An Operator-based Approach to Incremental Development of Conform Protocol State Machines

Arnaud Lanoix, Dieu Donné Okalas Ossami, Jeanine Souquières

An Operator-based Approach to Incremental Development of Conform Protocol State Machines

Arnaud LANOIX Dieu Donné OKALAS OSSAMI Jeanine SOUQUIERES
LORIA – CNRS – Université Nancy 2
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy
E-mail: {lanoix,okalas,souquier}@loria.fr

Abstract

An incremental development framework which supports a conform construction of Protocol State Machines (PSMs) is presented. We capture design concepts and strategies of PSM construction by sequentially applying some development operators: each operator makes evolve the current PSM to another one. To ensure a conform construction, we introduce three conformance relations, inspired by the specification refinement and specification matchings supported by formal methods. Conformance relations preserve some global behavioral properties. Our purpose is illustrated by some development steps of the card service interface of an electronic purse: for each step, we introduce the idea of the development, we propose an operator and we give the new specification state obtained by the application of this operator and the property of this state relatively to the previous one in terms of conformance relation.

Keywords: *Protocol state machine, incremental development, development operator, exact conformance, plugin conformance, partial conformance.*

1. Introduction

Software design is an incremental process where modifications of the system's functionalities can occur at every stage of the development. In order to increase the software quality, it is important to understand the impact of these modifications in terms of lost, added or changed global behaviors.

UML 2.0 [25] introduces protocol state machines (PSMs) to describe valid sequences of operation calls of an object. PSMs are a specialization of generic UML state machines without actions nor activities. Generic state machines are based on the widely recognized statechart notations introduced by Harel [11].

In protocol state machines, transitions are specified in terms of pre/post conditions and state invariants can be given. PSMs are used for developing behavioral abstractions of complex, reactive software. Typically, these state machines provide precise descriptions of component behavior and can be used – combined with a refinement process – for generating implementations. This framework provides a convenient way to model the ordering of operations on a classifier. Notice that the literature about PSMs is quite poor [19, 10].

The notion of conformance of PSMs is an important issue for the development. It is considered in UML 2.0, but limited to explicitly declaring, via the protocol conformance model element, that a specific state machine "conforms" to a general PSM. The definition given in [25] remains very general and does not ease its use in practice.

The conformance between development steps has been studied in formal specification approaches. For example, the B method proposes a refinement mechanism [24, 3, 1]: a system development begins by the definition of an abstract view which can be refined step by step until an implementation is reached. In the framework of algebraic specifications, this notion of conformance has been studied and has given several specification matchings [31]. Meyer and Santen propose a verification of the behavioral conformance between UML and B [21].

This notion is also very important in the field of test. In this domain, conformance is usually defined as testing to see if an implementation faithfully meets the requirements of a standard or a specification. Conformance testing means the use of conformance relations, like the $conf$ or $ioco$ relations [29], based on Labeled Transition Systems (LTS) or process algebras. Other notions of conformance in the context of LTS are the equivalence relations [6], (bi)simulations [23, 8] and refinement [4, 13].

Some notions of conformance have been taken into account for the statecharts [11] or UML 1.x state diagrams. The equivalence of state machines has been studied in [18], the conformance testing in [16] and some refinements in [2, 20, 12]. The majority of these works are based on a semantics of state machines given in terms of LTS using extended hierarchical automata [22, 15, 30].

The idea of following an incremental construction is not new and has been addressed in several works. Some propositions for the incremental design of a part of the statechart specifications are discussed in [27, 10]. An operator-based framework to the incremental development of multi-view UML and B specifications is defined in [26].

This work deals with the incremental development process of PSMs, and, in particular, with the expression of the property between two development steps by means of the conformance relations. Based on formal specification matchings and refinement, we propose three conformance relations, called ExactConformance, PluginConformance and PartialConformance expressing three levels of the preservation of the behavior. In order to help a conform step-by-step construction process, we propose development operators. In [14], we have introduce some operators to deal with subPSMs. This paper extends the approach proposed in [14] by providing other development operators to refine a PSM thanks to the modifications performed on its associated interface.

The paper is structured as follows. Section 2 introduces our running case study and presents UML 2.0 protocol state machines. After a presentation of the UML 2.0 PSM redefinition, Section 3 gives three conformance relations, namely exact, plugin and partial conformances. Section 4 presents some development steps of the case study; for each step we introduce the idea of the development, we propose an operator, we give the new specification state and the property of this state relatively to the previous one in terms of conformance. Section 5 concludes and gives some perspectives.

2. Protocol state machines

This section introduces the UML protocol state machines and the example used throughout this paper.

2.1. Case study: CEPS card

We consider as running example, a part of the Common Electronic Purse Specifications (CEPS) [5]. The system is based on an infrastructure of terminals on which a customer can pay for goods, using a payment card which stores a certain - reloadable - amount of money. In the sequel, we will focus on the card application.

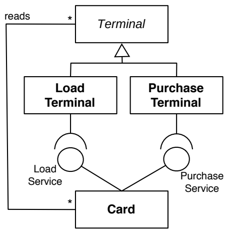


Figure 1. CEPS architecture

Figure 1 shows the architecture of the system: Card represents a payment card while LoadTerminal and PurchaseTerminal represent respectively terminals used to reload the card and terminals used for purchases. Card provides the PurchaseService and LoadService interfaces to communicate with the respective terminals.

2.2. UML 2.0 protocol state machines

PSMs are introduced in UML 2.0 [25] as state machine variants defined in the context of a classifier (interface or class) to model the order of operations calls. PSMs differ from generic state machines by the following restrictions:

- States cannot show entry actions, exit actions, internal actions, or do activities.
- State invariants can be specified.
- Pseudostates cannot be deep or shadow history kinds.
- Transitions cannot show effect actions or send events as generic state machines can.
- Transitions have pre and post-conditions; they can be associated to operation calls.

A PSM may contain one or more regions which involve vertices and transitions. A protocol transition connects a source vertex to a target vertex. A vertex is either a pseudostate or a state with incoming and outgoing transitions. States may contain zero or more regions.

- Pseudostates can be *initial*, *entry point*, *exit point* or *choice* kinds; a choice pseudostate realizes a conditional branch.
- A state without region is a *simple* state; a *final* state is a specialization of a state representing the completion of a region.
- A state containing one or more regions is a *composite* state that provides a hierarchical group of (sub)states; a state containing more than one region is an *orthogonal* state that models a concurrent execution.
- A *submachine* state is semantically equivalent to a composite state. It refers to a submachine (subPSM) where its regions are the regions of the composite state.

2.3. Example: PurchasePSM

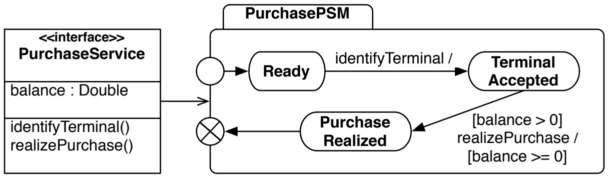


Figure 2. PurchasePSM

In the sequel, we focus on the PurchaseService interface and its associated PSM PurchasePSM given Figure 2.

The interface PurchaseService provides an attribute, balance, which represents the amount of money available on the card. The PSM PurchasePSM describes the following behavior: its initial state is Ready. First, the purchase terminal, used to read the card, is authentified and the TerminalAccepted state is reached. Next, the PSM reaches the PurchaseRealized state if there is enough money on the card, which is ensured by the precondition [balance $>$ 0].

3. Conformance relations

The protocol conformance relation [25] is used to explicitly declare that a specific state machine conforms to a general PSM. The given semantics is the preservation of pre/post conditions and state invariants of the general PSM in the more specific one. For our point of view, the definition of the protocol conformance relation remains too very general to be used in practice and does not allow the designer how to decide on conformance between two PSMs.

State machine redefinition is also considered in UML 2.0. A specialized state machine is an extension of a general state machine where regions, vertices and transitions have been added or redefined. So, it has additional elements.

A simple state can be redefined to a composite state by adding one or more regions. A composite state can be redefined by either extending its regions or by adding regions as well as by adding entry and exit points. A region can be extended by adding vertices and transitions and by redefining states and transitions. A submachine state may be redefined by another submachine state that provides the same entry/exit points and adds new entry/exit points.

Let PSM$_1$ and PSM$_2$ be a PSM and another PSM obtained by a transformation of PSM$_1$ by performing a development step. In order to study the construction-based conformance between PSM$_1$ and PSM$_2$, we introduce three relations. These relations describe different levels of behavioral preservations corresponding to properties of the new PSM relatively to the previous one.

1. PluginConformance: PSM$_2 \sqsubseteq$PSM$_1$.

   We have a PluginConformance relation between PSM$_2$ and PSM when PSM$_2$ provides all the functionalities of PSM$_1$ and when the new functionalities provided by PSM$_2$ don't conflict with the ones of PSM$_1$. We are able to "plugin" PSM$_2$ for PSM$_1$.

2. PartialConformance: PSM$_2 \sqsupseteq$PSM$_1$.

   The PartialConformance relation is the reciprocal relation of the PluginConformance relation: PSM$_2 \sqsupseteq$PSM$_1$ iff PSM$_1 \sqsubseteq$PSM$_2$. In other words, this relation occurs between PSM$_2$ and PSM$_1$ when PSM$_2$ provides less functionalities than PSM$_1$, but all the functionalities provided by PSM$_2$ are provided by PSM$_1$.

3. ExactConformance: PSM$_2 \equiv$PSM$_1$.

   We have an ExactConformance relation between PSM$_2$ and PSM$_1$ if the two PSMs are equivalent and completely interchangeable. All Observable functionalities provided by PSM$_1$ and by PSM$_2$ must be the same. The ExactConformance relation is symmetric.

   The ExactConformance relation is a specialization of both PluginConformance and PartialConformance relations; we can easily demonstrate that if PSM$_2 \equiv$PSM$_1$ then PSM$_2 \sqsubseteq$PSM$_1$ and PSM$_2 \sqsupseteq$PSM$_1$.

Notice that the ExactConformance relation is a strong requirements often incompatible with a construction process. Sometimes a weaker match as PluginConformance or PartialConformance can be enough.

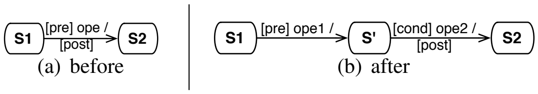
There is no formal definitions of the previous relations in this paper. Interested reader might find some proposals in [18, 16, 20, 12]. We focus on their uses to guid an incremental developement.

4. Conform development

Let us see some development steps of the case study, starting from PurchasePSM and its associated interface PurchaseService, presented Figure 2. Our objective is to elaborate from this state a more complete PSM that presents the functionalities provided by the card following the interface modifications. For each step, we give the general idea of the evolution involved which respects to the new associated interface, the development operator which is applied on the current state and the conformance property that is preserved, which is the properties of the new state relatively to the previous one.

4.1. Introducing Sequences of operations

Figure 2 gives an abstraction of the authentication process. The operation identifyTerminal() can be decomposed by the sequence of operations readCertificate(term_id), followed by acceptTerminal().



(a) before

(b) after

Figure 3. refine_by_sequences()

This sequence is formally described by an UML annotation. The syntax used is the following:
ope() := ope1() ; [cond] ope2()

that expresses the substitution of ope() by ope1() followed by ope2() under the condition [cond] (see Figure 3).

We define a construction operator refine_by_sequences() which substitutes the considered transition by the sequence of new transitions as shown Figure 3. If [cond] is defined, then PartialConformance is preserved by this operator; otherwise, ExactConformance is preserved.

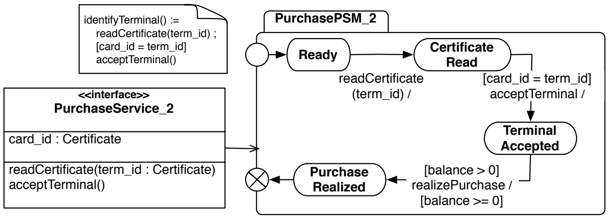
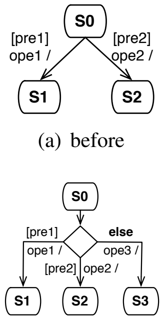


Figure 4. PurchasePSM_2

The PSM PurchasePSM_2, given Figure 4, corresponds to the application of the operator refine_by_sequences() on the transition identifyTerminal which substitutes identifyTerminal by readCertificate(term_id) and acceptTerminal. Figure 4 shows also the modifications of the interface associated to PurchasePSM. A new attribute card_id is added to authenticate a terminal by exchange of certificates[1].

4.2. Introducing complementary behaviors



(a) before

(b) after

Figure 5. complement_transition()

When looking at the transition acceptTerminal between the states CertificateRead and TerminalAccepted on Figure 4, we remark that all the possible cases are not considered. The case where a valid terminal certificate is read, expressed by the precondition [card_id = term_id], is the only one to be taken into account. What happens when term_id is not a valid certificate? This new requirements involves the introduction of a new transition and a new state.

The operator complement_transition() proposes to introduce from a selected vertex and its outgoing transitions, a (default) complementary transition by using a choice pseudostate as shown Figure 5. Since the operator complement_transition() adds new functionalities, PluginConformance is preserved.

Applying the complementary_transition() operator on the state CertificateRead leads to a new PSM PurchasePSM_3 shown Figure 6. A choice pseudostate and a new state TerminalRefused are introduced.

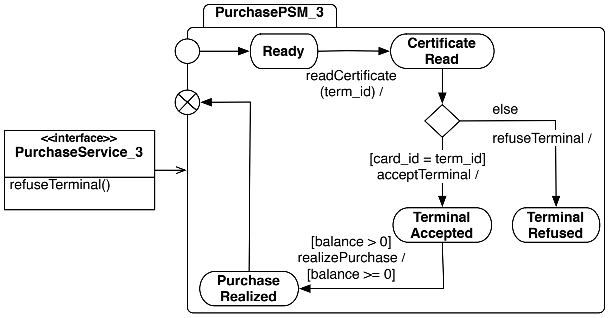


Figure 6. PurchasePSM_3

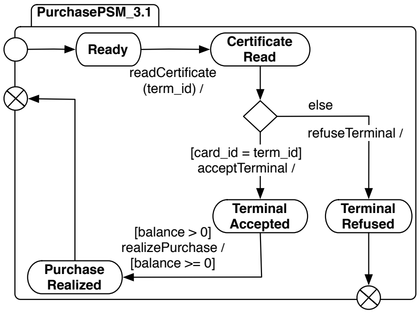

PurchasePSM_3.1
Ready
Certificate Read
readCertificate (term_id) /
else
refuseTerminal /
[card_id = term_id] acceptTerminal /
Terminal Accepted
Terminal Refused
[balance > 0] realizePurchase / [balance >= 0]
Purchase Realized

Figure 7. PurchasePSM_3.1

Figure 7, a new exit point is introduced jointly with a transition from the TerminalRefused state to the new exit point using basic construction operators add_vertex() and add_transition() defined in [14]. Then, PluginConformance is preserved.

4.3. Reusing refine_by_sequences()

Let us consider now the transition realizePurchase between TerminalAccepted and PurchaseRealized states. We want to decompose this transition into two successive transitions initializePurchase(amount) and achievePurchase to describe more precisely the purchase functionality.

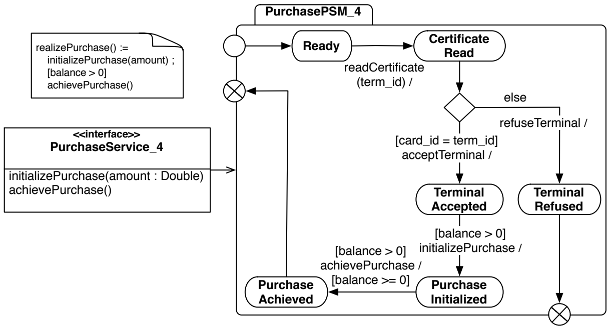


Figure 8. PurchasePSM_4

The previous operator refine_by_sequences() is applied again to obtain a new PSM PurchasePSM_4 given Figure 8.

[1] Notice that PurchaseService_2 interface shows only the updated informations of PurchaseService.

4.4. Introducing conditional behaviors

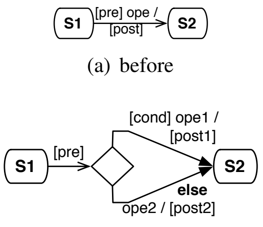


Figure 9. refine_by_conditions()

In the current development state, the achievePurchase transition is still abstract. It corresponds to two (conditional) behaviors: if there is enough money on the card to pay the purchase, then the purchase is realized and the balance is debited. Otherwise, the purchase must be canceled.

A construction operator refine_by_conditions() is defined to substitute the considered transition by a conditional behavior expressed by an UML annotation which respects the following syntax:
ope() := **if** [cond] **then** ope1() [post1] **else** ope2() [post2]
Figure 9 illustrates this operator. It preserves the ExactConformance when the following obligation proofs are satisfied:

- (pre@**pre** **and** cond@**pre** **and** post1) **implies** post
- (pre@**pre** **and not** cond@**pre** **and** post2) **implies** post

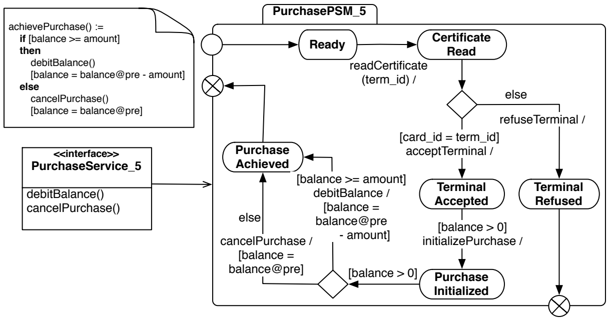


Figure 10. PurchasePSM_5

The application of refine_by_conditions() on achievePurchase gives the new PSM PurchasePSM_5 by substituting the achievePurchase transition by debitBalance and cancelPurchase (see figure 10).

Since (balance@**pre** > 0 **and** balance@**pre** >= amount **and** balance = balance@**pre** - amount) **implies** (balance >= 0), and, (balance@**pre** > 0 **and** balance@**pre** < amount **and** balance = balance@**pre**) **implies** (balance > 0) are satisfied, we conclude that ExactConformance is preserved.

4.5. Splitting states

We can observe in PurchasePSM_5 that the two transitions debitBalance and cancelPurchase reach the same state PurchaseAchieved. Nevertheless, they describe different behaviors. We want to split PurchaseAchieved into two different states BalanceDebited and PurchaseCanceled to illustrate the difference.

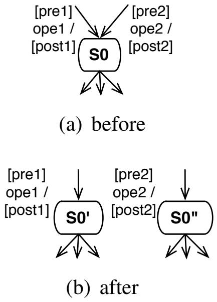


Figure 11. split_state()

The construction operator split_state() depicted Figure 11 considers a vertex and its incoming transitions. For each incoming transition, the vertex is duplicated. All the outgoing transitions are also duplicated. Since this construction operator only duplicates behaviors, it preserves ExactConformance.

The application of this operator to the state PurchaseAchieved gives two new states BalanceDebited and PurchaseCanceled as shown Figure 12.

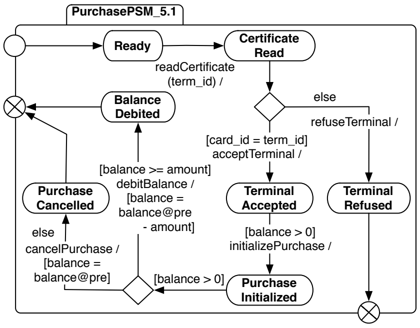

PurchasePSM_5.1
Ready
Certificate Read
readCertificate (term_id) /
Balance Debited
else
refuseTerminal /
[card_id = term_id] acceptTerminal /
[balance >= amount] debitBalance / [balance = balance@pre - amount]
Purchase Cancelled
Terminal Accepted
Terminal Refused
else
cancelPurchase / [balance = balance@pre]
[balance > 0] initializePurchase /
[balance > 0]
Purchase Initialized

Figure 12. PurchasePSM_5.1

When applying once again the split_state() operator to the exit pseudostate, we obtain the PSM PurchasePSM_5.2 given Figure 13.

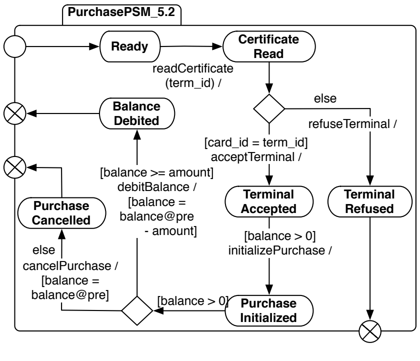

PurchasePSM_5.2
Ready
Certificate Read
readCertificate (term_id) /
Balance Debited
else
refuseTerminal /
[card_id = term_id] acceptTerminal /
[balance >= amount] debitBalance / [balance = balance@pre - amount]
Purchase Cancelled
Terminal Accepted
Terminal Refused
else
cancelPurchase / [balance = balance@pre]
[balance > 0] initializePurchase /
[balance > 0]
Purchase Initialized

Figure 13. PurchasePSM_5.2

An overview of a part of the followed development process is given Appendix A. Each development state is composed of a PSM and its associated interface and transitions between development states express the application of a development operators and the properties between two states: Refinement for interfaces and Conformance for PSMs.

5. Conclusion and future work

Specifying complex systems is a difficult task which cannot be done in one step. In a typical design process, the designer starts with a first draft model and transforms it by a step-by-step process into a more and more complex model.

The design approach we propose in this paper uses a set of construction operators to make evolve protocol state machines preserving behavioral properties. Three Conformance relations ExactConformance, PluginConformance and PartialConformance have been defined. The use of these operators has been illustrated on the development of a part of the CEPS case study.

Further work will focus on a generalization of our step-by-step construction of PSM by studying other construction operators, like operators for removing elements. We are currently exploring other particularities of PSMs like state invariants and transition post-conditions.

We also consider the formalization of the definition of the Conformance relations ExactConformance, PluginConformance and PartialConformance inspired by results in formal methods like refinement [1] and specification matchings [31]. The verification of the conform development can be done by translating the obtained PSM into a tool-supported language such that B [17, 28] or TLA [7, 9].

Another perspective concerns the implementation of a tool to assist in the development of PSMs based on our construction operators.

References

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] M. Al'Achhab. Specification and verification of hierarchical systems by refinement. In *Modelling and Verifying Parallel Processes (MOVEP'04)*, 2004.

[3] R. J. Back. A calculus of refinements for program derivations. *Acta Informatica*, (25):593–624, 1988.

[4] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *Fundamental Aspects of Software Engineering (FASE'00)*, volume 1783 of *LNCS*, pages 266–283. Springer Verlag, 2000.

[5] CEPSCO. Common electronic purse specifications, functional requirements, v6.3, 1999.

[6] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24(2):211–237, 1987.

[7] T. Deiss. An Approach to the Combination of Formal Description Techniques: Statecharts and TLA. In *1st International Conference on Integrated Formal Methods, IFM'99*, pages 231–250. Springer, 1999.

[8] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3):219–236, May 1990.

[9] C. Freinkel. An Approach to Combining UML and TLA+ in Software Specification. Technical reports, University of Nevada, Reno, 2003.

[10] O. Gout and T. Lambolais. UML Protocol State Machines Incremental Construction: a Conformance-based Refinement Approach. Research Report RR05/027, LGI2P, 2005.

[11] D. Harel. *Modeling Reactive Systems With Statecharts*. Mac Graw Hill, 1998.

[12] A. Knapp, S. Merz, M. Wirsing, and J. Zappe. Specification and refinement of mobile systems in MTLA and mobile UML. *Theoretical Computer Science*, 2005.

[13] O. Kouchnarenko and A. Lanoix. Refinement and verification of synchronized component-based systems. In K. Araki, S. Gnesi, and M. D., editors, *Formal Methods (FM'03)*, volume 2805 of *LNCS*, pages 341–358. Springer Verlag, 2003.

[14] A. Lanoix and J. Souquières. A step-by-step process to build conform UML protocol state machines. Research Report ccsd-00019314, LORIA, Fev 2006.

[15] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *3rd Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 331–347. Kluwer, 1999.

[16] D. Latella and M. Massink. On testing and conformance relations of UML statechart diagrams behaviours. In ACM, editor, *Int. Symposium on Software Testing and Analysis*, 2002.

[17] H. Ledang and J. Souquières. Contributions for modelling UML state-charts in B. In *Third International Conference on Integrated Formal Methods - IFM'2002*, Turku, Finland, 2002.

[18] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of statecharts. In *Proc. of the 7th Int. Conf. On Concurrency Theory (CONCUR'96)*, pages 687–702. Springer-Verlag, 1996.

[19] V. Mencl. Specifying component behavior with port state machines. *ENTCS*, 101C:129–153, 2004.

[20] S. Meng, Z. Naixiao, and L. S. Barbosa. On semantics and refinement of UML statecharts: A coalgebraic view. In *Proc. of the 2nd In. Conf. on Software Engineering and Formal Methods (SEFM'04)*, 2004.

[21] E. Meyer and T. Santen. Behavioral Conformance Verification in an Integrated Approach Using UML and B. In *(IFM00), Integrated Formal Methods*, volume 1945 of *LNCS*, page 358. Springer Verlag, 2000.

[22] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Third Asian Computing Science Conference on Advances in Computing Science (ASIAN'97)*, pages 181–196, London, UK, 1997. Springer Verlag.

[23] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[24] J. M. Morris. A theoretical basis for stepwise refinement and programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[25] Object Management Group. UML superstructure specification, v2.0, 2005.

[26] D.-D. Okalas Ossami, J. Souquières, and J.-P. Jacquot. Consistency in UML and B multi-view specifications. In *Proc. of the Int. Conf. on Integrated Formal Methods, IFM'05*, number 3771 in LNCS, pages 386–405. Springer-Verlag, 2005.

[27] P. Scholz. Incremental design of statechart specifications. *Science of Computer Programming*, 40(1):119–145, 2001.

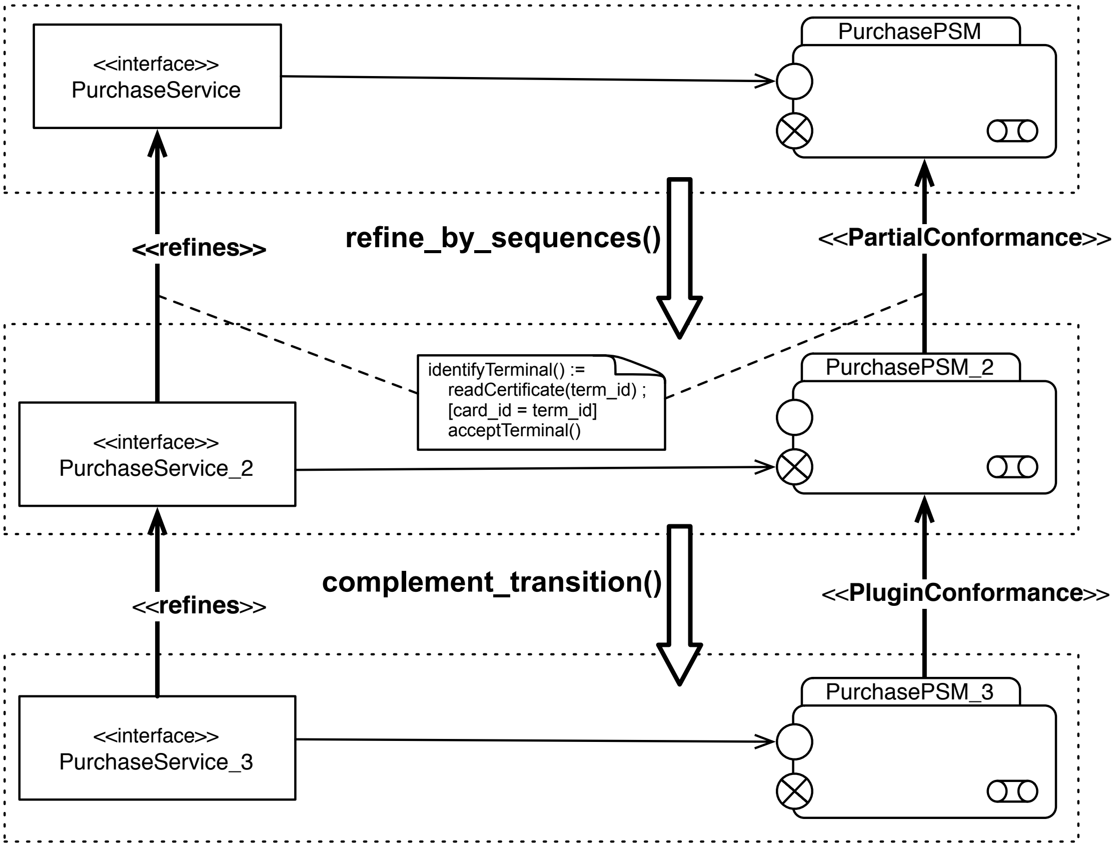
[28] E. Sekerinski and R. Zurob. Translating statecharts to b. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 128–144, London, UK, 2002. Springer-Verlag.

[29] J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[30] M. Von der Beeck. Formalization of UML-Statecharts. In *UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 406–421. Springer-Verlag, 2001.

[31] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transaction on Software Engeniering Methodolology*, 6(4):333–369, 1997.

A. Incremental development of PurchasePSM



<<interface>>
PurchaseService
PurchasePSM
refine_by_sequences()
<<refines>>
<<PartialConformance>>
identifyTerminal() :=
readCertificate(term_id) ;
[card_id = term_id]
acceptTerminal()
PurchasePSM_2
<<interface>>
PurchaseService_2
complement_transition()
<<refines>>
<<PluginConformance>>
PurchasePSM_3
<<interface>>
PurchaseService_3