An Operator-based Approach to Incremental Development of Conform Protocol State Machines
Arnaud Lanoix, Dieu Donné Okalas Ossami, Jeanine Souquières

To cite this version:

HAL Id: inria-00001263
https://inria.hal.science/inria-00001263
Submitted on 24 Apr 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.
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, 8, 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 Sante 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 inoco 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, 12].

The idea of following an incremental construction is not new and has been addressed in several works.
sitions 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 pro-
cess 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 confor-
mance relations, called ExactConformance, PluginConfor-
mance 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 associ-
ated 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 redefi-
ition, 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 oper-
ator, we give the new specification state and the property of
this state relatively to the previous one in terms of confor-
mance. 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 exam-
ple, 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 - reload-
able amount of money. In the
sequel, we will focus on the card
application.

Figure 1 shows the architecture of the system: Card
represents a payment card while LoadTerminal and Pur-
chaseTerminal represent respectively terminals used to
reload the card and terminals used for purchases. Card pro-
vides 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 pseu-
dostate or a state with incoming and outgoing transitions.
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 redefi-
ition, 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 oper-
ator, we give the new specification state and the property of
this state relatively to the previous one in terms of confor-
mance. Section 5 concludes and gives some perspectives.
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 authenticated 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 \( \text{balance} > 0 \).

3. Conformance relations

The protocol conformance relation \([\ref{25}]\) 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 \( PSM_1 \) 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 requirement 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 development.

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().

![Figure 3. refine_by_sequences()](image)

This sequence is formally described by an UML annotation. The syntax used is the following:

\[ \text{ope()} := \text{ope1()} ; \text{[cond]} \text{ope2()} \]
that expresses the substitution of \( \text{ope1} \) by \( \text{ope2} \) followed by \( \text{ope2} \) under the condition \([\text{cond}]\) (see Figure 3).

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

**Figure 4. PurchasePSM.2**

The PSM \text{PurchasePSM.2}, given Figure 4, corresponds to the application of the operator \text{refine}_by\_sequences() on the transition \text{identifyTerminal} which substitutes \text{identifyTerminal} by \text{readCertificate(term_id)} and \text{acceptTerminal}. Figure 4 shows also the modifications of the interface associated to \text{PurchasePSM}. A new attribute \text{card_id} is added to authenticate a terminal by exchange of certificates.

4.2. Introducing complementary behaviors

When looking at the transition \text{acceptTerminal} between the states \text{CertificateRead} and \text{TerminalAccepted} on Figure 2, we remark that all the possible cases are not considered. The case where a valid terminal certificate is read, expressed by the precondition \([\text{card_id} = \text{term_id}]\), is the only one to be taken into account. What happens when \text{term_id} is not a valid certificate? This new requirement involves the introduction of a new transition and a new state.

The operator \text{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 \text{complement}_transition() adds new functionalities, \text{PluginConformance} is preserved.

Applying the complementary_transition() operator on the state \text{CertificateRead} leads to a new PSM \text{PurchasePSM.3} shown Figure 6. A choice pseudostate and a new state \text{TerminalRefused} are introduced.

**Figure 5. complement_transition()**

**Figure 6. PurchasePSM.3**

A choice pseudostate and a new state \text{TerminalRefused} are introduced jointly with a transition from the \text{TerminalRefused} state to the new exit point using basic construction operators \text{add_vertex()} and \text{add_transition()} defined in [14]. Then, \text{PluginConformance} is preserved.

4.3. Reusing \text{refine}_by\_sequences()

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

**Figure 7. PurchasePSM.3.1**

**Figure 8. PurchasePSM.4**

The previous operator \text{refine}_by\_sequences() is applied again to obtain a new PSM \text{PurchasePSM.4} given Figure 8.
4.4. Introducing conditional behaviors

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:

\[
\text{ope}() := \text{if } [\text{cond}] \text{ then } \text{ope1}() \text{ else } \text{ope2}() \text{ [post2]}
\]

Figure 9 illustrates this operator. It preserves the ExactConformance when the following obligation proofs are satisfied:

- \((\text{pre@pre and cond@pre and post1}) \implies \text{post})\)
- \((\text{pre@pre and not cond@pre and post2}) \implies \text{post})\)

The application of this operator to the state PurchaseAchieved gives two new states BalanceDebited and PurchaseCanceled to illustrate the difference.

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.

When applying once again the split_state() operator to the exit pseudostate, we obtain the FSM PurchasePSM.5 given Figure 13.

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.

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

A. Incremental development of PurchasePSM

```
identifyTerminal() :=
    readCertificate(term_id) ;
    [card_id = term_id]
acceptTerminal()

refine_by_sequences()

complement_transition()
```

```
<<interface>>
PurchaseService

<<refines>>

<<interface>>
PurchaseService_2

identifyTerminal() :=
    readCertificate(term_id) ;
    [card_id = term_id]
acceptTerminal()

<<refines>>

<<interface>>
PurchaseService_3
```

```
<<PartialConformance>>

<<PluginConformance>>

refine_by_sequences()

complement_transition()
```