Protocol Modeling with Model Program Composition

Margus Veanes and Wolfram Schulte

Microsoft Research, Redmond, WA, USA
{margus,schulte}@microsoft.com

Abstract. Designing and interoperability testing of distributed, application-level network protocols is complex. Windows, for example, supports currently more than 200 protocols, ranging from simple protocols for email exchange to complex ones for distributed file replication or real time communication. To fight this increasing complexity problem, we introduce a methodology and formal framework that uses model program composition to specify behavior of such protocols. A model program can be used to specify an increment of protocol functionality with a coherent purpose, which can be understood and analyzed separately. The overall behavior of a protocol can be defined by a composite model program, which defines how the individual parts interoperate.

1 Introduction

Protocols are abundant; we rely on the reliable sending and receiving of email, multimedia, and business data. But protocols, such as SMB [28], can be very complex and hard to get right. They require careful design to guarantee reliability and failure resilience; they require careful and efficient implementations, to not clog the system; and they require careful documentation and interoperability testing, so that different vendors understand the same protocol.

A protocol typically has many different facets. Each facet provides a partial view of the overall functionality of the protocol with a coherent purpose. An example of a facet is a set of rules that describes how message ids are allowed to be computed and communicated between a client and a server in a client-server protocol.

In this paper we provide a methodology and a formal framework for specifying protocol facets as separate model programs. A model program is a collection of guarded update rules indexed by actions. A model program of a single facet can be subject to liveness and safety analysis, which can be infeasible to perform for the whole protocol model. Instead, one can apply compositional reasoning in the following sence. If a model program satisfies one property and another model program satisfies another property, then the composition of those model programs satisfies both properties. Distilling facet model programs also fosters reuse, since facets, such as an algorithm for request cancellation in a particular client-server protocol, typically reappear in similar protocols.

Model programs of different facets of a protocol can be composed into a single model program. Composition of model programs is syntactic, but the underlying trace semantics is based on the classical theory of labeled transition systems (LTSs) [31, 32]. This enables a direct application of the formal LTS based theory of testing using

IOCO [9] or interface automata refinement [15]. The step semantics of model programs is based on the theory of abstract state machines (ASMs) [25] with a rich background universe [6]. This enables explicit state exploration techniques [21] and symbolic analysis techniques that support the needed background theories [36], as well as a range of other ASM technologies [8] to be applied to model programs.

A key property of the composition of model programs is that actions may include parameters as logic variables. When actions are synchronized, values are shared through unification from one model program to another, which is different from communication through actions by composition of input/output automata [33], where input actions in one model are synchronized with output actions in the other model. We provide tool support for analyzing safety and liveness properties for basic and composed model programs within the NModel framework [34]. We have integrated model program composition into a model-based test environment in NModel so that interoperability tests can be driven from those combined models. The NModel framework uses C# for writing model programs and is explained in detail in [30], which also discusses the use of model programs as a practical modeling technique.

To summarize, this paper makes the following contributions.

- We introduce a novel modeling technique for protocols using a decomposition of a protocol into different facets that are modeled separately and composed using model programs.
- We define formally the composition of model programs that simplifies and extends the definition of parallel composition of model programs in [38]. In particular, the composition admits sharing of state variables and can be used for state-dependent scenario control.
- We illustrate the use of this modeling technique and composition on an excerpt of an industrially relevant and non-trivial SMB2 protocol.

The remainder of the paper is organized as follows. Section 2 defines model programs and related notions needed in the sequel. Section 3 defines model program composition. Section 4 illustrates the application of the technique to a sample protocol. Section 5 explains some aspects of the implementation and experiments. Section 6 is about related work. We finish off the paper with a short conclusion.

2 Model programs

Model programs can be viewed as abstract state machines (ASMs) [25] indexed by actions. The main use of model programs is as high-level specifications in model-based testing tools such as Spec Explorer [1, 37] and NModel [34]. In Spec Explorer, one of the supported input languages is the abstract state machine language AsmL [2, 26]. AsmL is used in this paper as the concrete specification language for update rules that correspond to basic ASMs with a rich background [6] $\mathcal{T}$ including arithmetic, sets, maps, tuples, user defined data types, etc.

We let Σ denote the overall signature of function symbols. Part of the signature, denoted by Σ^{var}, contains function symbols whose interpretation may vary from state to state. The remaining part Σ^{static} contains symbols whose interpretation is fixed by

the background theory. A ground term over Σ^{static} is called a *value term*. Formally, the *interpretation* of a value term t is the same in all states and is denoted by $[\![t]\!]$. An example of a value term t, using AsmL syntax, is a *range expression* `{3..7}`; whose value $[\![t]\!]$ is the set of all integers from 3 to 7.

A subset of Σ^{static}, denoted by Σ^{action} are free constructors called *action symbols*. An *action* is a value term $f(t_1, \ldots, t_n)$ where f is an action symbol, also called an f-action. We also say *action* for $[\![f(t_1, \ldots, t_n)]\!] = f([\![t_1]\!], \ldots, [\![t_n]\!])$. For all action symbols f with arity $n \geq 0$, and all i, $1 \leq i \leq n$, there is a unique *parameter variable* denoted by $f.i$. We write Σ_f for $\{f.i\}_{1 \leq i \leq n}$. Note that if $n = 0$ then $\Sigma_f = \emptyset$.

Definition 1. A *model program* P is a tuple (V_P, A_P, I_P, R_P), where

- V_P is a finite subset of Σ^{var}, called the *state variables of* P;
- A_P is a finite subset of Σ^{action}, called the *action symbols of* P;
- I_P is a formula over $\Sigma_P = \Sigma^{\text{static}} \cup V_P$, called the *initial state condition of* P;
- R_P is a family $\{R_P^f\}_{f \in A_P}$ of *action rules* $R_P^f = (G_P^f, U_P^f)$, where
  - G_P^f is a formula over $\Sigma_P \cup \Sigma_f$ called the *guard* or *enabling condition of* R_P^f;
  - U_P^f is an update rule over $\Sigma_P \cup \Sigma_f$ called the *update rule of* R_P^f.

We often say *action* to also mean an action rule or an action symbol, if the intent is clear from the context.

Example 1 (Credits). The following model program is written in AsmL. It specifies how a client and a server need to use message ids, based on a sliding window protocol (see Section 4). 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,c)")]
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,c,_)")]
Res(m as Integer, c as Integer)
   require m in requests
   require requests(m) >= c
   require c >= 0
   window := window union {maxId + i | i in {1..c}}
   requests := RemoveAt(requests,m)
   maxId := maxId + c
```

Its three state variables are indicated with the keyword `var`. *Credits* has two actions `Req` and `Res`, indicated with the `[Action]` attribute on the corresponding method definition. The initial state condition is given by the initial assignment of values to the state variables. The argument of the `[Action]` attribute provides the arity of the action symbol and the mapping from the formal parameter names used in the method definition to the corresponding parameter variables for the action symbol.[1] Each occurrence of the

[1] If the mapping coincides with the method signature, this argument can be omitted.

placeholder '_' indicates that the corresponding parameter variable is not referenced. The `Req` action rule $R^{\texttt{Req}}_{Credits}$ has the following components. The guard $G^{\texttt{Req}}_{Credits}$ is the conjunction of all of the `require`-statements. The update rule $U^{\texttt{Req}}_{Credits}$ is defined by the body of the method. Note that the parallel update rule is the default in AsmL, thus both assignments in the `Req` action are executed in parallel as a single transaction, although in this case a sequential execution would yield the same updates. The `Res` action rule is analogous. To summarize,

$$
\begin{aligned}
V_{Credits} &= \{\texttt{window}, \texttt{maxId}, \texttt{requests}\},\\
A_{Credits} &= \{\texttt{Req}, \texttt{Res}\},\\
I_{Credits} &= (\texttt{window} = \{0\} \wedge \texttt{maxId} = 0 \wedge \texttt{requests} = \{\mapsto\}),\\
G^{\texttt{Req}}_{Credits} &= (\texttt{Req}.2 \in \texttt{window} \wedge \texttt{Req}.3 > 0),\\
U^{\texttt{Req}}_{Credits} &= (\texttt{requests} := \texttt{Add}(\texttt{requests}, \texttt{Req}.2, \texttt{Req}.3) \parallel\\
&\qquad \texttt{window} := \texttt{window} \setminus \{\texttt{Req}.2\}).
\end{aligned}
$$

We introduce a special class of model programs used here for scenario control. A *finite state model program* is a model program all of whose state variables have a finite range. There is a straightforward encoding of regular expressions over the alphabet of actions with placeholders to finite state model programs.[2] Given such a regular expression ρ we write *FSMP*(ρ) for the corresponding finite state model program.

Example 2 (FSMP(`Req(_,0,2)`*)). The following model program P is a finite state model program, since $V_P = \emptyset$. Intuitively, P describes the closure `Req(_,0,2)`*.

```
[Action("Req(_,m,c)")]
Req(m as Integer, c as Integer)
  require m = 0 and c = 2
  skip
```

Let P be a fixed model program. A *P-state* is a mapping of V_P to values.[3] Given a P-state S, an extension of S with the parameter assignment $\theta = \{x_i \mapsto v_i\}_{1 \le i \le n}$ is denoted by $(S;\theta)$. Given an extended P-state S, the *reduction* of S to V_P is denoted by $S \restriction V_P$. Given an action $a = f(t_1, \ldots, t_n)$, let θ_a denote the parameter assignment $\{f.i \mapsto [\![t_i]\!]\}_{1 \le i \le n}$.

Let S be a P-state, and let a be an f-action. We use the notion of *firing* of an update rule U in a state S [25], denoted here by *Fire*(S, U), that yields the updated state, provided that *Fire*(S, U) is defined (a consistent update set exists).[4] Then a is *enabled* in S if $(S;\theta_a) \models G^f_P$ and $S' =$ *Fire*$((S;\theta_a), U^f_P) \restriction V_P$ is defined. Then a *causes a transition from S to S'*.

A *labeled transition system* or *LTS* is a tuple $(\mathcal{S}, \mathcal{S}_0, L, T)$, where $\mathcal{S}$ is a set of *states*, $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of *initial states*, L is a set of labels and $T \subseteq \mathcal{S} \times L \times \mathcal{S}$ is a *transition relation*.

[2] Model programs also have an *accepting state condition* that has been omitted from the discussion in this paper.

[3] More precisely, this is the foreground part of the state, the background part is the canonical model of the background theory $\mathcal{T}$.

[4] There is no consistent update set when for example U is a parallel update of two distinct values to the same state variable.

Definition 2. Let P be a model program. The *LTS of* P, denoted by $[\![P]\!]$ is the LTS $(\mathcal{S}, \mathcal{S}_0, L, T)$, where $\mathcal{S}_0$, is the set of all P-states s such that $s \models I_P$; L is the set of all actions over A_P; T and $\mathcal{S}$ are the least sets such that, $\mathcal{S}_0 \subseteq \mathcal{S}$, and if $s \in \mathcal{S}$ and there is an action a that causes a transition from s to s' then $s' \in \mathcal{S}$ and $(s, a, s') \in T$.

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 run or the trace is *finite* if $\kappa < \omega$. We write *Traces*(P) for the set of all finite traces of P.

To illustrate the notion of a trace, consider $P = \mathit{FSMP}(\texttt{Req(\_,0,2)}^*)$. In this case $[\![P]\!]$ has a single state s_0 that is the empty mapping, because there are no state variables. There is a transition $(s_0, \texttt{Req}(v, 0, 2), s_0)$ in $[\![P]\!]$ for all values v. Thus a trace of P is any sequence of `Req`-actions whose second argument is 0 and third argument is 2, which explains the intuition provided in Example 2.

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 the parallel compositions of the update rules of the component model programs. We use '$\|$' to denote parallel composition of update rules (ASMs) [25].

Definition 3. Let P and Q be model programs such that $A = A_P = A_Q$. The *composition* $P \oplus Q$ is $(V_P \cup V_Q, A, I_P \wedge I_Q, (G^f_P \wedge G^f_Q, U^f_P \parallel U^f_Q)_{f \in A})$.

The following facts follow immediately from the definition of composition. Let P and Q (possibly with indices) denote model programs with the same action signature.

Fact 1 (Commutativity) $[\![P \oplus Q]\!] = [\![Q \oplus P]\!]$.

Fact 2 (Associativity) $[\![(P_1 \oplus P_2) \oplus P_3]\!] = [\![P_1 \oplus (P_2 \oplus P_3)]\!]$.

A straightforward technique to lift two model programs to use the same action signature, that is commonly used to compose FSMs and LTSs, is provided by the following basic action signature extensions.

Definition 4. Let P be a model program and f an action symbol not in A_P. The *enabling extension of* P *for* f, denoted by P^f, is the extension of P such that $A_{P^f} = A_P \cup \{f\}$ and $R^f_{P^f} = (\mathit{true}, \mathit{skip})$. The *disabling extension of* P *for* f, denoted by P^{-f}, is the extension of P such that $A_{P^{-f}} = A_P \cup \{f\}$ and $R^f_{P^{-f}} = (\mathit{false}, \mathit{skip})$.

Example 3 (OrderedRequests). Consider the following model program, called *OrderedRequests*.

```
var window as Set of Integer

[Action("Req(_,m,_)")]
Req(m as Integer)
  require m = Min(window)
  skip
```

It requires the second argument of a `Req` action to be the smallest element in `window`. Note that $I_{OrderedRequests} = \mathit{true}$ because the initial values of the state variables are unspecified, i.e. all states of $[\![\mathit{OrderedRequests}]\!]$ are initial states. The enabling extension $\mathit{OrderedRequests}^{\mathtt{Res}}$ adds the action rule $(\mathit{true}, \mathit{skip})$ for `Res` to *OrderedRequests*. The model programs $\mathit{OrderedRequests}^{\mathtt{Res}}$ and *Credits* in Example 1 have the same action signature.

The enabling (or disabling) extension of P for a set of action symbols F not in A_P is denoted by P^F (or P^{-F}). Note that $P^{\emptyset} = P^{-\emptyset} = P$. Let P and Q be model programs. Let $P \uplus Q \stackrel{\text{def}}{=} [\![P^{A_Q \setminus A_P} \oplus Q^{A_P \setminus A_Q}]\!]$ and $P \circledcirc Q \stackrel{\text{def}}{=} [\![P^{-A_Q \setminus A_P} \oplus Q^{-A_P \setminus A_Q}]\!]$. Intuitively, '$\uplus$' is an operator, where all actions whose symbol is not in the shared action signature are interleaved; '$\circledcirc$' on the other hand disables all such actions.

In the sequel, we overload the composition operator '$\oplus$' so that, for arbitrary model programs P and Q, $P \oplus Q$ stands for $P \uplus Q$.

3.1 Trace intersection

When composition is used in an unrestricted manner then 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. In general this happens if the composed model programs share state variables. The following proposition follows from [38, Theorem 1].

Proposition 1. *Let P and Q be model programs such that $A_P = A_Q$ and $V_P \cap V_Q = \emptyset$. Then $\mathit{Traces}(P \oplus Q) = \mathit{Traces}(P) \cap \mathit{Traces}(Q)$.*

The main reason why this property is important is that it makes it possible to do compositional reasoning over the traces in the following sence. If all traces of P satisfy a property φ and all traces of Q satisfy a property ψ then all traces of $P \oplus Q$ satisfy both properties φ and ψ.

3.2 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 $\mathit{WriteSet}(Q)$ be the set of all state variables of Q that appear as left hand sides of assignment rules in Q.

Proposition 2. *Let P and Q be model programs such that $A_Q \subseteq A_P$, and $\mathit{WriteSet}(Q)$ and V_P are disjoint. Then $\mathit{Traces}(P \oplus Q) \subseteq \mathit{Traces}(P)$.*

In this case composition of P and Q does not introduce traces that were not traces of P. A typical use of such composition is *guard strengthening* that is illustrated in Example 4.

Example 4. Let P be the model program *Credits* in Example 1 and let Q be the model program *OrderedRequests* in Example 3. In this case $V_Q = \{\texttt{window}\} \subset V_P$ and *WriteSet*$(Q) = \emptyset$. In $P \oplus Q$, Q strengthens the guard $G_P^{\texttt{Req}}$ so that all other choices for the parameter `m` besides the smallest element in `windows` are eliminated, which is a particular valid scenario for P. It is not possible to achieve this effect easily with "pure" composition as in Proposition 1.

4 Sample protocol

We consider an excerpt of the new SMB2 protocol, a successor of the Windows file-sharing client-server protocol SMB [28], which is used for filesharing between Vista machines and future Windows hosts. We consider a fixed client and a fixed server. The client sends *requests* to the server and the server sends *responses* back to the client. One can decompose SMB2 into various facets, that, when modeled individually, would comprise between 20 and 30 model programs. We look at two facets that are representative from the point of view of complexity and size. The excerpt is henceforth called SP.

- *Credit negotiation* describes how the client and the server need to use message ids, based on a sliding window algorithm.
- *Cancellation* describes how the client can cancel a previously sent request.

Concrete messages of the protocol are mapped to (abstract) actions where message fields that are not relevant for the given facets have been omitted. We consider three action symbols and the following message fields. Each message has a *command* field that indicates the operation communicated between the client and the server. This command field is either mapped to the first argument of the action, or it is mapped to the action symbol `Cancel` when the command is a special cancellation command.

- `Req` is a ternary action symbol that represents a request from the client to execute a command. A *request* is an action `Req(`c`,`m`,`n`)`, where c is a command, m is a message id and n is a number of requested credits.
- `Res` is an action symbol that takes four arguments and represents responses from the server. A *response* is an action `Res(`c`,`m`,`n`,`s`)` where c is a command, m is a message id, n is a number of granted credits, and s is a status value.
- `Cancel` is a unary action symbol that represents a "meta" request from the client to cancel a previous request. A *cancellation request* is an action `Cancel(`m`)` where m is a message id.

4.1 Credit negotiation

The client can use certain message identifiers to communicate with the server. The set of available message identifiers can be seen as a window of numbers that changes over time. The window is, strictly speaking, not a consecutive interval of numbers because the client does not have to use the available numbers in any particular order. This is an important aspect of the specification that leaves open implementation specific details

of the client-side of the protocol. An identifier of a request can only be used once. The client can ask for credits in the requests that it sends to the server in order to expand the window. The server may grant credits in its responses to the client. The number of credits granted in a response determines how the window grows or shrinks as time progresses. Note that the server may grant credits using different implementation specific algorithms the details of which are left open by the specification.

The *Credits* model program is defined uniformly for all of the commands, except for `Cancel`, see Example 1.

The state variable `window` is the set of all message ids that the client may use to send new requests to the server, `requests` is a map containing all the outstanding credit requests with message ids as keys, and `maxId` is the largest id that has been granted by the server. In the initial state of the model the only possible message id is 0, the maximum id is also 0, and there are no pending requests.

The `Req` action records in the state variable `requests` that message `m` has an outstanding credit request for `c` credits, and removes `m` from the window. The actual command (the first argument) is irrelevant here. The guard of this action rule requires that `m` appears in the window and that the requested number of credits is positive. The `Res` action updates the window with the new ids and updates the value of the maximum id. This action is enabled if the given id is an outstanding request, and the granted credits do not exceed the requested credits.

Validation The client *starves* if it runs out of message ids and cannot send further requests. An important safety requirement of the credits algorithm is that the client must not starve. Note that this does not mean that the server always has to grant at least one credit to the client in every response. It may be that the client has pending requests and the server will eventually grant the client more credits. Thus, the state invariant describing this safety condition is that if there are no pending requests then the window must be nonempty.

```
[StateInvariant]
ClientHasEnoughCredits()
  require (requests = {->}) implies (window <> {})
```

A natural question that arises here is if the *Credits* model program has any *unsafe* states, i.e., states that are reachable (through a trace) from the initial state that violate the state invariant. We use the finite state model program $\mathit{FSMP}(\texttt{Req(\_,0,2)}^*)$ in Example 2 to restrict the number of requested credits to 2 and the message id to 0. $[\![\mathit{Credits} \oplus \mathit{FSMP}(\texttt{Req(\_,0,2)}^*)]\!]$ is shown in Figure 1 and reveals an unsafe state reached by the trace `Req(_,0,2),Res(_,0,0,_)`. The labels on the states show the values of the state variables of the credits model program listed in the same order they appear in Example 1. We need to strengthen the guard of the `Res` action so that if there are no pending requests and the window is empty, then the granted number of credits must be at least one. Notice that if the window is empty and no credits are granted

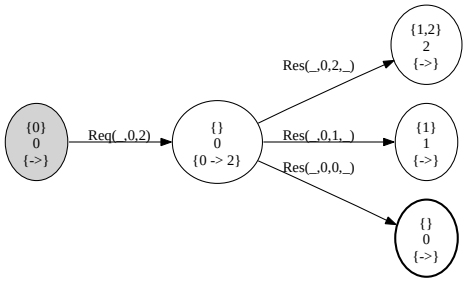

Fig. 1. Exploration of `Credits` $\oplus$ *FSMP*(`Req(_,0,2)`*).

then there must be at least two message ids pending when the new condition is checked (indicated by '`<--`' below), because the update rule will remove one of the ids.

```
[Action("Res(_,m,c,_)")]
Res(m as Integer, c as Integer)
   require m in requests
   require requests(m) >= c
   require c >= 0
   require requests.Size > 1 or window <> {} or c > 0  //<-- added condition
   window := window union {maxId + i | i in {1..c}}
   requests := RemoveAt(requests,m)
   maxId := maxId + c
```

4.2 Cancellation

Cancellation enables the client to cancel requests that have been sent to the server. In order to cancel a previously sent request with message id m, the client sends a cancellation message to the server that identifies the request to be cancelled by including its id in the message. The model program is shown in Figure 2. Notice that it is natural to refer to the `window` of the `Credits` model program for the valid message ids in a request.

The state variable `reqMode` records for each message id whether it has been sent or cancelled by the client. Initially, no request has either been sent or cancelled, so the value of `reqMode` is the empty map.

The `Req` action records the mode of the message as `Sent`. The `Cancel` action is always enabled, it updates a `Sent` mode to `Cancel` mode, and ignores the request otherwise (this behavior is needed for robustness). The `Res` action removes the pending request and requires that the request has indeed been cancelled by the client if the status is *false*. Note that the client may try to cancel a request but is too late to do so, when the server has already completed it but the response has not yet reached the client due to network latencies. Therefore, the status of a response to a request that the client tried to cancel, is either *true* or *false*, so that a potential race condition that would otherwise arise in the specification is avoided.

Validation *Cancellation* behaves uniformly for all message ids. It is therefore enough to fix a single message id, say 5, to expose all possible isomorphic behaviors. As above,

```
enum Mode
  Sent     //Client has sent the request
  Cancel   //Client has asked to cancel the request

var reqMode as Map of Integer to Mode = {->}

[Action("Req(_,m,_)")]
Req(m as Integer)
  require m in window
  reqMode := Add(reqMode,m,Sent)

[Action]
Cancel(m as Integer)
  if reqMode(m) = Sent
    reqMode := Add(reqMode,m,Cancel)

[Action("Res(_,m,_,status)")]
Res(m as Integer, status as Boolean)
  require m in reqMode.Keys
  require (status or reqMode(m) = Cancel) //status=false means cancelled
  reqMode := RemoveAt(reqMode,m)
```

Fig. 2. *Cancellation* model program.

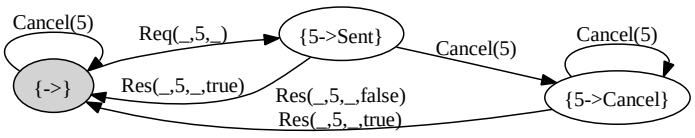

Fig. 3. Exploration of *Cancellation* $\oplus$ *Cancel5*.

we use a finite state model program to do this.

$$\mathit{Cancel5} = \mathit{FSMP}(\{\texttt{Cancel(5)},\texttt{Req(\_,5,\_)},\texttt{Res(\_,5,\_,\_)}\}^*)$$

$[\![\mathit{Cancellation} \oplus \mathit{Cancel5}]\!]$ is shown in Figure 3. The labels on the states show the value of `reqMode`. Using more message ids does not provide any additional useful information about *Cancellation*, but blows up the state space exponentially in the number of distinct message ids. With k distinct message ids there are 3^k states.

4.3 Composition

Once the individual facets have been modeled and validated in isolation, we can compose some or all of their model programs to validate their interactions. We use an additional model program called *Commands*: if a request with id m has command c, then the response with id m must also have command c, i.e., the server cannot respond with a command that is different from the one it was requested to execute. Note that it is convenient to refer to `window` of the *Credits* model program in the *Commands* model program for the domain of message ids. (The definition of the *Commands* model program is straightforward, using a map from message ids to commands.) We assume that the commands are `A` and `B`.[5] Note that only the first two arguments of `Req` and `Res` actions are relevant in the *Commands* model program. Moreover,

[5] In reality, SMB2 has 19 commands.

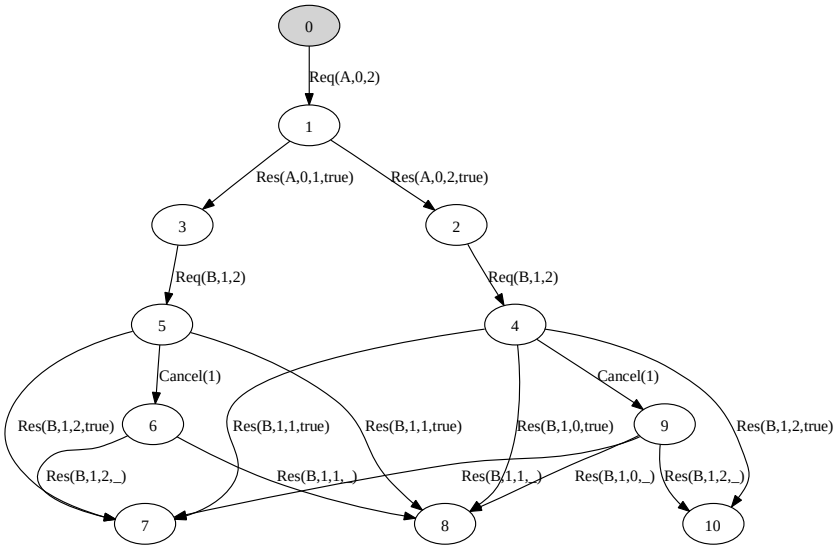

Fig. 4. Exploration of `SPscenario`.

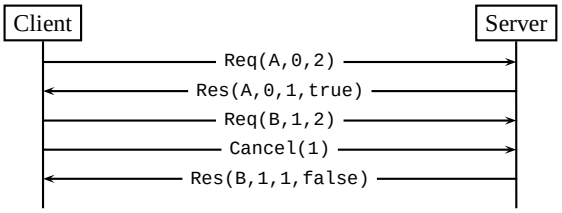

Fig. 5. A trace in Figure 4 from state 0 to state 8.

we use two scenario model programs: $AB = FSMP(\texttt{Req(A,\_,\_)Req(B,\_,\_)})$ and $M = FSMP(\{\texttt{Cancel(1),Req(\_,\_,2)}\}^*)$. *AB* restricts the client behavior so that a single `A` request is followed by a single `B` request. M restricts the client behavior so that only message 1 is ever cancelled, and all requests ask for two credits. Exploration of the composition

$$SPscenario = Credits \oplus Cancellation \oplus Commands \oplus AB \oplus M$$

is illustrated in Figure 4. All self-loops of `Cancel(1)` are hidden. All occurrences of placeholders (for the status argument of responses) indicate that both *true* and *false* are possible. Notice that the server behavior is unconstrained. In the states 7, 8 and 10, the value of `window` is, respectively, $\{2, 3\}$, $\{2\}$, and $\{2, 3, 4\}$, corresponding to all the possible ways in which the server could grant credits on the way from the initial state. A particular trace from the initial state to state 8 in Figure 4 is illustrated in Figure 5.

5 Implementation and experiences

All experiments in this paper have been made within the NModel framework using C# as the modeling language. The complete examples, as well as the full source of NModel itself, can be downloaded from [34]. The exploration and the composition examples have been carried out using the *mpv* utility of NModel.

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 sub-model program of the full model program. This mechanism can be used to construct separate facet model programs that share state variables, as discussed in this paper. The main composition operator in NModel assumes that the composed model programs do not share state variables.

The *FSMP* 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. [29, Theorem 2.1].

For conformance testing of the server, the client actions are declared *controllable* and the server actions (in this case responses) are declared *observable*. For online (or on-the-fly) test execution, with the *ct* utility of NModel, the composed model program is explored *lazily* by firing the actions one at a time, i.e. building up a trace of the model program incrementally. Due to the lazy exploration, scalability is not an issue. The discussion about *accepting states* has been omitted in this paper. Accepting states are used to define states where a trace may end, thus providing a way to finish a test in a clean way.

Model program analysis in NModel is based on explicit state exploration over abstract states. Much of the algorithmic support builds on earlier work in Spec Explorer [37]. In addition, the exploration includes a pruning technique based on isomorphism checking of states that use objects and unordered data structures [40].

NModel does currently not support symbolic analysis. We are investigating an SMT approach for doing reachability analysis of model programs [36], where we use Z3 [41, 5] for our implementation, as it supports background theories [17, 16] for arithmetic as well as sets and maps. A prototype is being implemented for a fragment of model programs written in AsmL. Integration of this analysis into NModel is future work.

The entire SMB2 specification contains over 300 pages of natural language specification and corresponds to roughly 20 facets The specification is written in a way where the different facets are specified in separate sections of the document and therefore the corresponding model programs are closely tied to these sections. Thus, having separate facet model programs matches well with the style of the natural language specs and makes it possible to do requirements tracking in the corresponding model programs.

The internal version of the modeling tool based on model programs is called Spec Explorer 2007 and is being developed and used internally in Windows as a core technology for protocol modeling and model-based testing. In Spec Explorer, model programs and composition are used for modeling and scenario control of industrial application-level network protocols. The entire protocol SMB2, has been modeled. The use of composition between contract model programs and model programs for scenario control

(test purposes) is one of the core techniques for controlling exploration [24]. For complex protocols it may be hard to identify facets due to dependencies. A crude classification of the protocols we have looked at is whether remote procedure call or message passing is being used, where SMB2 belongs to the latter kind. Being able to decompose a large protocol into facets is crucial for the latter kind of protocols.

At least half of the effort in model-based conformance testing of protocols is actually spent in harnessing of the implementation. A big part of this effort goes into implementing a protocol-specific adapter from concrete messages on the wire to abstract actions. When defining a mapping from concrete messages on the wire not all of the fields of messages are relevant. For example, some of the fields in a message are solely related to well-formedness of the message structure, checking of which can be part of a message validation layer that is orthogonal to the behavioral model.

6 Related work

The notion of facets as behavioral aspects of a protocol is similar to protocol *features*. Feature oriented specifications have a long standing in the telecommunication industry [42], because it makes specifications easy to change and individual features easy to understand, but it also introduces semantic challenges due to unintended feature interactions [10]. More recently, features, as increments of program functionality, are being used in *feature oriented programming* (FOP) for step-wise refinement of systems, and are supported by theory and tools using algebraic specifications [4]. In FOP, features are viewed as program transformations, and the purpose is to support feature oriented development through program synthesis and generative programming [4]. This is quite different from model programs that provide a partial view of the expected behavior of a system as an LTS, where the system itself is a black box, that is typically a combination of different applications from different vendors. However, the relationship between the mathematical underpinnings of model programs and FOP deserves a closer look.

Composition of model programs is a lazy automata-theoretic composition of the underlying LTSs, where actions are composed by unification. The unification between action parameters happens through the conjoined action guards. The motivation comes from the domain of model-based testing and analysis tools such as Spec Explorer [37]. A survey of model-based approaches to software modeling, with an emphasis on testing, is given in the recent book [35]. The notion of model program composition is a simplified and extended version of parallel composition of model programs in [38]. Work related to other forms of composition of automata is discussed in [38]. The use of several feature classes within in a single C# model program in NModel [34] allows for sharing of state variables across features. This enables state-dependent parameter generation and guard strengthening, which is, in general, not possible with composition of model programs with disjoint state signatures. The semantics of model programs can also be formulated in terms of labeled Kripke structures. This formulation has the advantage that one can adapt techniques that are used for model checking of temporal properties of concurrent software systems, including counterexample-guided abstraction refinement and compositional reasoning [12].

In aspect oriented programming two concerns crosscut when the related method behaviors intersect [19]. In the current paper the crosscutting of concerns corresponds to interacting behaviors between different facets of a protocol. The sharing of information is achieved through unification of actions, that allow data to be shared between traces but make the sharing explicitly visible in action traces. Model program composition might be a viable approach for formalizing certain forms of composition of trace based aspects [18] or model weaving of stateful aspects in aspect oriented modeling [13].

The main application of model programs is 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. This is related to aspect oriented approaches to trace monitoring [3]. In the context of testing of reactive systems with model programs [39], the action symbols are separated into controllable and observable ones. In that context the semantics of a model program as an LTS [31, 32] is fundamental in order to use IOCO [9], or refinement of interface automata [14], for formalizing the conformance relation.

Model program composition as defined in this paper is independent of the mechanism of exploration or analysis. Various approaches, including explicit state exploration [30] as well as symbolic reachability analysis [36], may be applied. The main difference compared to composition of *action machines* [23] 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 and are an integral part of the composition, using a three-valued logic. More about model-based testing applications and further motivation for the composition of model programs can be found in [11, 23, 39, 37].

Model programs are also related to symbolic transition systems that have an explicit notion of data and data-dependent control flow [20].

The *FSMP*(ρ) construction introduced here is a subset of a more general coordination language approach for scenario control called *Cord* [22].

Besides protocol modeling, model program composition is also being investigated as a technique for modeling and analyzing scheduling problems in embedded real-time systems [27].

When considering *interaction* of model programs that require synchronization or communication on objects rather than actions, then composition of model programs may be too limited. A more general foundation can be based on interactive abstract state machines [7].

Conclusion

The modeling approach introduced in this paper is being applied in a variety of industrially relevant modeling and testing contexts. In particular, model programs are being adopted as a technique for protocol modeling within Microsoft. The use of composition of model programs is an important part of this effort that enables scenario control as

well as a divide-and-conquer approach to model complex protocols. Individual facet model programs can be analyzed separately, they can be composed for interoperability analysis and for constructing the oracle for the full protocol model for test case generation and conformance testing.

References

1. Spec Explorer. URL:http://research.microsoft.com/specexplorer, released January 2005.
2. AsmL. URL: http://research.microsoft.com/fse/AsmL/.
3. P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhoták, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble, and M. Verbaere. Aspects for trace monitoring. In *FATES/RV'06*, volume 4262 of *LNCS*, pages 20–39. Springer, 2006.
4. D. Batory. A tutorial on feature oriented programming and the AHEAD tool suite. In *Proc. Summer School on Generative and Transformation Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 3–35. Springer, 2006.
5. N. Bjørner and L. de Moura. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08)*, LNCS. Springer, 2008.
6. A. Blass and Y. Gurevich. Background, reserve, and gandy machines. In *Proc. 14th Annual Conference of the EACSL on Computer Science Logic*, pages 1–17. Springer, 2000.
7. A. Blass and Y. Gurevich. Ordinary interactive small-step algorithms, I. *ACM Transactions on Computation Logic*, 7(2):363–419, April 2006.
8. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
9. E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In *Summer School MOVEP'2k*, volume 2067 of *LNCS*, pages 187–193. Springer, 2001.
10. M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
11. 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.
12. S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proc. of the 4th Intl. Conf. on Integrated Formal Methods (IFM '04)*, volume 2999 of *LNCS*, pages 128–147. Springer, 2004.
13. T. Cottenier, A. van den Berg, and T. Elrad. Stateful aspects: the case for aspect-oriented modeling. In *AOM'07*, pages 7–14. ACM, 2007.
14. L. de Alfaro. Game models for open systems. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.
15. L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE'01*, pages 109–120. ACM, 2001.
16. L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *21st Int. Conf. on Automated Deduction, (CADE'07)*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
17. L. de Moura and N. Bjørner. Model-based theory combination. In *5th International Workshop on Satisfiability Modulo Theories, (SMT'07)*, pages 46–57, Berlin, Germany, July 2007.
18. R. Douence, P. Fradet, and M. Südholt. *Aspect-Oriented Software Development*, chapter Trace-based Aspects, pages 201–218. Addison Wesley, September 2004.
19. T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Commun. ACM*, 44(10):33–38, 2001.

20. L. Frantzen, J. Tretmans, and T. Willemse. A symbolic framework for model-based testing. In *FATES/RV'06*, number 4262 in LNCS, pages 40–54. Springer, 2006.
21. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA'02*, volume 27 of *Software Engineering Notes*, pages 112–122. ACM, 2002.
22. W. Grieskamp and N. Kicillof. A schema language for coordinating construction and composition of partial behavior descriptions. In *5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM)*, 2006.
23. W. Grieskamp, N. Kicillof, and N. Tillmann. Action machines: a framework for encoding and composing partial behaviors. *IJSEKE*, 16(5):705–726, 2006.
24. W. Grieskamp, D. MacDonald, N. Kicillof, A. Nandan, K. Stobie, and F. Wurden. Model-based quality assurance of windows protocol documentation. In *First Intl. Conf. on Software Testing, Verification and Validation, ICST*, Lillehammer, Norway, April 2008.
25. Y. Gurevich. *Specification and Validation Methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, 1995. url: research.microsoft.com/ gurevich/Opera/103.pdf.
26. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of asml. *Theor. Comput. Sci.*, 343(3):370–412, 2005.
27. J. Helander, R. Serg, M. Veanes, and P. Roy. Adapting futures: Scalability for real-world computing. In *Proc. 28th IEEE Real-Time Systems Symposium*, pages 105–116. IEEE, 2007.
28. C. Hertel. *Implementing CIFS: The Common Internet File System*. Prentice Hall, 2003.
29. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
30. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2007.
31. R. Keller. Formal verification of parallel programs. *Communications of the ACM*, pages 371–384, July 1976.
32. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th annual ACM Symposium on Principles of distributed computing*, pages 137–151. ACM, 1987.
33. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
34. NModel. http://www.codeplex.com/NModel, released May 2007.
35. M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006.
36. M. Veanes, N. Bjørner, and A. Raschke. An SMT approach to bounded reachability analysis of model programs. In *FORTE'08*, LNCS. Springer, 2008. In this volume.
37. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008.
38. M. Veanes, C. Campbell, and W. Schulte. Composition of model programs. In J. Derrick and J. Vain, editors, *FORTE 2007*, volume 4574 of *LNCS*, pages 128–142. Springer, 2007.
39. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proc. ESEC/FSE-13*, pages 273–282. ACM, 2005.
40. M. Veanes, J. Ernits, and C. Campbell. State isomorphism in model programs with abstract data structures. In *FORTE'07*, volume 4574 of *LNCS*, pages 112–127. Springer, 2007.
41. Z3. http://research.microsoft.com/projects/z3, released September 2007.
42. P. Zave. Feature interactions and formal specifications in telecommunications. *Computer*, 26(8):20–29, 1993.