Enhancing Component Behavior Specifications with Port State Machines

[Technical Report 2003/4]

Vladimir Mencl

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, http://nenya.ms.mff.cuni.cz,
phone: +420 2 2191 4267, fax: +420 2 2191 4323

Abstract

Protocol State Machines (PSM) in UML 2.0 [13] describe valid sequences of operation calls. To support modeling components, UML 2.0 introduces structured classifiers, featuring Ports associated with provided and required interfaces. Unfortunately, PSMs are 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 the relation is not possible in general; thus reasoning on consistency in component composition is not possible with PSMs.

Behavior Protocols [18] capture the behavior of a component via a set of traces (sequences of atomic events), forming a language upon a finite alphabet. A textual notation similar to regular expressions is provided to approximate the behavior with a regular language. In [1, 18], the *compliance* and *consent* relations are defined to reason on consistency of component composition; a verifier tool [19] is available for the compliance relation.

In this paper, we propose Port State Machines (PoSM) to capture the interleaving and nesting of operation calls on a Port. Building on our experience with behavior protocols [18], we introduce notation shortcuts to conveniently capture an operation call as two atomic events *request* and *response*; moreover, the notation also explicitly distinguishes events on provided and required interfaces.

We demonstrate how communication on a Port can be modeled with a PoSM in a way that yields a regular language. Based on this, we apply the compliance relation of behavior protocols to PoSM, allowing us to reason on behavior compliance of components in software architectures; the existing verifier tool can be applied to PoSMs.

This work was partially supported by the Grant Agency of the Czech Republic project 102/03/0672.

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 internal structure and featuring Ports associated with provided and required interfaces. Based on these concepts, the Component metaclass is defined, providing a possibly hierarchical component model, with external communication of the component encapsulated in the component's Ports.

In component-based software engineering, a basis for reasoning on behavioral compliance is highly desirable in order to validate software architectures and to reason on component "compatibility".

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 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, reasoning on soundness of 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 checking behavioral compliance. The observable behavior of a component is typically captured as communication on its *provided* and *required* interfaces [4, 5, 6, 16]. 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. However, as a consequence, only one "direction of communication" can be captured with a PSM. A PSM can capture communication only on a single Classifier, typically an Interface. The concrete usage of the interface in a Port determines whether the events captured by the PSM are received (for provided) or sent (for a required interface).

UML State Machines employ the *run-to-completion* semantics, i.e., only after a transition of the 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 nested calls (e.g., a call-back or statically limited recursion), and neither 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 received completes and thus, the same restrictions on nested calls apply to PSMs. Consequently, although a PSM specifies a sequence of operation calls, this sequence cannot be properly reflected as a *trace* for further behavioral reasoning, due to the non-atomicity of the events (operation-call) in the sequence. Moreover, the sequence cannot capture nesting of calls, although this is a common pattern in component communication.

Establishing a decidable compliance relations upon PSMs is unfortunately not feasible, as such a compliance relation (upon the languages generated by the PSMs) would be undecidable for the following reasons: (i) 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); thus the constraint language may be of arbitrary power. (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 bottom line 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 [18]; 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 [18], our research group developed Behavior Protocols, modeling behavior of *agents* as traces of atomic events. Applied to the SOFA component model [16], 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 [19] for checking this relation is available. The SOFA hierarchical component model and UML 2.0 Components build on similar concepts.

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 as in 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 identifiers in the UML metamodel (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 an received request event and an sent response event,

while an operation call issued on a required interface is represented by an sent request event and an received response event. To hide such technical details from the modeler, PoSM notation defines convenient shortcuts.

2.1. PortStateMachine and PortTransition metaclasses

We use the UML 2.0 extension mechanisms; technically, PoSM is situated in the UML metamodel as the PortStateMachine metaclass (subclassing ProtocolStateMachine); a transition in a PoSM is a PortTransition (subclassing ProtocolTransition). A Port Transition features two attributes: Communication-Direction, capturing whether the event specified by its trigger is received or sent and OperationCallPart, capturing whether the transition represents the request or response part of the operation call. A Port Transition must have exactly one trigger; the trigger must be a CallTrigger, referring to an operation on an Interface of the Port the PoSM is associated with. Figure 1 shows the metaclasses described above and their relation to the UML metamodel.

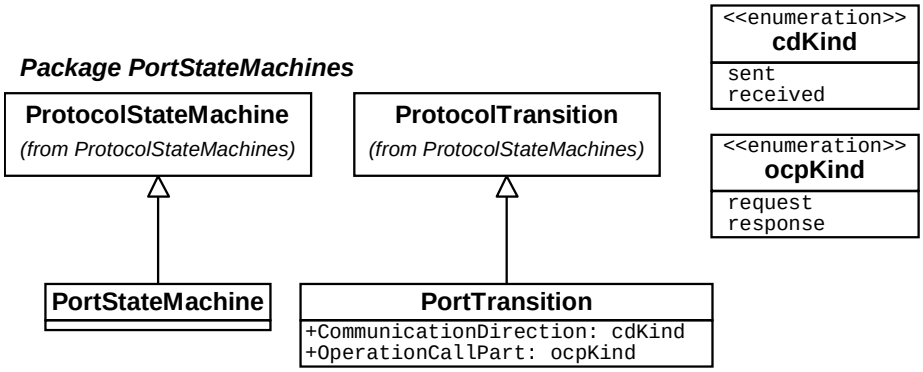

Figure 1: Port State Machines: abstract syntax (metamodel extension)

Note that compared to Protocol State Machines, a single PSM transition is represented with two transitions in a PoSM; thus an intermediate state has to be employed in between the transitions.

2.2. Port State Machine meta-model constraints

In order to provide a basis for a decidable compliance relation, we impose additional constraints on the Port State Machines and Port Transitions. The deferrableTrigger association of each state in a PoSM must be empty, so that no event deferring may occur. Currently, we do not support constraints in PoSMs; thus, a transition in a PoSM may not specify any guards. Transitions other than PortTransition are permitted in a PoSM; however, such transitions may not specify any triggers, i.e., they can only accept the *completion event.*

2.3. Notation

The PoSM notation utilizes the notation of Behavior Protocols [17, 18]. 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 atomic events, labeled with event tokens where the operation name has either the suffix ↑ for request or ↓ for response. E.g., sequence *?op*↑ **;** *!op*↓(**;** is the operator for sequencing) models receiving call of the operation *op* as receiving a request for *op* and sending a response. Here, the shortcuts *?op*, *!op*, and *?op{Prot}* stand for sequences *?op*↑ **;** *!op*↓, *!op*↑ **;** *?op*↓ and *?op*↑ **;** *Prot* **;** *!op*↓ 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. The notation is demonstrated in Fig. 2.

Figure 2 a) shows the sequence *?a*↑; *!a*↓ with an explicitly modeled (though anonymous) intermediate state. For convenience, Fig. 2 b) employs a shortcut to model the same sequence. The arrow actually represents two transitions; the circle on the transition indicates the existence of the implicit intermediate state. Here, only "**?a**" is used; the shortcut is semantically equivalent to the two transitions explicitly modeled in Fig. 2 a). 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.

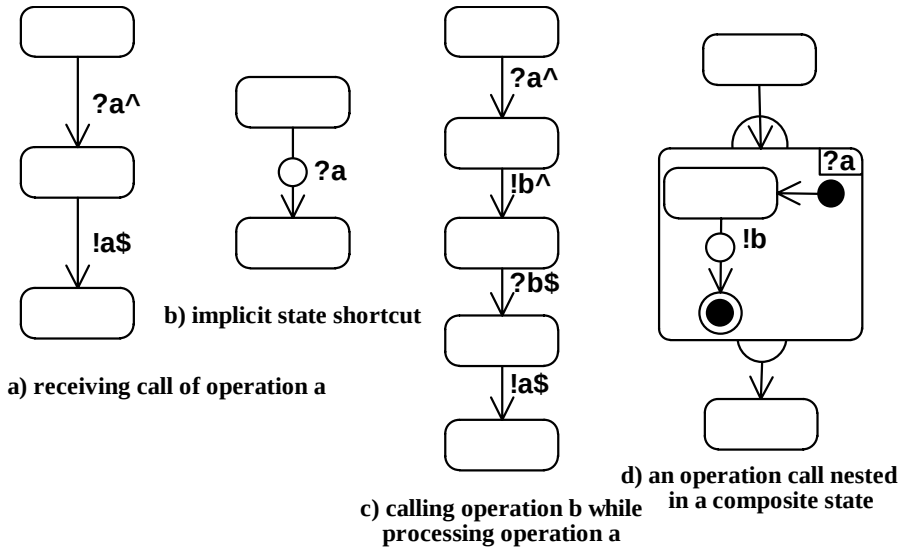

Figure 2: Port State Machines notation

Nested calls can be modeled with PoSMs; in Fig. 2 c), operation *b* is called while the call of operation *a* is being processed. This can be conveniently captured via a shortcut

employing a composite state (Fig. 2 d); the composite state roughly corresponds to the intermediate state used in Fig 2 b), only the label is attached to the state instead to the transition. To reflect that the composite state substitutes the intermediate state, semicircles are attached to the connections of the transitions representing parts of the operation call with the state. Inside the composite state, call of operation *b* is modeled employing the PoSM notation.

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 example presented later, the character "." (dot) will be used to join these identifiers. Figure 3 shows a more elaborate example of a Port State Machine, this example is discussed in the case study in Sect. 4.

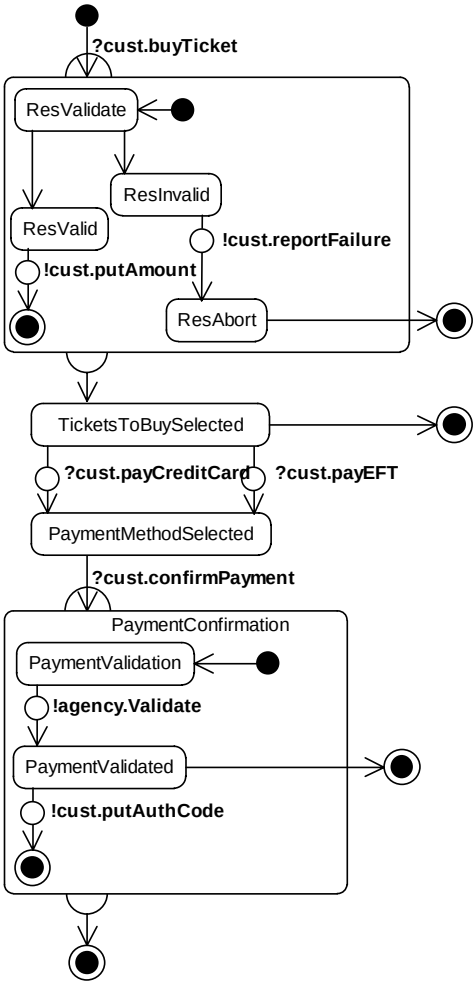

Figure 3: Port State Machine acquired from the Pro-case "Pay for a ticket"

3. Composition Verification for PoSMs based on Behavior Protocols

Behavior Protocols [18] provide a *behavior compliance* relation, which can be used to verify composition of components behavior descritions. In this section, we first briefly review behavior compliance as it is defined in behavior protocols, then we show how behavior compliance can be used to address consistency issues in the composition of software components. Later, we show how the behavior compliance definition can be applied to PoSMs and finally, we discuss how it can 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 ACT processed by A. Behavior of an agent A (denoted L(A)) is captured as the set of all traces of A, forming a language upon ACT.

Behavior of A may be described with a behavior protocol $Prot^A$, an expression syntactically generating a set of traces over ACT* (denoted $L(Prot^A)$, conveniently a regular language). Employing a regular expression-like notation, behavior is described using event tokens of events from ACT and the following operators (given in priority order): ***** (repetition), **;** (sequencing), **+** (alternative), | (parallelism, based on arbitrary interleaving of traces) and || (parallel-or, shortcut for A + B + A|B). Further, the composed operators are *composition* ($\sqcap_X$), *adjustment* ($|_X|$) and *consent* ($\triangledown_X$). The notation also uses the shortcuts discussed in Sect. 2.3 and parentheses. An example of a behavior protocol is available in Fig. 4; the example will be discussed and related to PoSM in the case study in Sect. 4.

Composition $A \sqcap_X B$ yields the behavior resulting when agents described by protocols A and B are composed together; X is the set of event tokens for events transmitted between these agents. Traces from L(A) and L(B) are arbitrarily interleaved, except for occurrences of *?x !x* (or *!x ?x*) in the resulting trace ($x \in X$), which are replaced by τx (an internal action).

The adjustment operator also interleaves traces from L(A) and L(B), but exact match (not ? / ! correspondence) of events from X is required and only pairs of traces 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 $A \triangledown_X B$ contains no erroneous traces.

As to the definition of behavior compliance, we say that L(A) is compliant with $L(Prot^A)$ on set $S \subseteq ACT$ (S divided into inputs S_{prov} and outputs S_{req}) if L(A) can respond to any sequence of inputs dictated by $L(Prot^A)$ and for such inputs, creates only outputs anticipated by $L(Prot^A)$; for the full definitions please refer to [15].

3.2. Composition Verification with Behavior Compliance

In [14], we identified the consistency issues to be considered in composition in a hierarchical component model. Basically, the issues are: (a) whether the *composed behavior* of components $A_1..A_n$ 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, used together with the composition operator. In a similar way, the issue (b) is addressed by the compliance relation. Finally, the issue (c) is addressed by the consent operator.

Note that a verifier tool [19] has been implemented testing the compliance relation and supporting the composition operator; thus, the issues (a) and (b) are decidable here. Enhancing the verifier tool to support the consent operator is subject of future research.

In case when behavior description is fragmented into several separate specifications, a prerequisite to addressing the issues listed above is handling the issue (d) whether a mechanism exists to assemble the whole picture behavior of A from the specification fragments (typically, this is the case in use case modeling). The behavior protocols operators provide a means to assemble the fragments into a single protocol.

3.3. Behavior Compliance in PoSMs

A PoSM specifies the events to be processed at either a provided or required interface of the Port specified by the PoSM.

The events processed in a single execution of a PoSM P^A can be captured in a trace and thus, by considering traces of all possible runs of P^A, P^A generates a language $L(P^A)$.

For brevity, we do not provide a formal definition here. Informally, we start by capturing the events processed in a single run-to-completion step s_i (following the UML 2.0 specification, and including all enabled transitions) in a sequence se_i; here events processed concurrently by orthogonal regions of the state machine arbitrarily interleave. Following the execution of P^A from the initial configuration to a final state, we form the trace t by choosing for each step s_i a sequencing se_i of the events processed in s_i; by concatenation we get a trace t of P^A. The set of all traces of P^A forms the language $L(P^A)$.

The behavior protocols compliance relation is defined on languages and thus can be also applied to languages generated by PoSMs.

Although composition and consent are protocol operators, they are defined only based on the languages generated by their arguments and thus, their definition can be extended to PoSMs. Thus, the consistency issues (a), (b) and (c) can be addressed for PoSMs; here the existing behavior protocols compliance verifier can be employed.

Note that the compliance relation is based only on the events generated by the PoSMs; neither the names of state, nor the structure of the state machine are considered in the compliance relation.

Behavior assembly can be achieved with PoSM by combining state machines using simple constructs modeling repetition, sequencing, alternative (via transitions) and parallelism (via orthogonal regions), addressing the issue (d).

4. Case Study: Applying PoSM to Use Case Modeling

In [14], we developed Generic UC View, a simple formal model for use cases, identifying criteria for suitable compliance relations. Evaluating that textual use cases do not permit reasoning on behavior compliance, we introduced Pro-cases, a notation for use cases based on behavior protocols [18]. Figure 4 shows a Pro-case; a fully fledged example of a Pro-case model is available in [15]. Pro-cases employ the behavior protocols notation as described in the previous section; the τ symbol indicates an internal action of the system described. In Fig. 4, **bold** font is used to show a typical walk-through of the Pro-case.

Port State Machines, being able to capture the communication of an entity (component) with entities (components) it is connected with, can be employed as a notation for use cases. The Pro-case demonstrated in Fig. 4 can be conveniently transformed to a PoSM. Figure 3 shows a PoSM modeling the same behavior (the left-most path from the top to the bottom corresponds to the typical walkthrough highlighted in the Pro-case); however, note that both the PoSM shown in Fig. 3 and the Pro-case shown in Fig. 4 were obtained independently from the original textual use case (not shown here). Transforming a Pro-case into a PoSM is possible in general; omitting internal actions, the transformation is straightforward: sequencing (**;**) translates into sequenced states; alternative (**+**) into multiple outgoing transitions, operation call nesting (expressed via **{}**) is reflected as nesting of composite states, parallelism (not demonstrated here) would be modeled via (concurrent) orthogonal regions. Note that in this process; states have to be created as necessary. In the PoSM shown in Fig. 3, state names were manually inserted to make the PoSM specification more expressive; in an automated process, anonymous states (without a name) might be used instead.

```
?cust.buyTicket { τValidateReservation ;
   ( !custCb.putAmount + !custCb.reportFailure )
} ;
(  ( ?cust.payCreditCard + ?cust.payEFT ) ;
   ?cust.confirmPayment {
      !agency.validatePayment ;
      ( τRecordPayment ;
        !custCb.putPaymentConfirmationCode +
        NULL )
   } + NULL
)
```

Figure 4: Pro-case “Pay for a ticket”

When a failure condition is detected, a use case typically ends (aborts). In a PoSM, this is captured as the transitions from within composite states to final states shown in Fig. 3; in the Pro-case example, this is modeled using the alternative (**+**) operator, with the special token NULL (empty protocol) optionally used for the failure branch. The event names in Fig. 3 use the names of interfaces separated by a dot “**.**”; `cust` is a provided interface, while `agency` and `custCb` are required interfaces.

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 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 machine) no recursion). Thus, PoSMs permit to establish a compliance relation and apply the behavior protocols compliance verifier.

The work presented in [11] addresses behavior verification of state-chart specifications by defining an equivalence based on bisimulation of labeled transition systems; 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.

Method State Machines (MSMs) introduced in [21] 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.

UseCaseMaps [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 shows 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 [22] considers generating a state machine from a sequence diagram; thus, contrary to our approach, transforming an event-based model to a state-based model.

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 [23], 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 investigate operations for assembling behavior scattered in multiple PoSMs into a single PoSM (assembling the "whole picture" behavior). Moreover, we aim to formally define the composition operator for PoSMs so that the composed behavior of multiple connected components (together realizing a compound component) can be checked for compliance with the specification for the compound component.

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 [19] 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 [18], 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 specified for a Port by a PoSM can be captured as a set of traces, i.e., a language upon a finite alphabet. As the language is regular, a compliance relation can be established to reason on compliance of PoSM specifications. Conveniently, an already existing verification tool [19] can be employed for this task. Composition of behavior of neighboring components is under investigation.

References

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

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

[3] Amyot, D., Mussbacher, G.: On the Extension of UML with Use Case Maps Concepts. UML 2000, York, UK, October 2-6, 2000, in Proceedings 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, Dec 1998

[5] D'Souza, D. Components with Catalysis, 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., Lepper, M., Schulte, W., Tillmann, N.: Testable Use Cases in the Abstract State Machine Language, 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., Kim, Y. G., Cha, S. D., Bae, D.-H., Ural, H.: A test sequence selection method for statecharts. Software Testing, Verification & Reliability 10(4): 203-227 (2000)

[10] Latella, D., Majzik, I., Massink, M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. Formal Aspects of Computing 11(6): 637-664 (1999)

[11] Latella, D., Massink, M.: A Formal Testing Framework for UML Statechart Diagrams Behaviours: From Theory to Automatic Verification. HASE 2001: 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., Mencl, V.: Getting "Whole Picture" Behavior in a Use Case Model, accepted to IDPT2003, Beijing, China, June 2003, available at http://nenya.ms.mff.cuni.cz/

[15] Plasil, F., Mencl, V.: Use Cases: Assembling "Whole Picture Behavior", TR 02/11, Dept. of Computer Science, University of New Hampshire, Durham, NH, U.S.A., Nov 2002

[16] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP Architecture for Component Trading and Dynamic Updating, Proceedings of the ICCDS '98, Annapolis, IEEE Computer Soc., 1998

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

[18] Plasil F., Visnovsky, S.: Behavior Protocols for Software Components. Transactions on Software Engineering, IEEE, vol 28, no 11, Nov 2002

[19] SOFA Behavior Protocol Verifier, the SOFA project, http://nenya.ms.mff.cuni.cz/projects/sofa/tools/

[20] Stevens, P.: On Use Cases and Their Relationships in the Unified Modelling Language in Proceedings, FASE 2001 (part of ETAPS 2001), Genova, Italy April 2-6, 2001, Springer LNCS 2029, ISBN 3-540-41863-6

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

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

[23] Uchitel, S., Kramer, J.: A Workbench for Synthesising Behaviour Models from Scenarios, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada