Specifying Component Behavior with Port State Machines

Vladimir Mencl [1]

Charles University, Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
mencl@nenya.ms.mff.cuni.cz

Abstract

Protocol State Machines (PSM) in UML 2.0 [13] describe valid sequences of operation calls. To support modeling components, UML 2.0 introduces a *Port* associated with a set of provided and required interfaces. Unfortunately, a PSM is applicable only to a single interface, either a provided or required one; moreover, nested calls cannot be modeled with a PSM. Furthermore, the definition of *protocol conformance* is rather fuzzy and reasoning on this relation is not possible in general; thus reasoning on consistency in component composition is not possible with PSMs.

Behavior Protocols [17] capture the behavior of a component via a set of traces. A textual notation similar to regular expressions is provided to approximate the behavior with a regular language. In [1,17], the *compliance* relation and *consent* operator are defined to reason on consistency of component composition; a verifier tool [18] is available for the compliance relation.

In this paper, we propose the Port State Machine (PoSM) to model the communication on a Port. Building on our experience with behavior protocols, we model an operation call as two atomic events *request* and *response*, permitting PoSM to capture the interleaving and nesting of operation calls on provided and required interfaces of the Port. The trace semantics of PoSM yields a regular language. We apply the compliance relation of behavior protocols to PoSMs, allowing us to reason on behavior compliance of components in software architectures; the existing verifier tool can be applied to PoSMs.

Key words: behavior specification, consistency reasoning, state machines, software components, composition, UML 2.0

[1] This work was partially supported by the Grant Agency of the Czech Republic project 102/03/0672. The results will be applied in the OSMOSE/ITEA project.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

1.1 UML 2.0: State Machines and Protocol State Machines

The Unified Modeling Language (UML) [12] features StateMachines based on the widely recognized State-chart notation [8]; the execution of a State Machine can be observed in terms of *events* accepted and *actions* executed (potentially overlapping). The upcoming new version of the standard, UML 2.0 [13], introduces a specialization of State Machine, the Protocol State Machine (PSM), which can be used to model the ordering of operation calls on a Classifier (typically an Interface). Moreover, UML 2.0 introduces the concepts StructuredClassifier and EncapsulatedClassifier, providing support for modeling internal structure and featuring Ports; a Port is associated with a set of provided and required interfaces. Based on these concepts, a Component may be captured in a UML model, employing a possibly hierarchical component model; the external communication of the component is encapsulated in the component's Ports.

In component-based software engineering, a basis for reasoning on "compatibility" of software components is highly desirable in order to validate software architectures and define substitutability of components.

UML explicitly considers *"conformance"* of PSMs; however, the role of conformance is limited to explicitly declaring, via the ProtocolConformance model element, that a *specific* StateMachine (possibly a PSM) conforms to a *general* PSM. Note that UML defines the semantics of protocol conformance only partially (based on structural equivalence and matching guards on transitions); it is not clear under which circumstances protocol conformance may be declared and thus, it is not feasible to automatically decide on protocol conformance.

UML employs the protocol conformance in the Components framework, requiring *realization* of a Component (possibly a StateMachine specifying the component) to be conforming with specifications of all its Interfaces. Moreover, when a required interface I^R is connected to a provided interface I^P, the PSM of I^R must be conforming to the PSM of I^P. However, with no exact definition of protocol conformance, validating component architectures is not feasible.

1.2 Motivations

Although the State Machines in UML permit modelers to clearly communicate ideas to each other, they are not suitable to be used as the basis for defining "compatibility" of components. The observable behavior of a component is typically captured as communication on its *provided* and *required* interfaces [4,5,6,15]. However, in UML State Machines, significantly different mechanisms are employed to specify events received and sent. Events received (in case of a component corresponding to operations on the provided interfaces), are captured as *triggers* associated with transitions of the state machine. A

State Machine uses Activities to specify its responses to events received (i.e., events sent and internal actions). An Activity (a Petri-net like abstraction in principle) consists of Actions, some of these actions correspond to sending events. However, the spectrum of actions is rather huge and it is not possible to establish a one-to-one correspondence between the triggers and actions related to a communication; thus, it is not possible to derive the behavior resulting from the composition of communicating components (exchanging events) specified with State Machines.

A Protocol State Machine (further PSM), a refinement of the (generic) *behavioral* State Machine, imposes a restriction on its transitions, requiring that no Activities are associated with the execution of the PSM. Consequently, only one "direction of communication" can be captured with a PSM. The communication is specified independently of the direction of communication, only the way an interface described by a PSM is used in a Port determines whether the events captured by the PSM are received (for provided) or sent (for a required interface); a PSM cannot describe the interplay of events on the provided and required interfaces.

UML State Machines employ the *run-to-completion* semantics, i.e., only after a transition of a State Machine completes can another event be processed. Thus, while executing a method (modeled, e.g., as the effect activity of the transition), no other event may be processed by the State Machine, i.e., no other method call may be accepted. Thus, State Machines cannot capture interleaving of calls (several incoming calls processed at the same time), and neither nested calls (e.g., a call-back or statically limited recursion), nor they support (unlimited) recursion.

Surprisingly, the situation is no easier in PSMs – although no activity corresponding to the operation called is included in a PSM, a transition completes only after the method implementing the operation completes. Therefore, no call may be accepted before the call being processed completes and thus, the same restrictions on call interleaving and nested calls apply to PSMs. Consequently, although a PSM specifies a sequence of operation calls, the communication on a Port, associated with provided and required interfaces described by PSMs, cannot be captured with *traces* for further behavioral reasoning, due to the non-atomicity of the events (operation-call) in the sequences described by the PSMs. Moreover, the descriptions cannot capture nesting and interleaving of calls on the Port, although this is a common pattern in component communication.

Considering the lack of well-defined semantics, establishing a rigorously defined compatibility relation upon the behavior of PSMs is not feasible. Among others, UML assumes a constraint language to be used for guards of transitions, but no constraint language is prescribed; OCL is provided only as one of the options. Moreover, even when assuming OCL to be the only constraint language permitted, such a relation would hardly be decidable for the following reasons: (i) OCL expressions may access the attributes of the classifier, i.e.,

an internal state with potentially unlimited state space and (ii) Events may be deferred and processed later, thus the automaton gets a stack (though no semantics is given for the order of retrieval; thus the event pool rather resembles a bag). Here, the consensus is that verification of compliance is feasible only on regular automata (or other abstractions with equivalent expressive power). In certain cases, the relation may be decidable for a context-free grammar / stack automaton; however, actually evaluating (computationally) such a relation is likely to be unfeasible in general. A compliance relation is typically defined on regular languages, e.g., a decidable relation is defined in [17]; the work on the consent operator [1] provides an alternative approach [2]. Note that the approach taken in [10,11] also uses a subset of statecharts that can be converted to a finite LTS.

In case a trace model can be defined for the sequences of events described by a state machine (here, it is essential that the events are atomic), reasoning on compliance may be possible. When defining behavioral compliance, we see as important that (i) compliance is based only on the behavior described and not on the structure of the specification (ii) compliance is unambiguously defined (iii) deciding on compliance can be achieved in an automated way. Unfortunately, none of these is the case for ProtocolConformance defined in UML 2.0 (as discussed in Sect. 1.1).

Last but not least, we miss a layer of description between a PSM (focused on a single interface) and a behavioral State Machine specifying a component, i.e., a layer suitable for specifying communication on a Port (of a component).

Thus, the issues we identified are: (i) State Machines in UML do not capture interleaving of sent and received events. (ii) Composition of State Machines is not possible (iii) The form State Machines use does not permit establishing a decidable compliance relation. (iv) A specification mechanism is missing to capture the communication on a Port.

1.3 Goals and Structure of the Paper

In [17], our research group developed Behavior Protocols, modeling behavior of *agents* as traces of atomic events. Applied to the SOFA component model [15], behavior protocols capture the ordering of operation calls issued and handled by a SOFA component. Nesting of other events (possibly also operation calls) within an operation call is supported. Moreover, a decidable compliance relation is defined; a verifier tool [18] for checking this relation is available. SOFA is a hierarchical component model; a component (either *primitive* or *composed*) communicates with its neighboring components via a set of *provided* and *required* interfaces. The abstraction of a software component, as considered in SOFA, employs a set of features comparable to those available in UML 2.0 Components.

Considering the motivations discussed in Sect. 1.2, we propose *Port State Machine* (PoSM) with the following goals: (1) Provide a notation that allows

to capture interleaving of events sent and received (by a Port of a Component) and support nested calls in such a way that the behavior can be captured with a trace model based on atomic events. (2) Moreover, a verifiable compliance relation should be defined for PoSMs.

This paper is structured as follows: Sect. 2 introduces the Port State Machines (PoSMs), in Sect. 3, we show how composition verification can be achieved with PoSMs; a case study follows in Sect. 4. Sections 5 and 6 evaluate the contribution, discuss related work and line out future research; the paper concludes in Sect. 7.

1.4 Note on Conventions Used

In this paper, PSM stands for Protocol State Machines (introduced by UML 2.0), while PoSM (at convenience pronounced "possum") stands for Port State Machines, proposed in this paper. A sans-serif font is used to distinguish UML metamodel identifiers (names of packages, metaclasses, associations and attributes).

2 Port State Machines

We propose Port State Machines, building upon the UML 2.0 Protocol State Machines. To model operation calls (inherently non-atomic) with atomic events, PoSMs capture an operation call with two events, *request* (corresponding to start of the operation call) and *response* (completion of the operation call). Moreover, PoSMs explicitly distinguish between *sent* and *received* events. Here, an operation call handled on a provided interface is represented by a received request event and a sent response event; in a similar way, an operation call issued on a required interface is represented by a sent request event and a received response event. To hide such technical details from the modeler, PoSM notation defines convenient shortcuts. In Fig. 1, a Port State Machine explicitly specifies the request and response events, while in Fig. 3, the same behavior is described with the notation shortcuts (these will be described in Sect. 2.3).

2.1 PortStateMachine Metamodel

We define PortStateMachine as an extension of UML 2.0 ProtocolStateMachine, employing the UML 2.0 extension mechanisms. In Fig. 2, we show the newly introduced metaclasses PortStateMachine and PortTransition, as well as the related classes of the UML StateMachine specification to provide context for our extension.

PortStateMachine is defined as a subclass of ProtocolStateMachine. Thus, a PoSM contains one or more regions; a Region contains vertexes and transitions. A Transition connects a source vertex to a target vertex. A Vertex may be a

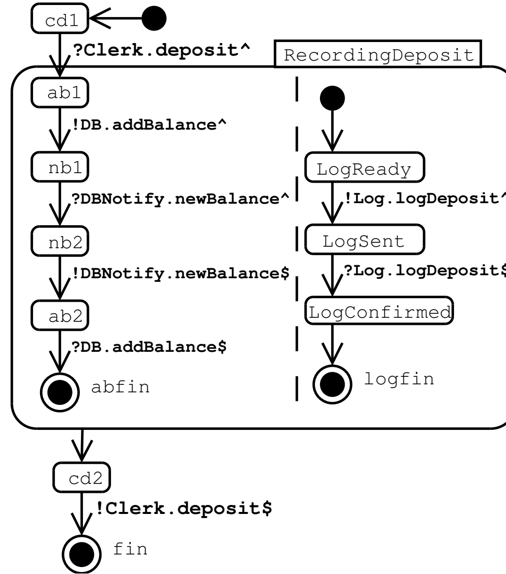

Fig. 1. Port State Machine with explicit request and response transitions

PseudoState or a State. A PseudoState is a syntactic construct to model entry and exit points of regions.

A State may contain zero or more regions. A State not containing any region is a *simple* state. FinalNode is a specialization of a State representing the completion of a region; a FinalNode may not contain any regions or have outgoing transitions.

A State containing one or more regions is a *composite* state, a syntactic construct to provide hierarchical grouping of states. When a simple state is active, all its containing composite states are active. In an active composite state, one of its substates is active.

A State containing more than one region is an *orthogonal state*. Orthogonal states model concurrent execution; in an active orthogonal state, a substate is active in each of its regions and a transition may be taken in any of the regions.

Example 2.1 In Fig. 1, `RecordingDeposit` is an orthogonal state; the calls `DB.addBalance` and `Log.logDeposit` progress in its orthogonal regions independently.

A PortTransition represents a single atomic communication event. PortTransition is a subclass of ProtocolTransition (which in turn is a subclass of Transition). A PortTransition must have exactly one trigger; the trigger must be a CallTrigger and must refer to an Operation of an Interface of the Port associated with the PoSM. PortTransition introduces two additional attributes, both of an enumerate type: OperationCallPart captures whether the transition represents the request or response part of the operation call. CommunicationDirection specifies whether the event is received or sent, its value must be sent for a request on a required interface or a response on a provided interface, and

received in the opposite cases (response on required interface or request on a provided interface).

Example 2.2 In Fig. 1, transition from `LogReady` to `LogSent` denotes sending a request for operation `Log.logDeposit`, while the ongoing transition to `LogConfirmed` denotes receiving the response for this operation.

With the goal to provide trace semantics and facilitate a compliance relation (as discussed in 1.2), PoSMs introduce the following additional restrictions on PortStateMachine instances and its contained elements:

(i) A transition in a PortStateMachine must be either a PortTransition or a ProtocolTransition; a transition that is not a PortTransition may not specify any triggers, i.e., can only accept the *completion event*. A PortTransition may only originate in a State (but may target a PseudoStates).

(ii) A transition in a PortStateMachine may not specify any constraints – its guard, preCondition and postCondition associations must be empty.

(iii) The deferrableTrigger association of each State must be empty.

(iv) Only one transition is taken for a single occurrence of an event, even when

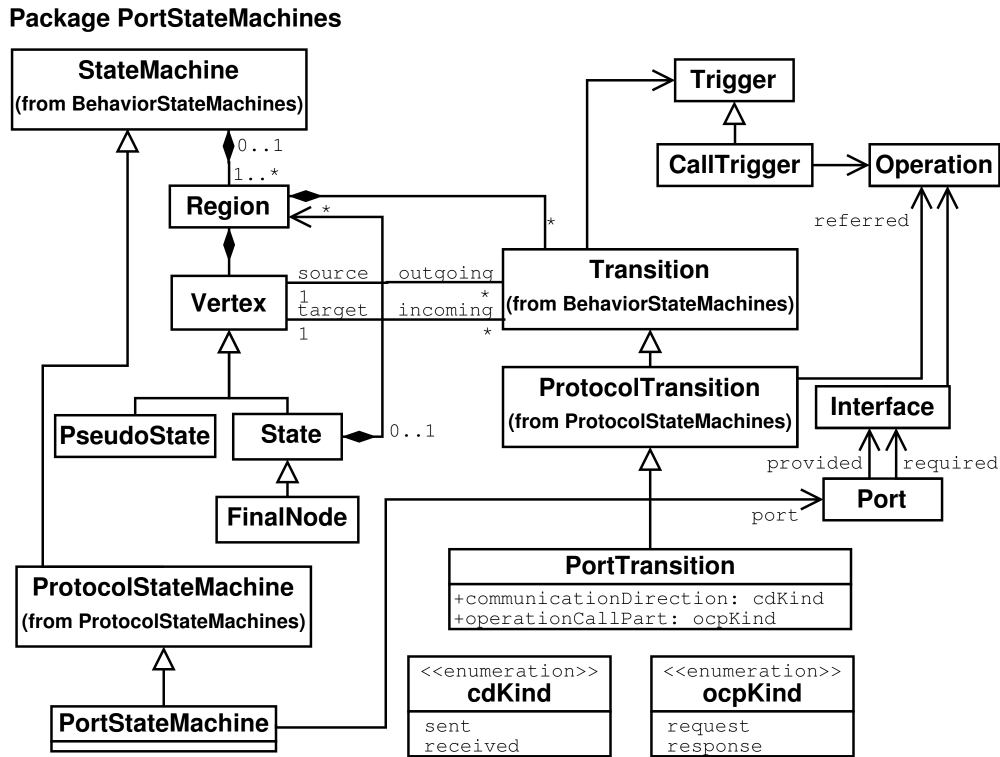

Fig. 2. Port State Machines abstract syntax: definition of PortStateMachine and PortTransition. For space constraints, the owning package name is shown only for selected metaclasses. Metaclasses Region, Vertex, PseudoState, State and FinalNode are owned by BehavioralMachines, Trigger and CallTrigger by CommonBehaviors, Operation by Kernel, Interface by Interfaces and Port by CompositeStructures.

multiple transitions in different regions of an orthogonal state specify the same trigger (opposite to the UML 2.0 orthogonal state semantics where all such transitions are taken simultaneously).

(v) The kind of a PseudoStates in a PoSM must be either initial or fork. For the sake of simplicity of the PoSM definition, we omitted the other PseudoState kinds: choice and junction (not meaningful without guards), deepHistory and shallowHistory (complex semantics; can be replaced with increased state space), join (complex semantics, can be partially replaced with FinalNodes) and exit and terminate (we focus on complete traces).

(vi) A transition from PseudoState may only target a vertex recursively contained by the region containing the PseudoState. (I.e., may not cross state boundaries outwards).

The restrictions specified above together with the constraints initially specified by UML 2.0 [13] assure certain properties; we highlight here those that will be used later:

(i) A transition originating from a State may cross several boundaries of containing states outwards, then cross several boundaries of composite states inwards and finally targets a Vertex,

(ii) A transition originating from a PseudoState is not a PortTransition but only a ProtocolTransition. A transition from an initial PseudoState within a region r either targets a Vertex directly contained by r or a Vertex within a State contained by r. Only one transition may originate from an initial PseudoState.

(iii) Given a fork PseudoState p_f contained in region r containing also a composite state s, multiple transitions may originate from p_f, each targeting a Vertex in a different region of s.

Example 2.3 The PoSM shown in Fig. 1, after receiving a request for operation `Clerk.deposit`, enters the orthogonal state `RecordingDeposit`. After both its regions complete, the PoSM eventually sends a response for the `Clerk.deposit` operation.

2.2 Trace Semantics of Port State Machines

We define the semantics of a PoSM P^A via the traces generated by P^A. We model the behavior as traces of *state events* and *communication events* forming the *communication language* and *execution language* of P^A.

Definition 2.4 Let St be the set of all states and let Reg be the set of all regions, directly or indirectly contained in a PoSM P^A. For a region $r \in Reg$, we denote $States(r)$ the set of states directly contained by r. In the same vein, for a state $s \in St$, $Regions(s)$ is the set of regions directly contained by s. $Regions(P^A)$ is the set of top-level regions directly contained by P^A.

Definition 2.5 State s_i is a *substate* of s_j, if there is $r \in Regions(s_j)$ such

that $s_i \in \mathit{States}(r)$. A state s_j *recursively contains* a state s_i if s_i is a substate of s_j, or there is a substate s_k of s_j such that s_k recursively contains s_i.

A region r_j recursively contains a state s_i, if $s_i \in \mathit{States}(r_j)$ or there is $s_j \in \mathit{States}(r_j)$ such that s_j recursively contains s_i.

Definition 2.6 Let SL be set of labels for states in St and OL be the set of labels for operations associated with transitions of P^A.

We define the domain of state events $SE = \{\mathit{entry}, \mathit{exit}\} \times SL$ and the domain of communication events $CE = \{\mathit{sent}, \mathit{received}\} \times OL \times \{\mathit{request}, \mathit{response}\}$. The set CE is the domain of events for communication traces of P^A and the set $S = SE \cup CE$ is the domain of events for execution traces of P^A. Note that state events only capture entering or leaving a State, but not a PseudoState.

Definition 2.7 A *configuration* c of P^A is a subset of St for which both the following conditions hold:

(i) for each region $r \in \mathit{Reg}$, c contains at most one state $s \in \mathit{States}(r)$

(ii) if $s_i \in c$ and s_i is a substate of s_j, then $s_j \in c$.

A state s_i is *active* in c if $s_i \in c$. A region r is *active* in c if there is a state $s_i \in \mathit{States}(r)c$ such that $s_i \in c$.

A configuration c is *stable*, if each top-level region of P^A is active and for each state $s_i \in c$, all regions of s_i are active.

Definition 2.8 The *label* of a PortTransition T associated via its trigger with an operation op is the event $e = < cd_T, label_{op}, ocp_T > \in CE$, where cd_T and ocp_T are the communication direction and operation call part attributes of T and $label_{op} \in OL$ is the label for op. A transition not associated with an operation does not have a label.

In a configuration c (of P^A) containing a state s_i, P^A may take a transition T originating from s_i, iff at least one of the following conditions holds:

(i) T has no label, and either s_i has no regions, or the active state of all regions of s_i is a FinalNode.

(ii) T is a PortTransition labeled with event $e \in CE$ and there is no state $s_j \in c$ such that (1) s_i recursively contains s_j and (2) a transition U also labeled e originates from s_j (in this case, we say that U has *higher priority* than T).

The innermost region recursively containing the source and target vertexes of T is the *least common ancestor* (LCA) of T, denoted $r_{lca,T}$. The LCA configuration $c_{lca,T}$ is obtained from c_1 by removing all states recursively contained in $r_{lca,T}$.

Example 2.9 For the PoSM shown in Fig. 1, $\{\mathit{RecordingDeposit}, \mathit{ab1}, \mathit{logfin}\}$ is a stable configuration, in which $\mathit{ab1} \mapsto \mathit{ab2}$ is the only legal transition and the left region of *RecordingDeposit* is the LCA.

Configuration $\{\mathit{RecordingDeposit}, \mathit{abfin}, \mathit{logfin}\}$ is also a stable configura-

tion of this PoSM, where the only legal transition is *RecordingDeposit* $\mapsto$ *cd2*; here, the single topmost region of the PoSM is the LCA.

Definition 2.10 From $c_{lca,T}$, T determines the target stable configuration the following way:

(i) T is an *engaged* transition.

(ii) An engaged transition targeting a State s causes s to become *active.*

(iii) An engaged transition targeting a PseudoState p causes transitions outgoing from p to be *engaged.*

(iv) All containing states of a state that becomes active become active (if they are not active yet).

(v) For each composite state that becomes active, all regions become active. If an engaged transition T_i targets a vertex in a region r, r becomes active by T_i *explicitly* and T_i determines the active state of r. Otherwise, r becomes active *implicitly* and the transition originating from an initial PseudoState of r becomes engaged. If there is no such transition, the model is ill-formed. Eventually, after processing all engaged transitions and according to these rules, all regions that must become active have an active state selected, yielding a stable configuration c_2. By observation, $c_{lca,T} \subseteq c_1 \cap c_2$.

A single *run-to-completion step* of P^A from a stable configuration c_1 to a stable configuration c_2 initiated by a transition T is captured with a trace $t_{T,k}$, acquired as concatenation of parts $t_{T,exit}$, $t_{T,com}$ and $t_{T,entry}$. The first part $t_{T,exit}$ is a sequence of state exit events reflecting the transformation of c_1 to $c_{lca,T}$ via a sequence of valid configurations $c_{j,exit}$. Next, $t_{T,com}$ contains the label of T if T is a PortTransition, or is a null sequence otherwise. Finally, $t_{T,entry}$ is a sequence of state entry events reflecting the transformation of $c_{lca,T}$ to c_2 via a sequence of valid configurations $c_{j,entry}$. Note that due to the loose ordering constraints on entry and exit events for orthogonal states, there may be multiple traces $t_{T,k}$ capturing the *run-to-completion step* from c_1 to c_2 via T.

The initial stable configuration c_{init,P^A} of P^A is determined by transitions from initial PseudoStates of top-level regions of P^A. Configuration $c_{fin,P^A,k}$ is a final configuration of P^A if the active state of each top-level region of P^A is a FinalNode.

We capture a single *run* of P^A with a trace t_{P^A}, acquired as concatenation of parts $t_{P^A,entry}$, $t_{P^A,k}$ and $t_{P^A,exit}$, where $t_{P^A,entry}$ is the sequence of state entry events to reach the initial stable configuration c_{init,P^A} of P^A from the empty configuration, $t_{P^A,k}$ is concatenation of a finite sequence of traces capturing a sequence of run-to-completion steps reaching a final configuration $c_{fin,P^A,k}$ from c_{init,P^A}, and $t_{P^A,exit}$ is a sequence of the state exit events to reach the empty configuration from a $c_{fin,P^A,k}$.

Definition 2.11 The set of all traces of all possible runs of P^A forms the *execution language* of P^A, denoted $LE(P^A)$. *Communication language* of P^A, denoted $LC(P^A)$ is the restriction of $LE(P^A)$ to the domain of communication events CE.

Example 2.12 The transition *RecordingDeposit* $\mapsto$ *cd2* of the PoSM shown in Fig. 1 may be captured with the following trace:

$$< \ exit_{abfin}, exit_{logfin}, exit_{RecordingDeposit}, entry_{cd2} \ >$$

This trace (which does not contain any communication event) may be followed in a run of the PoSM by a trace of the transition *cd2* $\mapsto$ *fin* (labeled with sending a response for *Clerk.deposit*; we use ? to denote receive, ! for send, ↑ for request and ↓ for response):

$$< \ exit_{cd2}, !Clerk.deposit\uparrow, entry_{abfin} \ >$$

The following example is a possible trace from the communication language of this PoSM:

$$\begin{aligned} < &?Clerk.deposit\uparrow,\ !Log.logDeposit\uparrow,\ !DB.addBalance\uparrow, \\ &?DBNotify.newBalance\uparrow,\ !DBNotify.newBalance\downarrow, \\ &?DB.addBalance\downarrow,\ ?Log.logDeposit\downarrow,\ !Clerk.deposit\downarrow \ > \end{aligned}$$

2.3 Notation

PoSMs introduce extensions to the UML state machine notation, with the goal to avoid an increase in complexity of the state machine diagrams, even when capturing the additional information required by PoSMs. In particular, the extensions permit to: (i) capture the additional attributes of a transition in its label, (ii) use implicit intermediate states and (iii) capture nested calls. The notation shortcuts are demonstrated in Fig. 3, concisely describing the same behavior as the PoSM in Fig. 1 (not employing the shortcuts).

The PoSM notation utilizes the notation of Behavior Protocols (BP) [16,17]. There, the event token $?a$ stands for receiving an event a and $!a$ for sending an event a. A call of an operation op is captured with a pair of atomic events; in the event labels, the suffix ↑ denotes request and ↓ response. E.g., sequence $?op\uparrow;!op\downarrow$ (here ; is the operator for sequencing) models receiving call of the operation op as receiving a request for op and sending a response. In BP, the shortcuts $?op$ and $!op$ stand for sequences $?op\uparrow;!op\downarrow$ and $!op\uparrow;?op\downarrow$; shortcuts $?op\{Prot\}$ and $!op\{Prot\}$ stand for $?op\uparrow;Prot;!op\downarrow$ and $!op\uparrow;Prot;?op\downarrow$ respective.

The notation for PoSMs employs these prefixes (?/!) and suffixes (↑/↓) in the event label to express the attributes of a `PortTransition`. Due to the limitations of the character set available in UML, we represent ↑ with `^` and ↓ with `$` respectively. We demonstrate the notation and the shortcuts in Fig. 4.

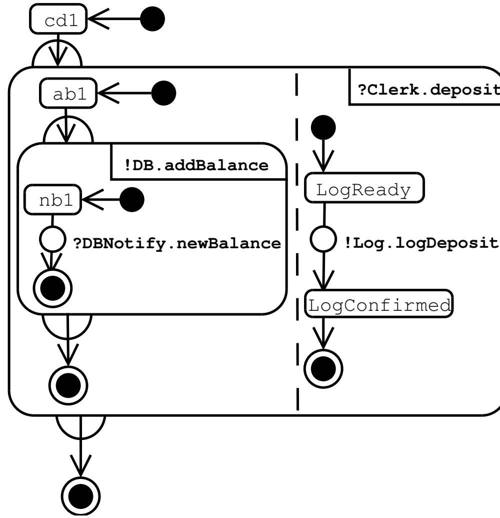

Fig. 3. Port State Machine employing *call transition* and *call state* shortcuts

Figure 4 (a) models receiving call of operation a, explicitly captured as two PortTransitions adjoined in an (explicit) intermediate state. Fig. 4 (b) employs a *call transition* as a shortcut to model the same sequence. A single call transition represents two PortTransitions and the intermediate state; mnemonically, the circle on the call transition reminds of the implicit intermediate state. The single label of the call transition ("?a") determines the trigger of both the PortTransitions; the communication direction of the first (request) transition is equal to the symbol used in the label, while the communication direction of the second (response) transition is the opposite.

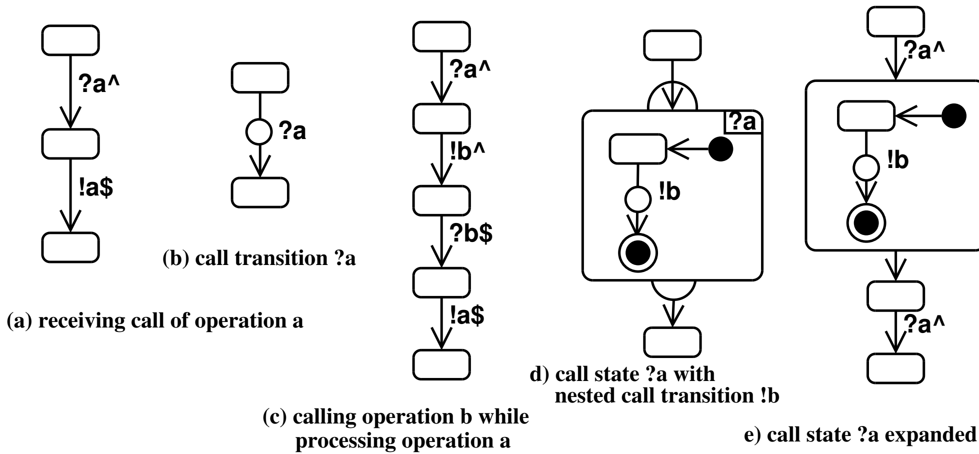

Fig. 4. Port State Machines notation

PoSM notation also provides a syntactic construct to model nested calls. In Fig. 4 (c), operation b is called while operation a is being processed. The same sequence of events can be captured with a *call state*, as demonstrated in Fig. 4 (d). The call state construct represents a structure of transitions and states, the core of which is a composite state containing the behavior that

occurs between the request and response.

In the same way as for a call transition, two PortTransitions (for request and response) are specified with only one occurrence of the operation call label. To preserve the general operation call semantics, a call state may only be entered with the request transition, may complete only after its internal behavior completes, and must complete with the response transition. Thus, a call state may have only one incoming and one outgoing transition. Moreover, to assure that the composite state may only exit with the completion event, the composite state exits with an unlabeled transition targeting an intermediate state, from which the response transition originates. Figure 4 (e) demonstrates the composite state, intermediate state and transitions represented by the call state shortcut in Fig. 4 (d).

The incoming transition of the call state must target the state itself, an initial PseudoState has to be used to specify where the region(s) of the call state start (a call state may have multiple regions). Syntactically, a call state employs the notation for a composite state and is distinguished with two semicircles attached to the top and bottom of the state; the operation call label is placed in the top-right corner.

Note that throughout this example, we used for brevity the symbols a and b to refer to an operation on an interface. Clearly, an identifier of the interface and an identifier of the operation are required to identify the operation unambiguously; in the other examples (e.g., figs. 1 and 3) the character "." (dot) is used to join these identifiers.

Also please note that we define the call state and call transition notation shortcuts by specifying how they expand into elements defined in the PoSM metamodel. We consider this approach to be the most efficient with respect to readability of the paper, in particular of the trace semantics definition. Alternatively, we might define call state and call transition in the metamodel and either extend the semantics definition also to these elements, or to define a transformation of a model employing these constructs to a model based only on the already considered metamodel elements; both these approaches are feasible.

2.4 Properties of Communication Traces

In this section, we define the well-formedness property of communication traces and show its relation to the PoSM notation shortcuts; moreover, we also claim that the communication language of a PoSM is a regular language, we support this claim with a proof sketch.

Definition 2.13 A communication trace t is *well-formed* if t can be transformed into an empty trace in a sequence of steps, where in each step i a pair of events e_i^{rq}, e_i^{rsp} representing a (single) call of operation $op \in OL$ is removed from t (e_i^{rq} preceding e_i^{rsp} in t). For receiving a call of op, $e_i^{rq} = \,!op\!\uparrow$ and $e_i^{rsp} = \,?op\!\downarrow$, for sending a call of op, $e_i^{rq} = \,?op\!\uparrow, e_i^{rsp} = \,!op\!\downarrow$.

If such a sequence of the removal steps exists that in each step, e_i^{rq} immediately precedes e_i^{rsp}, t is a *non-overlapping* communication trace.

A PoSM is well-formed if all its communication traces are well-formed.

Theorem 2.14 *If the syntactical definition of a PoSM P^A does not use explicit request and response* PortTransitions *(all* PortTransitions *are defined with the call transition and call state syntactical constructs) and each composite state in P^A may only exit with its completion event, then all traces from $LC(P^A)$ are well-formed. In addition, if P^A does not contain any orthogonal state, then all traces from $LC(P^A)$ are non-overlapping.*

Proof sketch: A call transition (and also a call state) always specifies a pair of transitions labeled with the request and response events; unless the call transition (a call state) is contained in a composite state with a labeled outgoing transition (that could cause the call to terminate without the response event), the call always completes. □

If not contained in an orthogonal state, events for different calls may not interleave in trace t, until t_i is empty, there is always an e_i^{rq}immediately followed by a e_i^{rsp}in t_i. □

Claim 2.15 *$LC(P^A)$ is a regular language.*

Proof sketch: A PoSM P^A can be transformed to a finite automaton. By following the structure of P^A, orthogonal regions may be replaced with Cartesian product of states; a composite state can be replaced with its substates, redirecting outgoing transitions to all substates (except those that already have a higher-priority transition) and redirecting the incoming transition to the substate targeted by transition from initial PseudoState. The generalized finite automaton (employing empty transitions). This way we yield a non-deterministic finite automaton, generating a regular language. □

3 Composition Verification with PoSMs

Behavior Protocols [17] provide a *behavior compliance* relation, which can be used to verify composition of components based on their behavior specifications. In this section, we first briefly review behavior compliance as it is defined in Behavior Protocols [17] and describe how behavior compliance can be used to address consistency issues in the composition of software components. Afterwards, we show how behavior compliance can be applied to PoSMs. Finally, we discuss how this can be used to address the consistency issues in composition of UML 2.0 components.

3.1 Behavior Compliance in Behavior Protocols

In behavior protocols, a single run of an agent A is captured as a sequence of atomic events (trace) from a finite domain S processed by A. Given a set of

labels *EventNames*, S is formed as $\{?, !, \tau\} \times \mathit{EventNames} \times \{\uparrow, \downarrow\}$ (here, τ denotes an event internally processed by A; the symbols ?, !, $\uparrow$ and $\downarrow$ stand for receive, send, request and response). Behavior of an agent A (denoted $L(A)$) is captured as the set of all traces of A, forming a language upon S.

Behavior of A may be described with a behavior protocol Prot^A, an expression syntactically generating a set of traces over S^* (denoted $L(\mathit{Prot}^A)$, conveniently a regular language). Employing a regular expression-like notation, behavior is described using event tokens for events from S and the following operators (given in priority order): `*` (repetition), `;` (sequencing), `+` (alternative), | (parallelism, based on arbitrary interleaving of traces) and $\|$ (parallel-or, $A\|B$ is a shortcut for $A + B + A|B$). Further, the composed operators are *composition* ($\sqcap_X$), *adjustment* ($|_X|$) and *consent* ($\bigtriangledown_X$). The notation also uses the shortcuts discussed in Sect. 2.3 and parentheses.

Example 3.1 The behavior described in the PoSM notation in Figs. 1 and 3 may be expressed in the behavior protocols notation as:

```
?Clerk.deposit{ !DB.addBalance{ ?DBNotify.newBalance }
                | !Log.logDeposit }
```

The communication trace demonstrated in example 2.12, capturing this behavior, is also a trace of this behavior protocol.

Composition $\mathit{Prot}^A \sqcap_X \mathit{Prot}^B$ yields the behavior resulting when agents A and B described by protocols Prot^A and Prot^B are composed together; X is the set of event labels from $\mathit{Events} = \mathit{EventNames} \times \{\uparrow, \downarrow\}$ of events transmitted between A and B. For each pair of traces $\alpha \in L(\mathit{Prot}^A), \beta \in L(\mathit{Prot}^B)$, the events from α and β arbitrarily interleave. In each such resulting trace, all events with label $x \in X$ are processed the following way: sequences of form $?x\ !x$ or $!x\ ?x$ are replaced by τx (an internal event); a trace containing events with label x that cannot be processed this way (unmatched !/?) is discarded from the result of the composition operator.

The adjustment operator also interleaves pairs of traces $\alpha \in L(\mathit{Prot}^A), \beta \in L(\mathit{Prot}^B)$, but exact match (not ? / ! correspondence) of events with label from X is required and only pairs α, β that match on events from X are included in the resulting behavior.

The consent operator (introduced in [1,2]) is similar to the composition operator, but generates *erroneous* traces for situations when interaction of A and B results into an error. The types of errors considered are *BadActivity* (A emits a but B is not ready to absorb a), *NoActivity* (similar to a deadlock situation) and *Divergence* (interaction of A and B never stops). The consent operator implicitly provides a relation for checking the composition of A and B, by considering the composition to be correct if A $\bigtriangledown_X$ B contains no erroneous traces.

Definition 3.2 We assume the set S is divided into disjoint sets S_{prov} (inputs,

events on provided interfaces) and S_{req} (outputs, events on required interfaces). Behavior $L(A)$ of agent A is *compliant* with behavior $L(Prot^A)$ of protocol $Prot^A$ on set S if (i) A can accept any sequence of inputs dictated by $Prot^A$ and (ii) for such inputs, A creates only outputs anticipated by $Prot^A$.

A formal definition is provided via the adjustment operator: $L(A)$ is compliant with $L(Prot^A)$ on set S iff:

(i) $L(Prot^A)/S_{prov} \subseteq L(A)/S_{prov}$

and

(ii) $\left(L(A)/S \mid_{S_{prov}} \mid L(Prot^A)/S_{prov}\right) \subseteq L(Prot^A)/S$.

where / is the operator for restriction.

By adjusting $L(A)/S$ with $L(Prot^A)/S_{prov}$ (the dictated inputs) over S_{prov}, only traces from $L(A)/S$ with inputs dictated by $L(Prot^A)$ are considered; these traces must be contained in $L(Prot^A/S)$. For reference, the original definition of behavior compliance is available in [17]. The case study in Sect. 4 provides demonstrations of the behavioral compliance relation.

3.2 Composition Verification with Behavior Compliance

In [14], we identified the consistency issues to be considered in component composition in a hierarchical component model. Basically, the issues are: (a) whether the *composed behavior* of components $\mathrm{A_1..A_n}$ forming together component S is compliant with the behavior specification for S; (b) whether two distinct specifications for a component specify “compatible” behavior; (c) and whether communication between A and B is correct.

In behavior protocols, the issue (a) is addressed by the compliance relation (employing the composition operator to obtain the composed behavior). The compliance relation may be also used to address the issue (b). Finally, the issue (c) is addressed by the consent operator.

Note that a verifier tool [18] is available to test the compliance relation (supporting the composition operator); thus, the issues (a) and (b) are decidable in behavior protocols. Enhancing the verifier tool to support the consent operator is subject of future research.

3.3 Behavior Compliance in PoSMs

The behavior protocols compliance relation is defined on languages (upon the domain of communication events) and thus, its definition is applicable to PoSMs as well. The set CE (domain of communication events) can be used in place of the set S. The set S_{prov} is the set of events on provided interfaces of the Port the PoSM is associated with, S_{req} is the set of events on required interfaces.

Although composition and consent are protocol operators, their semantics is defined solely based on the languages generated by their operands and thus,

their definition can be extended to communication languages of PoSMs.

Therefore, the consistency issues (a), (b) and (c) can be addressed for PoSMs; the existing behavior protocols compliance verifier may be employed to evaluate the compliance relation on PoSMs.

Note that the compliance relation is applied only to communication languages generated by PoSMs; neither the states, nor the structure of the state machine are considered in the compliance relation. Broadening the definition of compliance relation to execution languages is subject of future research.

3.4 Relation of Behavior Protocols and Port State Machines

Both PoSMs and behavior protocols describe behavior in a way that yields a set of communication traces, conveniently a regular language. It is possible to transform a behavior protocol into a PoSM, i.e., construct a PoSM generating the same communication language as the behavior protocol (restricted to the communication events contained in S).

In this process, (i) an event token explicitly specifying a request ($\uparrow$) or a response ($\downarrow$) is translated into an explicit PortTransition, (ii) a shortcut $?a$ or $!a$ is translated into a call transition, (iii) shortcut $?a\{Prot\}$ or $!a\{Prot\}$ is translated into a call state; the protocol *Port* is transformed into the internal behavior of the call state. A protocol may also specify internal events (τ), which do not influence the communication described by the protocol and are neither considered in the compliance relation; we omit them in the transformation. Following the syntactic structure of the protocol, we translate the sequencing operator (;) into sequenced states, repetition (*) into a loop transition, alternative (+) into multiple outgoing transitions; parallelism (|) is modeled via orthogonal regions.

Note that in this process, we create states as necessary to transform the structure of the protocol into a state machine. In the PoSM shown in Figs. 1 and 3, for selected states, names are provided to make the PoSM specification more expressive. In an automated process, anonymous states (without a name) have to be used instead. Here, automatically generated state labels may be employed to distinguish states in execution traces (in a way similar to how the states *ab1* or *logfin* are labeled).

In a similar vein, we may consider constructing a behavior protocol for a Port State Machine. However, in the general case, the only solution is to first transform the state machine into a regular automaton (expanding composite states) and afterwards, apply the generic algorithm for transforming a regular automaton into a regular expression. Such process would significantly impair readability of the resulting behavior protocol.

There may be interesting special cases, namely, when the only composite states used in the state machine are call states. Here, the transformation can be done separately at each level of nesting; exploring these special cases is subject of future research.

4 Case Study: Compliance of Port State Machine

The definition of the behavior compliance relation is based on the notion of *substitutability* [17]; in the SOFA Component model [15], behavior compliance is used to verify composition of software components. Given the specification of behavior of a component in the form of a *frame protocol*, the key question is, whether behavior of the realization of the component, as described by its *architecture protocol*, is compliant with the *frame protocol*.

This may be also applied to verify composition of UML 2.0 components; with PoSMs, we may reason on compliance of a realization described by a PoSM P_i^R with the specification described by PoSM P^S.

Let us consider the behavior specified by PoSM in Fig. 3 as the specification PoSM P^S. This PoSM specifies that while a call `?Clerk.deposit` is being processed, a call `!Log.logDeposit` is issued in parallel with issuing a call to `!DB.addBalance`, during which a call `?DBNotify.newBalance` is received. Note that here, "in parallel" means arbitrary interleaving of the traces generated by the orthogonal regions of the PoSM.

Example 4.1 Figure 5 shows PoSM specifications of three possible realizations of this specification. In Fig. 5 (a), PoSM P^{R_1} specifies that the call `!Log.logDeposit` occurs after the call `!DB.addBalance` completes. Such realization is (trivially) compliant with the specification, its behavior restricted to S_{prov} contains all traces from $LC(P^S)/S_{prov}$ (condition (i) of 3.2) and $LC(P^{R_1}) \subseteq LC(P^S)$, thus condition (ii) holds as well.

Example 4.2 The PoSM P^{R_2} in Fig. 5 (b) in addition specifies that the call `!Log.logDeposit` may occur zero or more times (instead of exactly once). Consequently, its language $LC(P^{R_2})$ is not compliant with $LC(P^S)$ – although condition (i) of 3.2 holds, condition (ii) does not: $LC(P^{R_2})$ contains traces capturing an arbitrary number of `!Log.logDeposit` calls, while $LC(P^S)$ contains only traces where the call `!Log.logDeposit` occurs exactly once.

Example 4.3 The PoSM P^{R_3} in Fig. 5 (c) instead specifies that the call `?DBNotify.newBalance` may be processed zero or more times; and that the call `!Log.logDeposit` will be issued while processing `?DBNotify.newBalance`, each time this call is received. Surprisingly, $LC(P^{R_3})$ is compliant with the $LC(P^S)$. Although the P^{R_3} can call `!Log.logDeposit` more than once (or not at all), this may occur only in runs where `?DBNotify.newBalance` is called more than once (or not at all). Thus, after reducing (via the adjustment operator) $LC(P^{R_3})$ to traces with inputs contained in $LC(P^S)$ (i.e., traces where `?DBNotify.newBalance` is called exactly once), the resulting behavior calls `!Log.logDeposit` exactly once (in an order permitted by the P^S) and therefore, condition (ii) of 3.2 holds; condition (i) holds trivially.

As the last example demonstrates, behavior compliance permits that behavior of a realization contains traces with outputs not expected by the specification, in case such traces result from inputs not permitted by the specifi-

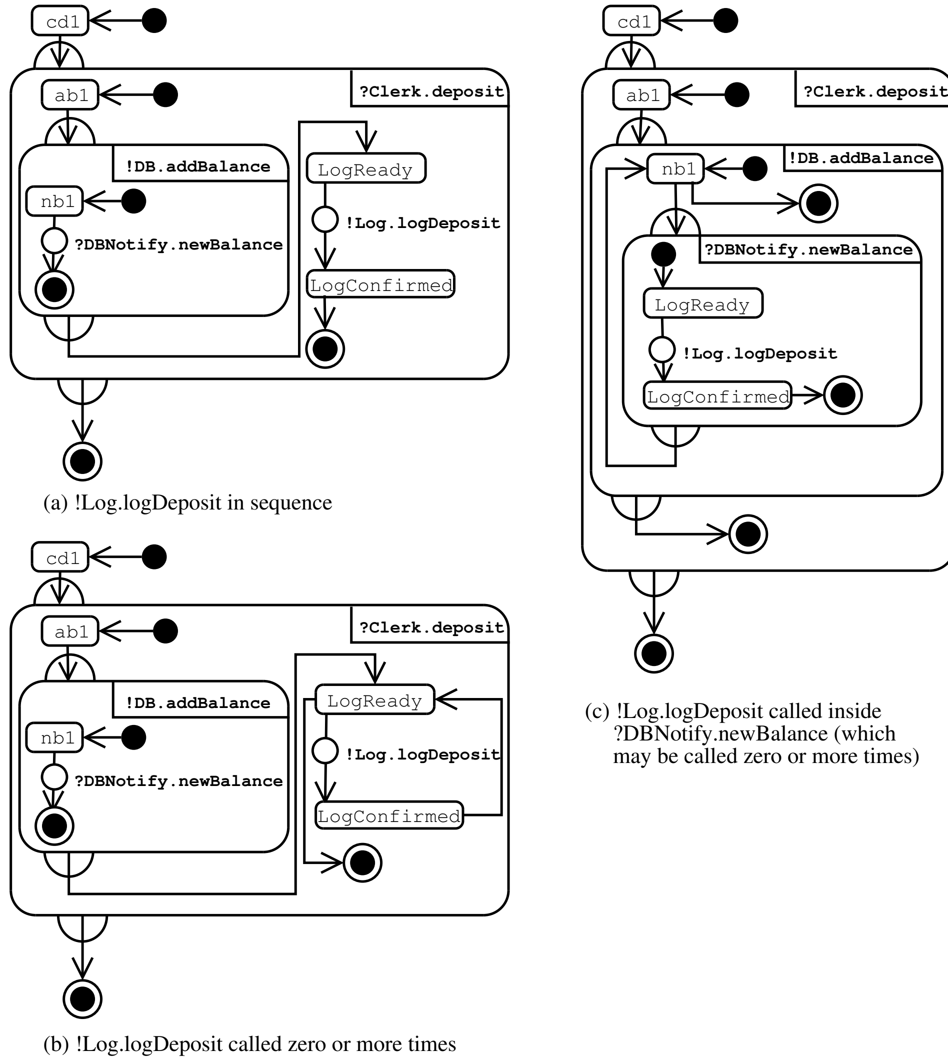

Fig. 5. Port State Machines describing possible realizations of the behavior specified by Fig. 3

cation. Consequently, substitutability based on behavior compliance permits a broader set of realizations to be used for a given component specification.

5 Evaluation and Related Work

Port State Machines permit to capture the interleaving of events (representing operation calls) on a set of provided and required interfaces associated with a Port of a UML 2.0 Component. PoSMs support modeling nested calls; technically, an arbitrary fixed depth of recursion can be modeled with a PoSM. Unlimited recursion (which inherently causes the generated language not to be regular) is avoided.

Conveniently, the language generated by a PoSM is regular (taking into account that there are no constraints, no event deferring and, inherently to

state machines, no recursion). Thus, PoSMs permit to establish a compliance relation and apply the behavior protocols compliance verifier [18].

The UML 2.0 Interactions (former sequence diagrams) also explicitly capture an operation call with atomic request and response events; also, trace model semantics is defined for Interactions. However, Interactions focus on describing communication among interconnected objects. Although it is possible to employ a formalGate to capture calls of an Operation exposed via an Interface of a Port, the notation does not support efficiently describing the ordering of communication on a Port.

The work presented in [11] defines an equivalence relation for state-chart specifications; bisimulation of labeled transition systems is used for testing the equivalence. In [10], the authors translate UML statecharts into PROMELA, the input language of SPIN. In a way similar to our approach, a subset of statecharts is chosen such that the statechart can be translated to a finite state automaton. However, call nesting is not considered in this approach.

In [20], two algorithms for testing conformance of LTS and behavior expressions are presented. The approach employs test cases, testing is done via synchronous parallel execution of a test case and the implementation. The test cases considered are deterministic, but the implementation may behave nondeterministically; thus, as an implementation passes a test case only if all possible runs pass, theoretically, a test case may have to be executed infinitely many times.

Method State Machines (MSMs) introduced in [19] extend state machines with the ability to model recursion. Recognizing the obstacles of the *run-to-completion* semantics, the authors model operation calls with two events, corresponding to request and response. A relation of compliance of a Protocol State Machine with a set of MSMs is defined; however, as a tradeoff for modeling recursion, the relation is not decidable. Moreover, the approach taken there is object-based, focused on the graph of operation calls among cooperating objects; it would not be possible to capture external communication on the interfaces of a software component with MSMs without a significant modification.

Use Case Maps [3,4] is a notation for visually expressing how a *scenario* (a particular run of a task to be completed by a system) traverses a component hierarchy. Thus, for a component, use case maps show the nesting of calls in a scenario. However, as use case maps are focused on individual scenarios, obtaining the “whole picture” of behavior on the interfaces of a component is not possible.

The *Rigorous Software Development Approach* coined in [21] considers generating a state machine from a sequence diagram with the aim to check for consistency and aid with generating code. However, neither composition, nor assembly is addressed here.

An abstract state machine language is employed in [7]; instead on reasoning on behavior compliance, the authors aim to generate test scenarios from the

abstract state machine specification; selecting test sequences is also considered in [9].

In [22], Message Sequence Charts (MSC) are translated into a labeled transition system (LTS) in order to facilitate model checking. A synthesis and analysis algorithm is provided; however, as the approach is focused on individual messages rather than on operation calls, call nesting is not addressed here.

6 Future Work

In our future work, we will use the OCL language to formally capture the compliance relation in the UML metamodel.

Moreover, we aim to propose a restricted constraint language, that would not break the regularity of the language generated by a PoSM, yet provide convenient modeling power. We consider developing a simple constraint language utilizing only the current state of the state machine (using an **in(state)** predicate to query orthogonal regions of the state machine); such a constraint language should fulfill the expectations: the language generated by a PoSM would remain regular, while the perceived expressive power of specifications would significantly increase.

With the aim to employ PoSMs to model use cases, our future goal is to further investigate operations for assembling behavior scattered in multiple PoSMs into a single PoSM. Currently, composition is defined only for communication languages of PoSMs, yielding the composed behavior as a language. We aim to explore composition of PoSMs at structural level, with the goal to construct a PoSM representing the composed behavior. Moreover, broadening the definition of behavior compliance to include also state events (for entering/exiting a state) remains a challenge.

To obtain a proof-of-the-concept, we aim to include the proposed UML extensions in a UML Profile implemented for a UML tool, providing support for PoSMs and employing the behavior compliance verifier tool [18] already available for behavior protocols.

7 Conclusion

In this paper, we proposed the Port State Machines (PoSMs). Building on UML 2.0 [10] Protocol State Machines and Behavior Protocols [17], Port State Machines allow to capture the interleaving of operation calls on a set of provided and required interfaces. Operation calls are captured as a pair of atomic events representing the start of the call (request) and end of the call (response). This way, nesting of operation calls (e.g., a call-back) can be captured in a specification.

Moreover, as PoSMs use atomic events, the behavior on a Port specified by a PoSM is captured as a set of traces, forming a language upon a finite

alphabet. The behavior compliance relation has been established to reason on compatibility of PoSM specifications. As the language of a PoSM is regular, the compliance relation is decidable; conveniently, an already existing verification tool [18] can be employed for this task.

Acknowledgments

A special thank goes to Jiří Adámek for his very helpful advices.

References

[1] Adamek, J., and F. Plasil. Behavior Protocols Capturing Errors and Updates. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, ETAPS, Warsaw, 2003

[2] Adamek, J. and F. Plasil. Behavior Protocols: Tolerating Faulty Architectures and Supporting Dynamic Updates. *Technical Report 02/10*, Dept. of Computer Science, University of New Hampshire, Durham, NH, U.S.A., Oct 2002

[3] Amyot, D., and G. Mussbacher. On the Extension of UML with Use Case Maps Concepts. In *Proceedings of UML 2000*, York, UK, October 2-6, 2000, LNCS 1939, Springer 2000

[4] Buhr, R.J.A.: *Use Case Maps as Architectural Entities for Complex Systems*, Transactions on Software Engineering, IEEE, vol 24, no **12** (1998)

[5] D'Souza, D. "Components with Catalysis," http://www.catalysis.org/, 2001

[6] Graham, I. "Object-Oriented Methods: Principles and Practice," Addison-Wesley Pub Co, ISBN: 020161913X, 3rd edition December 2000

[7] Grieskamp, W., and M. Lepper, W. Schulte, N. Tillmann. Testable Use Cases in the Abstract State Machine Language, In *Proceedings of APAQS'01*, December 10 – 11, 2001, Hong Kong

[8] Harel, D. *Statecharts: A visual formalism for complex systems*, Science of Computer Programming **8** (1987), pp. 231-274, Elsevier Science Publishers B.V. (North-Holland)

[9] Hong, H. S., and Y. G. Kim, S. D. Cha, D.-H. Bae, H. Ural. *A test sequence selection method for statecharts*, Software Testing, Verification & Reliability vol 10 no **4** (2000), pp. 203-227

[10] Latella, D., and I. Majzik, M., Massink. *Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker*, Formal Aspects of Computing vol 11 no **6** (1999), pp. 637-664

[11] Latella, D., and M. Massink. A Formal Testing Framework for UML Statechart Diagrams Behaviours: From Theory to Automatic Verification. In *Proceedings of HASE 2001*, IEEE Computer Society (2001), pp. 11-22

[12] OMG: Unified Modeling Language (UML), version 1.5, formal/2003-03-01, http://www.omg.org/uml/

[13] OMG: Unified Modeling Language: Superstructure, version 2.0, Final Adopted specification, ptc/03-08-02, http://www.omg.org/uml/

[14] Plasil, F., and V. Mencl. Getting "Whole Picture" Behavior in a Use Case Model, in Transactions of the SDPS: Journal of Integrated Design and Process Science, vol. 7, no. 4, pp. 63-79, Dec 2003, ISSN-1092-0617

[15] Plasil, F., and D. Balek, R. Janecek. SOFA/DCUP Architecture for Component Trading and Dynamic Updating, In *Proceedings of ICCDS '98*, Annapolis, IEEE Computer Soc. (1998)

[16] Plasil, F., and S. Visnovsky, M. Besta. Bounding Behavior via Protocols, In *Proceedings of TOOLS USA '99*, Santa Barbara, CA, Aug. 1999.

[17] Plasil F., and S. Visnovsky. *Behavior Protocols for Software Components*. Transactions on Software Engineering, IEEE, vol 28, no **11** (2002)

[18] Mach, M. *Model Checking of Behavior Protocols*, Master Thesis, Charles University in Prague, Sep 2003, Advisor: Frantisek Plasil, available at http://nenya.ms.mff.cuni.cz/

[19] Tenzer, J., and P. Stevens. Modelling recursive calls with UML state diagrams. In *Proceedings of FASE 2003 (part of ETAPS 2003)*, Warsaw, Poland, April 7-11, 2003, LNCS 2621, Springer

[20] Tretmans, J. *Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation*. Computer Networks and ISDN Systems, vol. 29 no **1**, pp. 49-79 (1996)

[21] Zuendorf, A.. From Use Cases to Code – Rigorous Software Development with UML, Tutorial T4, ICSE 2001: May 12-19, 2001, Toronto, Ontario, Canada

[22] Uchitel, S., and J. Kramer. A Workbench for Synthesising Behaviour Models from Scenarios, In *Proceedings of ICSE 2001*, 12-19 May 2001, Toronto, Ontario, Canada