checking between symbolic states. The computable approximations reflect the power of the underlying decision procedures that are being used.

More about model-based testing applications and further motivation for the composition of model programs can be found in [4, 8, 17, 16]. The most recent work related to model programs where composition is discussed from a practical perspective is the forthcoming textbook [11].

References

1. Spec Explorer. URL:http://research.microsoft.com/specexplorer, released January 2005.
2. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

3. E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 187–193. Springer, 2001.
4. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with Spec Explorer (extended abstract). In *FM 2005: Formal Methods*, volume 3582 of *LNCS*, pages 542–547. Springer, 2005.
5. L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.
6. K. El-Fakih, A. Petrenko, and N. Yevtushenko. Fsm test translation through context. In *Proceedings of the 18th IFIP International Conference on Testing of Communicating Systems (TestCom 2006)*, LNCS. Springer, 2006.
7. L. Frantzen, J. Tretmans, and T. Willemse. A symbolic framework for model-based testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV 2006*, number 4262 in LNCS, pages 40–54. Springer, 2006.
8. W. Grieskamp, N. Kicillof, and N. Tillmann. Action machines: a framework for encoding and composing partial behaviors. *International Journal on Software and Knowledge Engineering*, 16(5):705–726, 2006.
9. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
10. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
11. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2007. Submitted to publisher.
12. R. Keller. Formal verification of parallel programs. *Communications of the ACM*, pages 371–384, July 1976.
13. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.
14. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151. ACM Press, 1987.
15. J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.
16. M. Veanes, C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, and N. Tillmann. Model-based testing of object-oriented reactive systems with Spec Explorer, 2005. Tech. Rep. MSR-TR-2005-59, Microsoft Research. To appear as a book chapter in *Formal Methods and Testing*.
17. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 273–282. ACM, 2005.