Specifying Component Behavior with Port State Machines

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 in UML 2.0 [10] combine state-based behavior specifications with event-based descriptions of valid sequences of operation calls. To support modeling components, UML 2.0 introduces structured classifiers, featuring Ports associated with provided and required interfaces. Unfortunately, Protocol State Machines are applicable only to a single interface, either a provided or required one. Moreover, the definition of *protocol conformance* is rather fuzzy and reasoning on the relation is not possible in general.

In this paper, we propose Port State Machines to capture the interleaving and nesting of operation calls on a Port. Building on our experience with behavior protocols [15], 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 Port State Machines in a way that yields a regular language, formed by the set of traces of atomic events on the Port. Thus, Port State Machines form a basis suitable for behavioral reasoning; establishing a formally specified compliance relation is subject of future research.

1. Introduction

1.1. UML 2.0: State Machines and Protocol State Machines

The Unified Modeling Language (UML) [9] features StateMachines, which allow to model behavior in a state-based way; nevertheless, the execution of a State Machine can be observed as the sequence of *events* accepted and *actions* executed. The upcoming new version of the standard, UML 2.0 [10], introduces the ProtocolStateMachine (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 its Ports.

In Component-Based Software Engineering, a basis for reasoning on behavioral compliance of software is highly desirable in order to validate software architectures.

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 may protocol conformance 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 for specifying component behavior as a basis for model checking. The observable behavior of components is typically captured as communication on its *provided* and *required* interfaces [13, 5, 6, 4]. However, in UML State Machines, significantly different mechanisms are employed to specify events received (typically operations on the provided interfaces), captured as *triggers* associated with transitions, and events sent (typically operations on the required interfaces), captured as various *actions* inside *activities* specifying the effect of a transition, or associated with entering/exiting a state or as the doActivity of the state. As the spectrum of actions is rather huge and it is not possible to establish a one-to-one correspondence between triggers and actions related to communication, it is not possible to reason

on *composed behavior* of multiple communicating components specified with State Machines.

Protocol State Machines (further PSM), a refinement of the (generic) *behavioral* State Machine, impose a restriction on its transitions, requiring that no activities are associated neither with transitions, nor with states. This reduces the specification mechanism from a mixture of Activities (Petri-net like constructs in principle) and State Machines to pure State Machines. However, as a consequence, only one "direction of communication" can be captured with a PSM. Typically, a PSM is associated with an Interface, the concrete usage of the interface in a Port (possibly of a Component) determines whether the events captured by the PSM are received (in case of a provided interface) or sent (in case of required interface). Thus, a PSM can only capture the communication on a single interface.

UML State Machines employ the *run-to-completion* semantics, i.e., only after a transition of the State Machine completes may 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. Therefore, not only do not StateMachines support (unlimited) recursion, but not even nested calls (e.g., a simple call-back – single-level indirect recursion).

Surprisingly, the situation is no easier in PSMs – although no activities modeling the method are captured 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 recursion) 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. Furthermore, the sequence cannot capture nesting of calls (as nested calls are not supported by PSMs).

Being "close" to a regular automaton, PSMs evoke the idea to employ a tool to decide on compliance ("compatibility of behavior") of two PSMs. Unfortunately, that is not feasible, as such a compliance (basically, a relation upon the languages generated by the automata) would be undecidable for the following reasons: (i) Constraint language used for guards of transitions is arbitrary (thus of arbitrary power, though "without side-effects") (ii) Events may be deferred and processed later, thus the automaton gets a stack (though without clear semantics of the order of retrieval; thus rather resembling 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 for any non-trivial case. A compliance relation is typically defined on regular languages, e.g., a decidable relation is defined in [15]; the work on the consent operator [1] provides an alternative approach [2].

The widely recognized, state oriented, State-chart notation [8] (basis of UML State Machines) is intuitive to modelers; while the derived sequences of event are helpful to developers. Moreover, in case a trace model can be defined for the sequences of events (i.e., the events are atomic), reasoning on compliance may be done.

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.

Thus, our motivations are: (i) State Machines in UML do not properly capture interleaving of sent and received events. (ii) The form State Machines use does not permit establishing a decidable compliance relation. (iii) A State-based notation is intuitive to modelers. (iv) Traces of atomic events can be employed in formal reasoning. (v) A specification mechanism is missing to capture the communication on a Port.

1.3. Goals and Structure of the Paper

In [15], we developed Behavior Protocols, modeling behavior of agents as traces of atomic events. Applied to the SOFA component model [13], behavior protocols capture the interleaving of operation calls sent and received by a SOFA component. Nested calls can be captured here. Moreover a decidable compliance relation is defined; a verifier tool [16] for SOFA components is available.

A correspondence can be established between the SOFA hierarchical component model and UML 2.0 Components based on Structured Classifiers. Considering the motivations discussed in Sect. 1.2, we propose *Port State Machine* (PoSM) with the following goals: (1) Provide a state-based notation (2) that allows to capture interleaving of events sent and received (by a Port of a Component) (3) and nested calls (4) in such a way that the behavior can be captured as a trace of atomic events. (5) Moreover, a compliance relation should be possible to define.

This paper is structured as follows: Sect. 2 introduces the Port State Machines (PoSMs); a case study follows in Sect. 3. Sections 4 and 5 evaluate the contribution, discuss Related work and line out Future Work; the paper concludes in Sect. 6.

1.4. Note on conventions used

In this paper, PSM stands for Protocol State Machines (introduced by UML 2.0), while PoSM (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

Port State Machines build 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* and *response*. Moreover, in order to capture the interleaving of operation calls sent and received (via different interfaces of a Port), PoSMs explicitly distinguish between *sent* and *received* events (calls). To hide such technical details from the modeler, PoSM notation defines convenient shortcuts.

2.1. PortStateMachine and PortTransition metaclasses

Technically, a PoSM is captured in the model as the PortStateMachine metaclass (subclassing ProtocolStateMachine); a transition in a PoSM is a PortTransition (subclassing ProtocolTransition). A Port Transition features two attributes: CommunicationDirection, 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 relates the PoSMs to the UML metamodel.

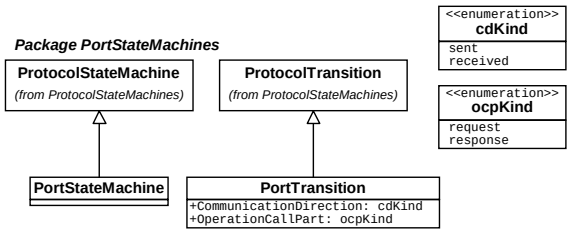

Figure 1: Port State Machines: abstract syntax (metamodel extension)

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 to join the transitions.

2.2. Port State Machine semantics

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. Constraints are not supported in PoSMs. For convenience, a constraint may be specified as the guard of a transition; however, such a constraint merely plays the role of a label or a comment and is not considered in evaluating the set of traces of events generated by the PoSM. (Technically, this can be achieved by a special constraint language that evaluates to *true* for any condition). 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 (and shortcuts)

The PoSM notation utilizes the notation of Behavior Protocols [15, 14]. There, *?a* stands for receiving event *a* and *!a* for sending event *a*. A call of an operation *a* is captured with atomic events, where the event name *a* has either the suffix ↑ for request or ↓ for response. E.g., sequence *?a*↑; *!a*↓(receiving request *a* and sending response *a*) models receiving call of operation *a*; here a shortcut *?a* can be employed. In a similar vein, shortcut *!a* stands for *!a*↑ ; *?a*↓. Moreover, the shortcut *a{Prot}* stands for *?a*↑; Prot ; *!a*↓.

The notation for PoSMs employs these prefixes (*?*/*!*) and suffixes (↑/↓) to express the attributes of PortTransition in the event label. Due to the limitations of the character set available, we represent ↑ with **^** and ↓ with **$** respectively. The notation is demonstrated in Fig. 2.

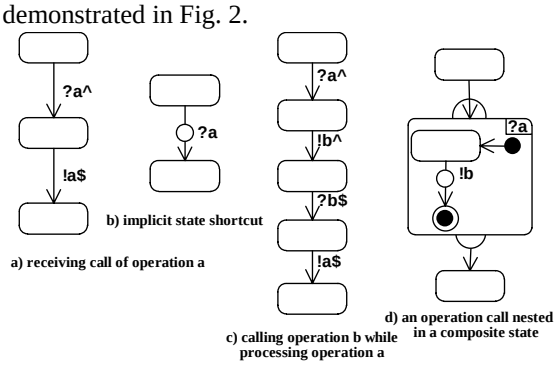

Figure 2: Port State Machines notation

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.

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 following section, the character "." (dot) will be used to join these identifiers.

3. Case Study: Applying PoSMs to Use Case Modeling

In [11], 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 [15]. Figure 3 shows a Pro-case; a fully fledged example is available in [12]. In Pro-cases, events use the same notation as explained in Sect. 2.3; the τ symbol indicates an internal action, either an action to be performed internally by the system (component), or it may represent a condition. The operators used here **;** and **+** represent sequencing (concatenation) and alternative, respectively. In Fig. 3, **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. 3 can be conveniently transformed to a PoSM; Figure 4 shows the same behavior modeled as a PoSM. Omitting internal actions and transforming conditions into guards, the transformation is straightfoward. Sequencing (**;**) translates into sequenced states; alternative (**+**) into multiple outgoing transitions, operation call nesting (expressed via **{}**) is reflected as nesting of composite states. Pro-cases also support parallelism (not demonstrated here); this would be modeled via (concurrent) orthogonal regions.

?custtr.buyTicket { **τValidateReservation** ;
(**!custtr.putAmount** + τInvalidReservation ;
!custtr.reportFailure)
} ;
((**?custtr.payCreditCard** + τSelectPayEFT ;
?custtr.payEFT + τInvalidReservation
) ;
?custtr.confirmPayment {
!agency.validatePayment ;
(**τRecordPayment ;**
!custtr.putPaymentConfirmationCode +
τInvalidPayment)
} + τAbortBuyTicket + τInvalidReservation
)

Figure 3: Pro-case "Pay for a ticket"

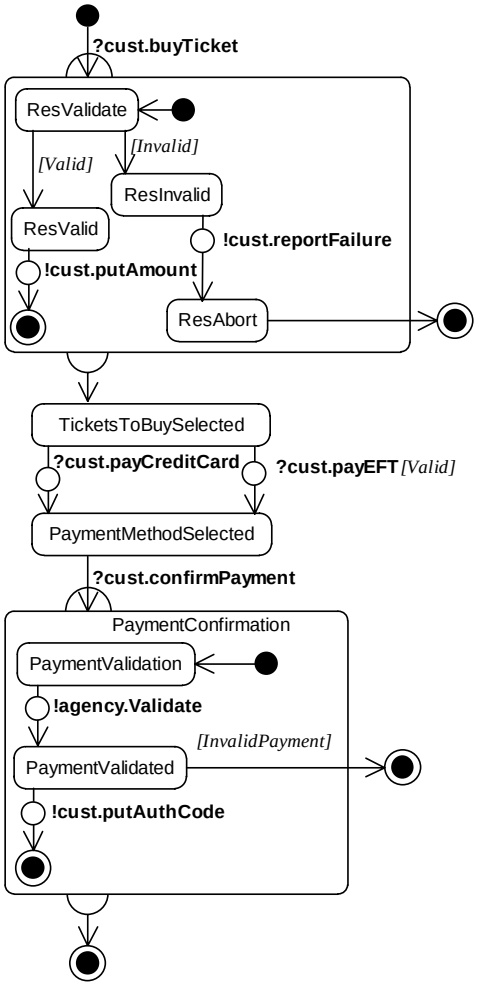

Figure 4: Port State Machine acquired from the Pro-case "Pay for a ticket"

4. 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 component Port. PoSMs support modeling nested calls; theoretically, arbitrary fixed depth of recursion can be modeled with a PoSM. Unlimited recursion (inherently causing 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 form a good basis for reasoning on behavior compliance; establishing a compliance relation and implementing a verifier tool is subject of future work.

Method State Machines (MSMs) introduced in [18] 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 here is object-based, focused on the graph of operation calls among cooperating objects; it would not be possible to capture external communication of a software component with MSMs without a significant modification.

UseCaseMaps [3,4] is a notation for 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 scenarios, obtaining the "whole picture" of behavior on the interfaces of a component is not possible.

The *Rigorous Software Development Approach* coined in [19] 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.

In [20], 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.

5. Future Work

In our future research, we aim to define relations and operations on Port State Machines. A compliance relation (possibly captured as ProtocolConformance in UML) relating Protocol State Machines of interfaces of the Port specified by a PoSM would be highly desirable. Moreover, we aim to employ PoSMs to model behavior of the (whole) component; then, goal will be to establish a compliance relation between a Port PoSM and the Component PoSM. A formal specification (utilizing the OCL language) of the compliance relations will be provided.

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.

With the aim to employ PoSMs to model use cases, our future goal is to define operations for assembling behavior scattered in multiple PoSMs into a single PoSM (assembling the "whole picture" behavior). Moreover, a *composition* operation, yielding the composed behavior of multiple connected components (possibly forming together a component at a higher level in the component containment hierarchy) would be desired.

In order to provide a proof of the concept, we aim to implement a UML profile extension supporting PoSMs for a UML tool. After establishing a compliance relation, a compliance verifier tool (possibly reusing the behavior protocols tool [16]) will be implemented.

6. Conclusion

In this paper, we proposed the Port State Machines (PoSMs). Building on UML Protocol State Machines and Behavior Protocols [15], Port State Machines allow to capture the interleaving of operation calls on a set of provided and required interfaces; thus PoSMs can be used to specify the behavior of a Port of a Component or possibly of the Component itself. Important feature of PoSMs is that the state-based behavior specification expressed with a PoSM generates language as a set of traces of atomic events; conveniently, this language is regular. Thus, a compliance relation can be established; formally specifying such a relation is subject of future research.

Acknowledgments

This work was partially supported by the Grant Agency of the Czech Republic project 201/03/0911 and 102/03/0672. A special thank goes to Jiří Adámek for his extremely helpful advices on the Port State Machines. I highly value your deep insight into formal methods.

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

[10] OMG, U2Partners: Unified Modeling Language: Superstructure, version 2.0, 3rd revised submission, ad/03-04-01, http://www.omg.org/uml/

[11] 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/

[12] 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

[13] 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

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

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

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

[17] 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

[18] 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

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

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