A Step-by-step Process to Build Conform UML Protocol State Machines

Arnaud Lanoix, Jeanine Souquières

▶ **To cite this version:**

Arnaud Lanoix, Jeanine Souquières. A Step-by-step Process to Build Conform UML Protocol State Machines. 2006. hal-00019314

HAL Id: hal-00019314
https://hal.science/hal-00019314v1

Preprint submitted on 20 Feb 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Step-by-step Process to Build Conform UML Protocol State Machines

Arnaud Lanoix
LORIA – CNRS
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy
Arnaud.Lanoix@loria.fr

Jeanine Souquières
LORIA – Université Nancy 2
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy
Jeanine.Souquieres@loria.fr

ABSTRACT

We propose an approach to the incremental development of protocol state machines using operators which preserve behavioral properties. We introduce two specializations of the protocol conformance relation proposed in UML 2.0, inspired from the work on formal methods as the specification refinement and specification matching. We illustrate our purpose 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, construction operator, exact conformance, plugin conformance

1. INTRODUCTION

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

UML 2.0 [22] introduces protocol state machines (PSMs) to describe valid sequences of operation calls of an object. These PSMs are a specialization of generic UML state machines without actions nor activities. Transitions are specified in terms of pre/post conditions and state invariants can be given. State machines are used for developing behavioral abstractions of complex, reactive software. Typically, 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 also that the literature about PSMs is quite poor [16].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SCESM '06, Shanghai, China
Copyright 2006 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

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 [22] remains very general and does not ease it use in practice.

The conformance between development steps has been studied in formal specification approaches. For example, the B method proposes a refinement mechanism [21, 4, 1]: a system development begins by the definition of an abstract view which can be refined step by step until an implementation is reached. The refinement over models is a key feature for developing incrementally models from a textually-defined system, while preserving correctness. It implements the proof-based development paradigm [3, 25]. In the framework of algebraic specifications, this notion of conformance has been studied and has given several specification matching [29]. Meyer and Santen propose a verification of the behavioral conformance between UML and B [18].

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 [26, 27], based on Labeled Transition Systems (LTS) or process algebras.

More generally, there are a lot of studies about conformance relations between two LTSs. Among them, we can cite equivalence relations [8], (bi)simulations [20, 9] or refinement [5, 12].

The notion of conformance have been taken into account for the statechart [10] or UML 1.x state diagrams [6]. The equivalence of state machines has been studied in [15], the conformance testing in [14] and some refinements in [2, 17, 11]. The majority of these works are based on a semantics of state machines given in terms of LTS using extended hierarchical automata [19, 13, 28].

The idea of following an incremental construction is not new and has been addressed in several works. For example, Scholz [24] makes proposition for the incremental design of a part of the statechart specifications. A formal definition of the consistency between UML and B, based on transformation rules is defined in [23]. The proposed framework based on multi-view specifications and development operators, takes into account the specification development process to guarantee the production of correct specifications.

Our work deals with the incremental development process of PSMs, and, in particular, with the conformance between

two development steps. In order to help a conform step-by-step construction process, we propose development operators. Based on formal specification matching, we propose two specializations of the protocol conformance relation, called ExactConformance and PluginConformance expressing two levels of the preservation of the behavior.

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 UML 2.0 PSM redefinition, Section 3 gives two specializations of the protocol conformance, namely the exact conformance and the plugin conformance. Section 4 presents some development steps of the case study; for each step we introduce the idea of the development, we propose an operator, and 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

The Unified Modeling Language (UML) features state machines is based on the widely recognized statechart notation introduced by Harel [10] to express behavior of various model elements (i.e. class or interface). UML 2.0 [22] introduces a specialization of state machines, called the ProtocolStateMachine (PSM), to express usage protocol. It is a convenient way to model life-cycle for objects by providing support for modeling the order of invocation of its operations.

2.1 Case study: CEPS card

We consider as running example, a part of the Common Electronic Purse Specifications (CEPS) [7]. 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.

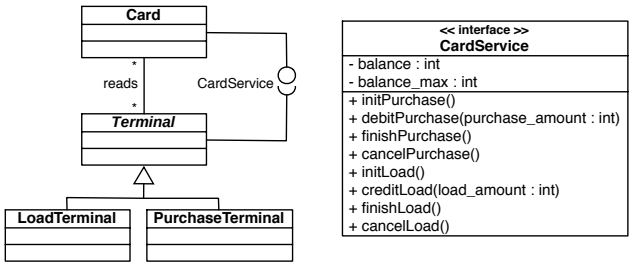

Figure 1: Card class diagram

Figure 1 show the main classes of the system: Card represents a payment card while LoadTerminal and PurchaseTerminal represent respectively terminals used to reload the payment card and terminals used for purchases. The Card provides the CardService interface to communicate with the terminals.

The interface CardService provides two attributes: balance represents the amount of money available on the card and balance_max the maximum amount of money associated to the card. As specified in the class diagram, terminals can only interact with the card through the methods provided by the interface. These methods are:

- initPurchase() models the initialization of a purchase,
- debitPurchase(purchase_amount: int) models the debit of purchase_amount from the card balance,
- finishPurchase() models the end of a purchase,
- cancelPurchase() models the cancel of a purchase,
- initLoad() models the initialization of a load,
- creditLoad(load_amount: int) models a credit of the card balance,
- finishLoad() models the end of a load,
- cancelLoad() models the cancel of a load.

2.2 UML 2.0 protocol state machines

A protocol state machine has the characteristics of a generic state machine (composite states, concurrent regions, etc.) with the next restrictions on states and transitions:

- 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; they are restricted to initial, entry point and exit point 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.

- 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 (sub PSM) where its regions are the regions of the composite state.

Figure 2 presents the abstract syntax of the ProtocolStateMachine model element.
We now introduce some basic definitions used below.

- An *unreachable* vertex is a vertex which is a target of any incoming transitions. This is expressed in OCL by vertex.incoming->isEmpty().
- Two outgoing transitions trans_i and trans_j of a same state are *inconsistent* if their respective preconditions are inconsistent. This is expressed in OCL by **not** ((trans_i.preCondition **implies** trans_j.preCondition) **or** (trans_j.preCondition **implies** trans_i.preCondition)).
- A *crossing* transition trans is a transition where its source state and its target state are not in the same region. This is expressed in OCL by **not** (trans.source.container = trans.target.container).

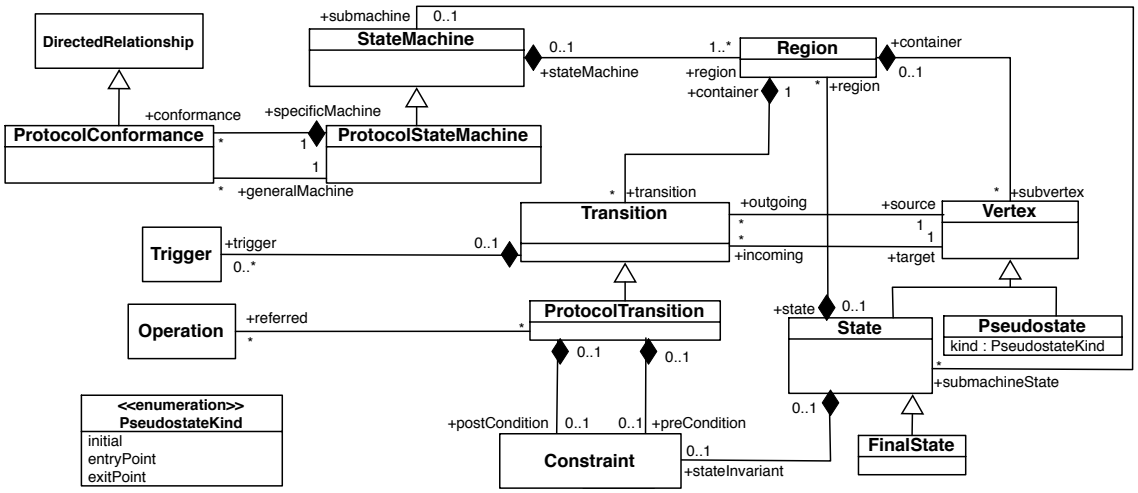

Figure 2: Overview of the abstract syntax of the ProtocolStateMachine **model element**

2.3 Example: CardPSM

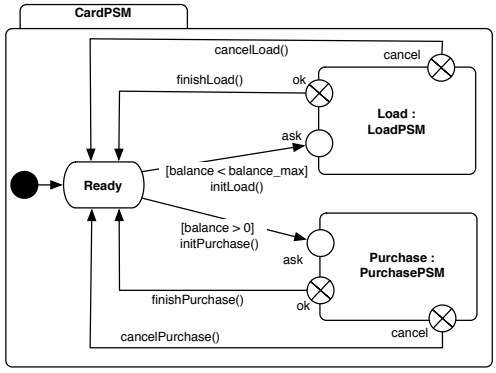

Figure 3: CardPSM

We associate a PSM called CardPSM to the CardService interface. As presented Figure 3, it includes two sub PSMs: PurchasePSM for the purchase functionalities, and LoadPSM for the load functionalities.

The initial state of CardPSM is Ready. Once a terminal activates the initPurchase() method and if there is money on the card (expressed by the precondition [balance>0]), a purchase is initialized, and the entry point ask of the submachine state Purchase of PurchasePSM is reached (see Figure 4).

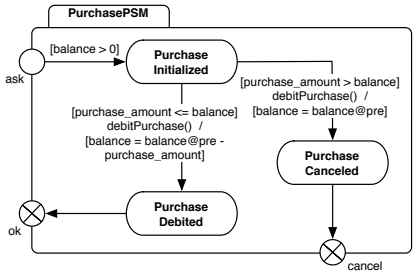

Figure 4: PurchasePSM

- If there is enough money on the card, which is ensured by precondition [purchase_amount <= balance], the debitPurchase() method is called and the purchase is realized. The sub PSM reaches the PurchaseDebited state. The money on the card must be decreased: this is expressed by the post-condition of the transition [balance = balance@pre - purchase_amount]). Finally, the exit point ok is reached. Then, PurchasePSM is exited and the card returns to the state Ready by the activation of the method finishPurchase().
- If there is no enough money on the card, that is ensured by the precondition [purchase_amount > balance], the purchase is canceled. First, a state PurchaseCanceled is reached, followed by the exit point cancel, that exits the sub PSM PurchasePSM. The Ready state of CardPSM is now reached using the method cancelPurchase().

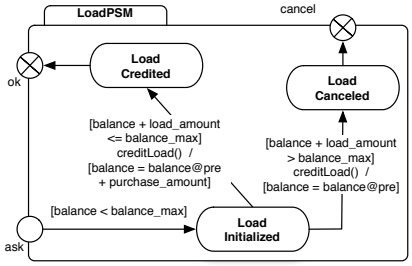

Figure 5: LoadPSM

A load is initialized from the state Ready of CardPSM when the precondition [balance < balance_max] is true and the initPurchase() method is called; that initializes an instance of LoadPSM. The sub PSM LoadPSM describes all the behaviors corresponding to a reload of the card, as shown Figure 5.

3. CONFORMANCE RELATIONS

The protocol conformance relation [22] is used to explicitly declare that a specific state machine conforms to a general

PSM (see Figure 2). 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 entry/exit points.

Our purpose is to introduce specializations of the protocol conformance relation to describe different levels of conformance preserved by the incremental construction. Let PSM and PSM' be respectively a PSM and a transformation (i.e. a redefinition) of this PSM.

1. ExactConformance: PSM'≡PSM.

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

2. PluginConformance: PSM'⊑PSM.

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

It is to be noted that the ExactConformance relation is a strong requirements often incompatible with a construction process, which adds add new functionalities. Sometimes a weaker match can be enough. As shown Figure 6, the ExactConformance relation is a specialization of the PluginConformance relation; we can easily demonstrate that if PSM'≡PSM then PSM'⊑PSM.

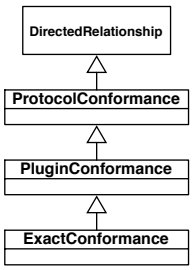

Figure 6: Hierarchy of conformance relations

We have introduced another conformance relation, denoted by PSM'⊒PSM, whose is the reciprocal relation of the PluginConformance relation: PSM'⊒PSM iff PSM⊑PSM'. In other words, this relation occurs between PSM' and PSM when PSM' provides less functionalities than PSM, but all the functionalities provided by PSM' are provided by PSM.

Notation. In UML 2.0 [22], the keyword "extended" is used to express that a state machine is an extension of another state machine. We propose the keywords "exact" and "plugin" to express the two conformance relations we have introduced.

4. CONFORM DEVELOPMENT

Let us see some development steps of the case study, starting from the protocol state machine CardPSM_0 presented Figure 7. It gives a first view of a part of the services offered by the interface CardService.

Its initial state is Ready. If there is money on the card, the state PurchaseInitialized is reached. Next, the PSM reaches the PurchaseDebited state, if there is enough money on the card, which is ensured by precondition [purchase_amount <= balance]: the money on the card must be decreased, expressed by the post-condition [balance = balance@pre - purchase_amount]. Finally, if the precondition [0 <= balance] is verified, the card returns to the state Ready.

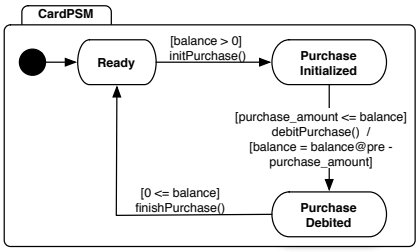

Figure 7: CardPSM_0

Our objective is to elaborate from CardPSM_0 a more complete PSM that presents the functionalities provided by the CardService interface. For each step, we give the general idea of the evolution involved, the development operator which is applied on the current state and the conformance property that is preserved.

4.1 Modifying preconditions

In this first description, the precondition [0 <= balance] of the transition finishPurchase is useless: [0 <= balance] is always implied by the previous transition. We want delete this precondition, i.e. replace it by a new one equals to true.

We have defined the operator Transition::change_preCondition(newPreCondition:Constraint) which replaces the preCondition of a Transition by a new one, newPreCondition. This operator preserves

- the PluginConformance if newPreCondition is weaker than preCondition. This is expressed in OCL by preCondition **implies** newPreCondition;
- the ExactConformance if newPreCondition is equivalent to preCondition. This is expressed in OCL by (preCondition **implies** newPreCondition) **and** (newPreCondition **implies** preCondition).

Figure 8 gives the result of the application of change_preCondition() on the transition finishPurchase of CardPSM_0.

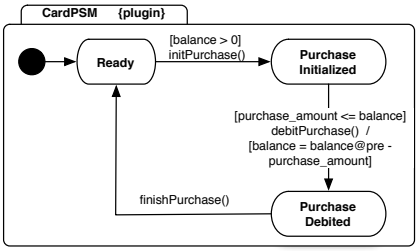

Figure 8: Step 1 – CardPSM_1

PluginConformance is preserved because 0 <= balance **implies** true.

4.2 Introducing complementary behaviors

When looking at the transition between the states PurchaseInitialized and PurchaseDebited, we see that all the possible cases are not expressed. What happens when purchase_amount > balance ? It can be noticed that this case is the complementary of the precondition of debitPurchase; it corresponds to the case where there is not enough money on the card to realise the initialized purchase. In this case, the transition debitPurchase cannot been done and a new state has to be introduced.

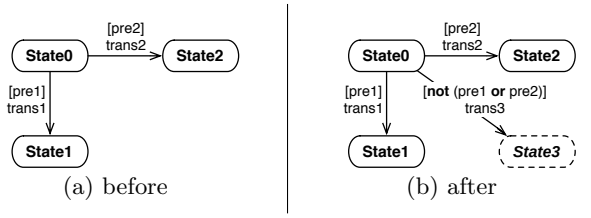

Figure 9: Operator Vertex::complementary_transition()

We have defined a construction operator Vertex::complementary_transition(), that suggests, from a *selected* Vertex and its outgoing transitions, a *complementary* transition such that the conjunction of all the preconditions of the Vertex.outgoing transitions with the precondition of the complementary transition is equal to true.

Figure 9 illustrates the behavior of this operator. It proposes a new transition trans3 and its target state State3 such that

$$(\text{pre1} \ \textbf{and} \ \text{pre2} \ \textbf{and} \ \text{pre3}) = \text{true}$$

where pre3 is the precondition of trans3 defined by

$$\text{pre3} = \ \textbf{not} \ (\text{pre1} \ \textbf{or} \ \text{pre2})$$

This operator is defined in terms of two basic operators:

- Region::add_vertex(newVertex: Vertex) that adds a new Vertex to an existing Region; it preserves ExactConformance, and
- Vertex::add_transition(newTransition: Transition) which adds a newTransition if no inconsistent transitions exist from the considered Vertex; generally, this operator preserves PluginConformance. In the case where Vertex is unreachable, ExactConformance is preserved.

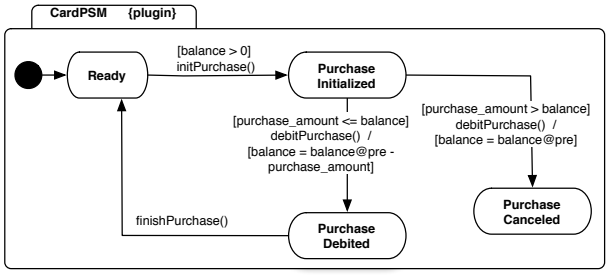

Figure 10: Step 2 – CardPSM_2

PluginConformance is preserved by the operator complementary_transition().

The application of complementary_transition() on the state PurchaseInitialized of CardPSM_1 gives the PSM CardPSM_2 presented Figure 10.

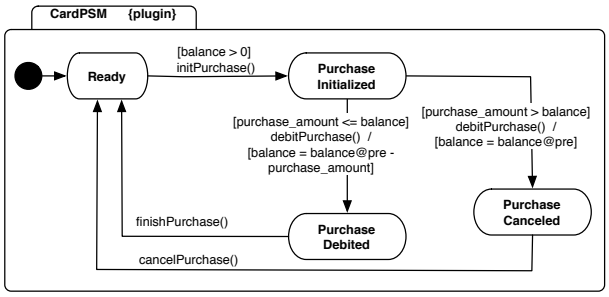

Figure 11: Step 3 – CardPSM_3

Figure 11, a new transition from the state PurchaseCanceled to the state Ready has been added by application of the operator add_transition().

4.3 Merging existing states

We have followed until now a bottom-up development process. The result of this process expressed by the PSM CardPSM_3 includes

- general informations on the card service interface, and
- dedicated informations about the purchase functionalities.

An idea to make evolve our model is to regroup those dedicated informations – the states PurchaseInitialized, PurchaseDebited and PurchaseCanceled – into a new composite state, giving it a name as presented Figure 12.

The operator Region::merge_states(SET(State)), which is parameterized by a set of states, is dedicated to regroup these *selected* states into a new composite state, as shown Figure 13. This operator preserves ExactConformance because it does not modify the behavior, it only change the "view" of the considered PSM. It is defined as a sequence of basic operators:

- add_vertex() to add a new state,
- State::composite() to transform this new state into a composite state by adding a new empty region,
- State::change_Container(newContainer:Region) to move the selected states to the new composite state.

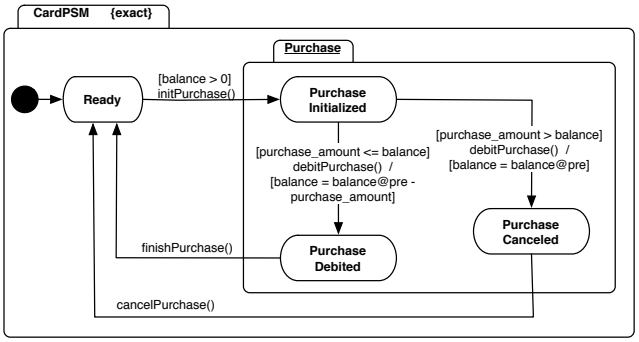

Figure 12: Step 4 – CardPSM_4

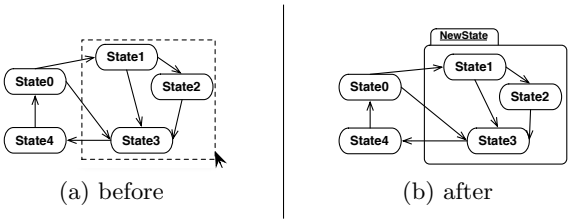

Figure 13: Operator Region::merge_states()

4.4 Adding an interface to a composite state

The composite state Purchase contains substates with incoming and outgoing crossing transitions, as shown Figure 12. We propose to build, from the Purchase box, a component that interfaces these crossing transitions, introducing explicit entry point and exit point pseudostates to replace all the crossing transitions using the construction operator State::add_interface(). It is defined as follow:

- for each crossing substate_i.incoming transition, identified transition_i, we add an entry point pseudostate entry_i. We connect transition_i to entry_i and we add a new transition from entry_i to substate_i which precondition is transition_i.preCondition;
- for each crossing substate_i.outgoing transition, denoted transition_i, we add an exit point pseudostate exit_i. We connect transition_i from exit_i and we add a new transition from substate_i to exit_i which precondition is transition_i.preCondition.

This transformation preserves ExactConformance.
The result of the application of this operator on CardPSM_4 gives the new PSM CardPSM_5 presented Figure 14.

An alternative to this development step could be to introduce initial and final pseudostates to replace the crossing transitions.

4.5 Extracting sub PSMs from an existing PSM

At this stage of the development, CardPSM_5 contains a composite state Purchase interfaced with the remainder of the PSM using entry point and exit point pseudostates. Our idea is to extract from this composite state a sub PSM and replace the composite state by a *submachine* state, which instantiates the extracted sub PSM.

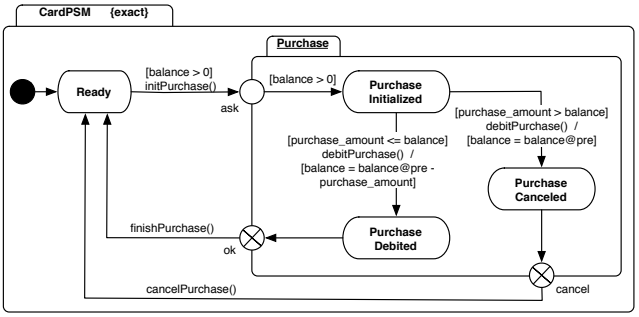

Figure 14: Step 5 – CardPSM_5

The operator State::extract_submachine() creates a subPSM from a composite state. The regions of the composite state are now the regions of the subPSM. It is the same for the substates, the transitions and the pseudostates of the composite state. As this transformation is defined in UML 2.0 as an equivalence relation, extract_submachine() preserves ExactConformance.

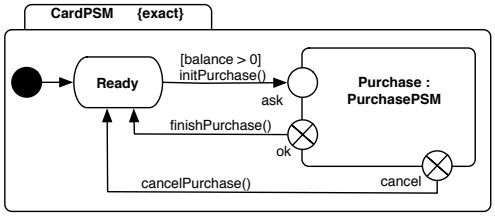

Figure 15: Step 6 – CardPSM_6

When applying the operator extract_submachine() to the composite state Purchase, we obtain the PSM PurchasePSM shown Figure 4. The CardPSM_5 machine is now defined using an instance of PurchasePSM as presented Figure 15.

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. Two Conformance relations ExactConformance and PluginConformance have been defined as specializations of the UML 2.0 protocol conformance relation. 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 method of PSM by studying other construction operators, particularly operators for removing elements: if the source of a transition is unreachable, then removing the transition preserves the ExactConformance relation, as removing an unreachable vertex or an empty region. We are currently exploring other particularities of PSMs like state invariants and transition post-conditions, as well as the study of the weakness of preconditions.

We also consider the formalization of the definition of the

Conformance relations ExactConformance and PluginConformance inspired by results in formal methods like refinement [1] and specification matching [29].

Another perspective concerns the implementation of a tool to assist in the development of PSMs based on our construction operators. Consider as example the operator complementary_transition() presented section 4.2. An issue could be an UML modeler which proposes automatically the complementary transition when we select a state.

6. 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] B-Core(UK) Ltd. *B-Toolkit User's Manual, Release 3.2*, 1996.

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

[5] 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.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] E. Meyer and S. T. 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.

[19] 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.

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

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

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

[23] 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.

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

[25] Steria. *Obligations de preuve: Manuel de référence, version 3.0*.

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

[27] J. Tretmans. Testing Concurrent Systems: A Formal Approach. In J. Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conf. on Concurrency Theory*, volume 1664 of *LNCS*, pages 46–65. Springer-Verlag, 1999.

[28] 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.

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