UML Protocol State Machines Incremental Construction: a Conformance-based Refinement Approach

Olivier Gout Thomas Lambolais
LGI^{2}P
École des mines d'Alès
Nîmes 30035 France
{Olivier.Gout, Thomas.Lambolais}@ema.fr
Submitted to SEFM05, LGI2P Research Report RR05/027

Abstract

This article introduces formal relations between UML Protocol State Machines in order to support their incremental construction. This is based on the use of conformance relations*. Protocol State Machines (*ProSMs*) introduced in UML2.0 are means to model external behaviors of objects and/or components in a service oriented way. These machines are specializations of generic UML State Machines without actions nor activities. In previous works, we adapted Labeled Transition System conformance relations to UML usual State Machines. Going further,* ProSMs *allow us to be more formal thanks to the presence of logical propositions such as state invariants, pre and post-conditions. Hence, we define a conformance relation as a specialization of the informal notion of Protocol Conformance of UML2.0 and we illustrate it on an example.*

1. Introduction

Our works deal with the construction processes in the early stages of software intensive system development. Such steps are: requirements engineering, analysis (abstract system specification and environment modeling) and abstract design. By the latter, we mean architectural steps but not detailed design and programming aspects.

Current software engineering practices usually imply the use of the UML notation de facto standard [8]. For our purposes we are concerned with dynamic models and especially UML state machines [8]. Hence, we address the problem of "UML state machines development", and in particular what we shall call "*abstract* UML state machines", *i.e.* first and simplified UML behavioral models. Basically, in a top-down process, steps go from abstract to concrete and precise state machines. On the opposite, in a bottom-up approach, development steps go from detailed to general state machines. Since we want our approach to be a pragmatic one, and we want to take into account current practices in software engineering, the development framework we are going to settle up should be able to describe both kinds of approaches: bottom-up, top-down, and even hybrid approaches.

In the category of "theoretical" top-down approaches, we may classify pure refinement approaches like the "B method" [1] in the case of the B language. This kind of approach is theoretically appealing, but we find it too constraining: the first steps must be very abstract and global, for them to implicitly include forthcoming details. Our goal is not to convert the B method for the UML language. Nevertheless, we are inspired by the refinement relation: we like the idea of a process "adding details, but preserving previous properties".

In the category of "pragmatic" approaches, we may find classical "extension" approaches in which a new version *is a* specialization of a former version, with new capabilities and responsibilities, such as new methods in specialized classes. Such *incremental* construction processes are also appealing, but a risk appears: are new versions compatible with previous ones? In other words, are new capabilities (extension points) preserving former ones?

Conformance between development steps tackles both aspects of *reduction* (*i.e.* refinement and property preservation) and *extension*: It combines both advantages of theory and practice. It has been initially proposed in the context of service and telecommunication protocols [4], to define the notion of 'conformance tests'. Afterwards, formal relations named conf, ioconf and ioco have been proposed for formal languages like process algebras based on *Labeled Transition Systems* [11].

Our point of view is to use such relations in the context of incremental development processes. In previous works,

we have defined conformance relations for sequential UML state machines. In this paper, the address the case of UML protocol state machines. The paper is structured as follows. The Section 2 introduces UML Protocol States Machines and shows an example. In Section 3, we informally recall previous conformance definitions over LTS and UML State Machines. They are the background for new conformance relation definitions given in Section 4. This relation is illustrated on an example in Section 5.

2. Protocol State Machines

In UML, State Machines are used to model the global behavior of objects. These models capture both internal and observable behaviors and do not necessarily make a distinction between the various orthogonal services offered by the object. This distinction can be done thanks to Protocol State Machines (ProSMs in the sequel) a new formalism proposed in UML2.0. An object can have several ProSMs for the different services it provides. These ProSMs can be associated to *interfaces* or to *ports* to specify how others objects (or components) have to proceed to use a service. In other words, a ProSM defines at any time which operation (or method) is enable and which is not, for a specified service.

2.1. Differences between ProSMs and SMs

According to the UML2.0 specification [9], ProSMs are specializations of State Machines. Consequently, ProSMs are similar to State Machines and can have the same characteristics: concurrency, hierarchy, event policy, etc. Nevertheless, ProSMs impose some restrictions on states and transitions.

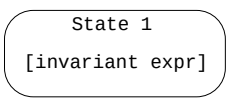
States. Changes in states from State Machines to ProSMs are threefold:

1. A state in a ProSM cannot have *actions* nor *activities*, so `entry`, `exit` and `do` features no longer exist in ProSMs.

2. Restriction on *deep history* and *shallow history* pseudo states: their use is no longer possible in ProSMs.

3. A state invariant typed `Constraint` which specifies a condition that is always true when the state is active.

Notation:



Transitions. Transitions are slightly different. As in states, actions have been suppressed and replaced by a post-condition that must be verified once the transition is triggered.
Notation:

$$\texttt{State}_1 \xrightarrow{\texttt{[precondition]event/[postcondition]}} \texttt{State}_2$$

The `precondition` expression has the same semantics as the `guards` in generic states machines. Considering the purpose of ProSMs, events in transitions are, most of the time, `CallEvents` corresponding to method-calls on the object. Other event types such as `TimeEvent` or `ChangeEvent` are also possible.

2.2. Example: Lift Control Service

An example of how ProSMs can be used to model different uses or scenario of an elevator control system is shown on figure 1.

The main classes (cf. figure 1) are `ElevatorSystem` that captures actions and features of the lift cabin and the `Passenger` class. These two classes communicate through the `GenericElevatorService` interface. The `Passenger` class *uses* this interface and this connection is represented by a circle and a half-circle in the diagram.

A Protocol State Machine is associated to this interface. The service offered is quite simple and a typical usage corresponds to the following sequence sketch:

$$\textit{Calling} \rightarrow \textit{Waiting} \rightarrow \textit{Getting in} \downarrow \textit{Getting out} \leftarrow \textit{Choosing a destination}$$

This simple ProSM is represented in figure 2. ProSM is rather simple, It must not be considered as an advanced specification of the elevator service. Note that UML2.0 ProSM standard definition [9] does not give any example of ProSM at the moment, as well as the current literature which is still very poor on the subject [7].

As specified in the class diagram, the passenger can only interact with the `ElevatorSystem` through three methods:

- `callTo(floor_number: int)` models the way to call for the lift.
- `goTo(floor_number: int)` models the way to indicate, once inside the lift, which floor to go.
- `openDoorRequest()` models the way to open the door.

All these methods are called through `CallEvents` in the ProSM. Other events are `ChangeEvent`

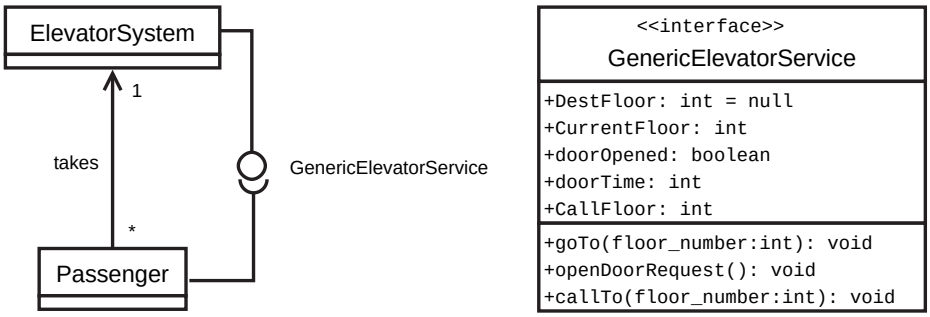


Figure 1. Elevator class diagram

or `TimeEvent` whose syntax are, respectively, `when(`⟨`condition`⟩`)` and `after(`⟨`time expression`⟩`)`. All the pre/post-conditions and invariants must be expressed using the attribute variables defined in the interface. The first state is the waiting state, the elevator door is closed. Once the `Passenger` activates the `CallTo` transition the elevator is informed of his call. This is ensured by the postcondition of the transition. Then, when the lift is at the desired floor (`CurrentFloor=CallFloor`) the `Passenger` is able to get in (`doorOpened`). During this state, the `Passenger` can activate the `goTo` transition to indicate his destination. After some duration (`doorTime`), the elevator closes the door and is ready to move. If the `Passenger` has not set his destination at this point, he can do a call to `goTo()` or request the door opening (`openDoorRequest()`). If the destination was already set or once he has set it (`DestFloor!=null`), the lift takes him to his destination and opens the door (`[DestFloor=CurrentFloor && doorOpened]`). We refer to figure 2 for more details.

3. Conformance background

Evaluation and *correction* means at each development step are the base of a rigorous approach. In the case of *generic* UML state machines (abbreviated SM in the sequel), evaluation means may be: simulation or animation, internal and external consistency checking (by external, we mean with other UML diagrams), property verification, code generation...

Unfortunately, current UML tools do not offer many state machine evaluation mechanisms. Even basic simulation mechanisms are very rarely proposed.

The simple SM construction process we advocate is based on a simple 'good sense' cycle: design, evaluation, correction,... For our purpose, the evaluation means we study is the conformance evaluation between development steps. Informally, such kind of relation checks that a 'refinement' *must accept* all what the abstract step *must accept*.

3.1. Conformance relations for Labeled Transition Systems

Conformance relations used in test generation derive from the following idea:

A realization I is a conform implementation of a specification S *if* all entries that S must accept must also be accepted by I. Equivally (since the negation of 'S must accept a' is 'S may refuse a') "all entries that I may refuse, after a trace of S, may also be refused by S".

This definition, using refusal sets of actions, is the one proposed by Tretmans and Brinksma in 1992 [10, 11] to formalize the ISO definition of conformance testing. Unfortunately, in practice, if I is a black-box implementation of S, it may be impossible to test the refusal of actions. Hence, other more practical conformance relations have be defined, such as input-output conformance ('ioco'). These relations have been defined on IOTS (Input Output Transition Systems). Basically, an implementation I is a conform realization of S if the output set of I after a trace s of inputs defined in I is included in the output set of S after s.

These relations have been successfully used in test generation tools. For our purpose, we have been studying an adaptation of such relations for UML State Machines.

3.2. Conformance Relation for *generic* State Machines

In previous works [2, 6], we proposed two conformance relations intended to be used in an incremental development of generic states machines. These relations are named 'eco' and 'aco' for *Event Conformance* and *Action Conformance*. Their adaptation to UML states machines has led us to consider non-concurrent state machines and to introduce nondeterminism capability and a notion of observability for events and actions.

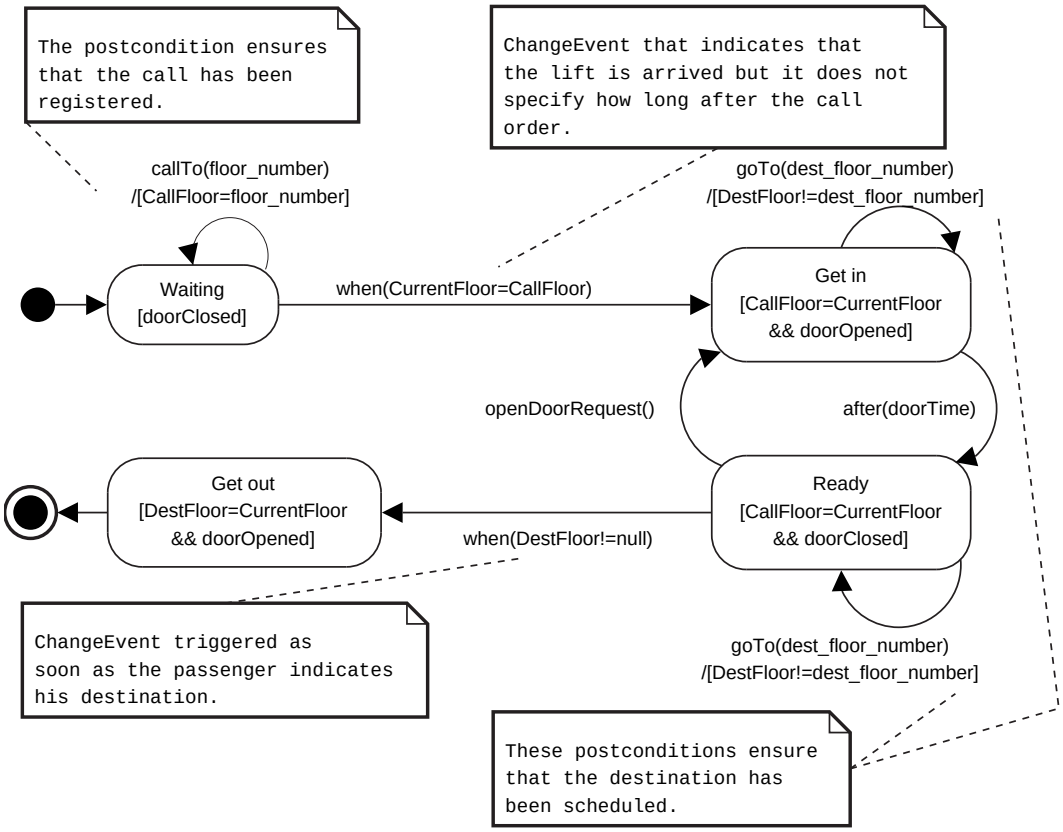


Figure 2. GenericElevatorService Protocol State Machine

Non determinism We considered non deterministic State Machines. Harel's Statecharts can be nondeterministic [3], but usually, UML State Machines are treated like operational and deterministic behavioral models. At the end of the construction process, we aim at obtaining deterministic models; but for the first abstract steps, non determinism is a very expressive mechanism.

Observability In UML State Machines, actions, events and activities are not explicitly said to be observable nor internal. Conformance relations, like simulation relations, compare two models from an external point of view: what are the external/visible interactions offered by a machine? What are the visible results of internal treatments? As an answer, we consider change events, time events and completion events like internal events. Call events and signal events have to be tagged as either internal or observable by the specifier.

Event conformance The event conformance relation is close to the initial "conf" relation defined by Tretmans [10]. In State Machines, transition triggers are events and not actions, so that we adapted initial definition in the following way. Informally, a realization R is event-conform with a specification S, written $R \ \mathrm{eco}\ S$, if all the observable events that S *must accept* after s_e, s_e being any trace of events of S (*i.e.* partial sequence of observable events), *must also be accepted* by R after s_e.

We refer to [2] for a more formal definition.

Action conformance Action conformance is inspired from input-output conformance "ioco" also defined by Tretmans [12]. A realization R is action-conform with a specification S, written $R \ \mathrm{aco}\ S$, if after any trace of events s_e, the set of observable actions that R *may* perform is included into the set of observable actions that S *may* perform after s_e. This relation is not fully formal because it uses the notion of action inclusion that is not yet defined in UML.

4. Conformance Relation for ProSMs

UML meta-model context. A relation of conformance between two ProSMs is already presented in the UML2.0 notation. Unfortunately, this relation remains informal because of its very general definition. The `ProtocolConformance` in UML must be explic-

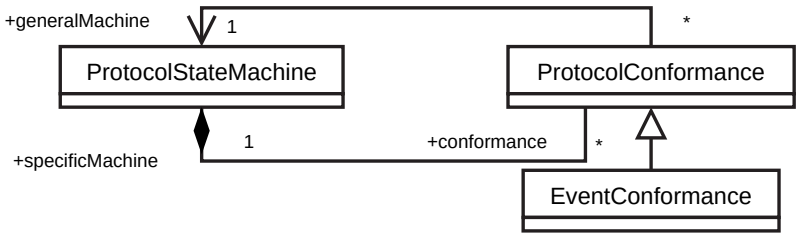


Figure 3. ProtocolConformance specialization

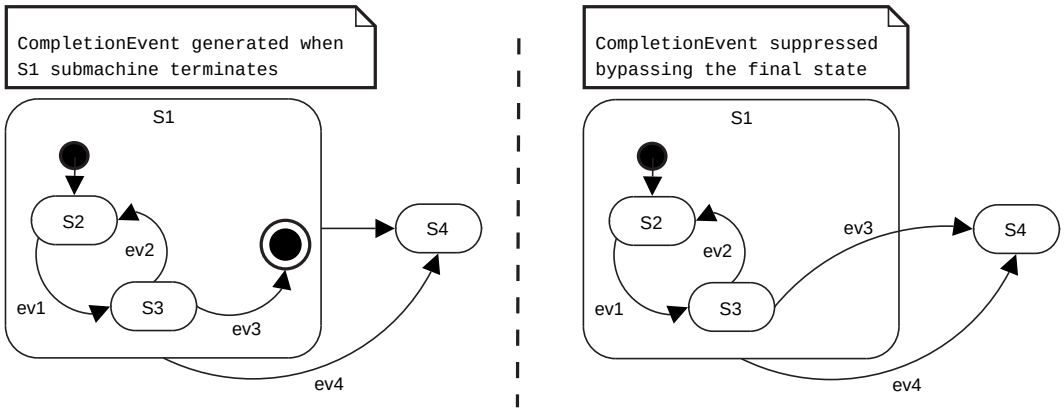


Figure 4. `CompletionEvent` **automatic generation example and transformation to observable model.**

itly declared by the ProSM that want to conforms with a more abstract ProSM. The semantics of the `ProtocolConformance` is based on the conservation of the rules (pre/post-conditions, invariants) of the general ProSM in the specific one but the way to maintain these properties is not addressed. Nevertheless, this relation is a good starting point to define a more refined relation. So, our relation is defined as a specialization of the `ProtocolConformance` and according to the UML notation, can be implemented in the UML meta model as described in figure 3.

As introduced in Section 2, ProSMs specify the ways of using a service offered by an object but not the ways of how the object realize this service. Considering this we must clarify some properties of ProSMs before defining precisely a conformance relation.

4.1. Properties of ProSMs

Observability. The observability notion is interpreted differently in ProSMs than in usual SMs. ProSMs describe explicitly *observable* behaviors where state represent "an exposed stable situation" [9]. Contrary to SMs which contains internal behaviors that model how services are realized, ProSMs do not contain internal behaviors. However, there is one kind of events that cannot be observed: the `CompletionEvent`. A `CompletionEvent` is generated each time an activity is completed. Even when there is no activity in ProSMs, completion rules apply to submachine with final state. Thus a transition without event specified can be fired when arriving in a final state (see example in figure 4).

This kind of transition (without event) is not acceptable for our study. A simple way to avoid this is to transform such ProSMs. Without lost of expressivity, we can bypass the final state by redirecting the transitions to final state to the state reached by the completion transition (see figure 4). All other event kinds are considered as observable (even `ChangeEvent`).

Nondeterminism. As for generic states machines, ProSMs have to be deterministic models. For identical reasons as exposed in Section 3.2 we consider that ProSMs may be non deterministic. Non determinism is the fact that with same initial conditions, two identical executions (an execution starts from an initial state) lead to different results. Since we assume that all events in ProSMs are observable, the only kind of nondeterminism is the observable nondeterminism (for example: a state with two outgoing transitions with the same label).

Concurrency and Composite States. Concurrency in ProSMs is authorized and modeled using orthogonal composite states, but some semantic flaws remain. At first, for sake of simplicity, we shall consider non concurrent ProSMs.

Hence, without concurrent regions and history pseudo-states, composite states are just a useful syntactic sugar. They are a means to factorize transitions and keep diagrams human readable but do not provide additional expressivity. Consequently, we do not treat composite states, considering that they can be automatically transformed in flat state machines distributing transitions to nested states (see figure 5). In the same manner, the invariant of the composite state is distributed to the internal states in conjunction with their proper invariant.

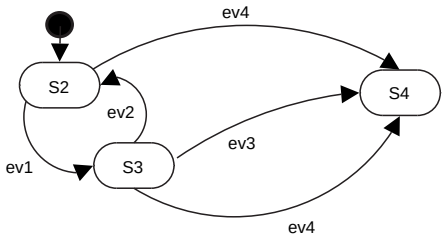


Figure 5. A flat state machine semantically equivalent to the one in Fig.4

This consideration will help us to express ProSMs in a simpler formalism.

Running context. An important statement is that we consider running ProSMs [1]. This means that we are able to determine the active states of the ProSM under study. We do not address here a semantics of execution of states machines. Thus, we assume the run-to-completion step [9] semantics for event processing. This means that we are able to determine the active state after a trace once the last transition have been trigerred and before a new event have been dispatched. In our definitions of flat ProSMs, a running state machine have exactly one active state.

4.2. Definitions

Having considered all the previous properties, we can now introduce a protocol State Machines semantic interpretation on transitions systems as follow.

Definition 4.2.1 (Protocol state machine interpretation) *A* ProSM P *is interpreted as a tuple* $\langle \mathcal{S}, \mathcal{S}_{is}, \mathscr{L}, \longrightarrow, s_A \rangle$ *where:*

- $\mathcal{S}_{is}$ *is the set of initial states;*
- $\mathcal{S}$ *is a set of states defined hereafter;*
- $\mathscr{L}$ *is a set of labels defined hereafter;*
- $\longrightarrow \subseteq \mathcal{S} \times \mathscr{L} \times \mathcal{S}$ *is the set of transitions. If* $(P, \alpha, Q) \in \longrightarrow$, *we write* $P \xrightarrow{\alpha} Q$;
- s_A *is the active state in the running context.*

Definition 4.2.2 (States, labels)

The set $\mathcal{S}$ *is a set of state labels,* $\mathcal{S} = (\mathrm{Names} \times \mathrm{Invariant}) \cup \mathrm{Names}$, *where*

- Names *is the set of states names available in* P;
- $\mathrm{Invariant}$ *is the set of state invariant expressions of* P. *Jointly we define* $\mathrm{Inv} \colon \mathcal{S} \to \mathrm{Invariant}$ *the function that for a state* s *associates its invariant.*

The set $\mathscr{L}$ *is a set of labels,* $\mathscr{L} = (Pre \times Ev \times Post) \cup (Pre \times Ev) \cup (Ev \times Post) \cup Ev$, *where*

- Pre *is the set of preconditions of transitions in* P;
- $Post$ *is the set of postconditions of transitions in* P;
- Ev *is the set of events of* P. *This set cannot be empty.*

The notion of classical observable transition of LTS can be extended to the state machines as follows:

Definition 4.2.3 (Transition) *Let* $\alpha \in L$ *be a label and* P *be a* ProSM *such that* $P = \langle \mathcal{S}, \mathcal{S}_{is}, \mathscr{L}, \longrightarrow, s_A \rangle$. *The relation* $\Longrightarrow$ *is defined on* ProSMs, *writing* $P \stackrel{\alpha}{\Longrightarrow}$ *if* $s_A \xrightarrow{\alpha}$.

Precisely, if $P = \langle \mathcal{S}, \mathcal{S}_{is}, \mathscr{L}, \longrightarrow, s_A \rangle$ *and* $P \stackrel{\alpha}{\Longrightarrow} P'$ *then* $s_A \xrightarrow{\alpha} s'_A$ *and* $P' = \langle \mathcal{S}, \mathcal{S}_{is}, \mathscr{L}, \longrightarrow, s'_A \rangle$.

This notation is extended to path transitions $\sigma = \alpha_1; \ldots; \alpha_n$, *writing* $P \stackrel{\sigma}{\Longrightarrow} P'$ *if there exist* $P_1, \ldots, P_n$ *such that* $P \stackrel{\alpha_1}{\Longrightarrow} P_1 \stackrel{\alpha_2}{\Longrightarrow} \cdots \stackrel{\alpha_{n-1}}{\Longrightarrow} P_{n-1} \stackrel{\alpha_n}{\Longrightarrow} P'$.

We define event trace sets as follows:

Definition 4.2.4 (Events traces, Refusal) *Let* P *be a Protocol State Machine and* $\sigma \in \mathrm{EvTraces}(P)$ *be an event trace.* $\mathrm{EvTraces}(P)$ *is the event traces set and* $\mathrm{ref}(P, \sigma)$ *is the refusal set defined as follows:*

- $\mathrm{EvTraces}(P) =_{\mathrm{def}} \{\sigma \in Ev^* \mid P \stackrel{\sigma}{\Longrightarrow}\}$
- $\mathrm{ref}(P, \sigma) =_{\mathrm{def}} \{ev \in Ev \mid \forall P', P \stackrel{\sigma}{\Longrightarrow} P' \stackrel{ev}{\not\Longrightarrow}\}$

The concept of refusal in a state corresponds to the events for which the evolution of the system is not explicitly specified. In fact the events do not make absolutely evolve the system at this time. In a strict sense, in State Machines, there is no refusal: just like the IOTS, in any state, the responses of the system are specified for any event. But when this answer corresponds to be unaware of a given event, we call it here a refusal.

[1] In UML this concept is meaningless because ProSMs are associated to `interfaces` that can not be instantiated.

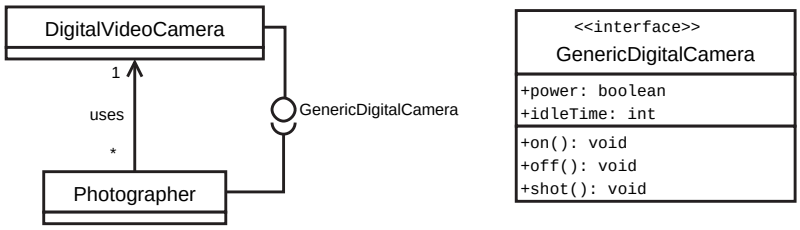


Figure 6. Digital Camera and Photographer system class diagram

Definition 4.2.5 (PreCond, PostCond, Invar) *Let P be a Protocol State Machine and $\sigma \in \mathrm{EvTraces}(P)$ be an event trace.* $\mathrm{PreCond}(P,\sigma)$ *is the set of the* last *preconditions that must have been verified to complete σ;* $\mathrm{PostCond}(P,\sigma)$ *(resp.* $\mathrm{Invar}(P,\sigma)$*) is the set of postconditions (resp. invariants) that must be verified after the trace σ and defined as follows:*

$$\begin{aligned}
\mathrm{PreCond}(P,\sigma) &=_{\mathrm{def}} \{p \in Pre \mid P \xrightarrow{\sigma';[p]ev}, \\
&\qquad with\ \sigma = \sigma';ev\} \\
\mathrm{PostCond}(P,\sigma) &=_{\mathrm{def}} \{p \in Post \mid P \xrightarrow{\sigma';ev/[p]}, \\
&\qquad with\ \sigma = \sigma';ev\} \\
\mathrm{Invar}(P,\sigma) &=_{\mathrm{def}} \mathrm{Inv}(s'_A)\ with\ P \stackrel{\sigma}{\Longrightarrow} P'\ and \\
&\qquad P' = \langle \mathcal{S}, \mathcal{S}_{is}, \mathscr{L}, \longrightarrow, s'_A \rangle
\end{aligned}$$

4.2.1 Conformance relation

These traces, action and refusal sets, guide us to define a conformance relation, in a similar way to that already defined for LTS and IOTS.

Definition 4.2.6 (Event conformance) *Let P and P' be two protocol state machines*

$$P' \ \mathrm{eco}\ P \quad \textit{if,} \quad \textit{for each } \sigma \in \mathrm{EvTraces}(P),$$

$$\begin{array}{lrcl}
 & \mathrm{ref}(P',\sigma) & \subseteq & \mathrm{ref}(P,\sigma) \\
\textit{and} & \mathrm{PreCond}(P,\sigma) & \models & \mathrm{PreCond}(P',\sigma) \\
\textit{and} & \big(\mathrm{PostCond}(P,\sigma) \cup \mathrm{Invar}(P,\sigma)\big) & \models & \mathrm{PostCond}(P',\sigma) \wedge \mathrm{Invar}(P',\sigma)
\end{array}$$

Thus we ensure that when preconditions are strengthened, the resulting postconditions and invariants are also strengthened.

It defines a relation that captures a kind of *behavioral substitutability* between ProSMs. This conformance relation takes into account both events sequences and logical properties (constraints). Verifying this relation is a means to deal with ProSMs refinement and thus to support incremental design of ProSMs.

5. Example

To illustrate the usage of the eco relation in an incremental design process, we propose incremental design process, we propose to study a simple case of incremental st to study a simple case of incremental step. We consider a basic digital camera interface offered by a digital video camera and used by a photographer. A first class diagram is given in figure 6. We consider two attributes:

- `power` is a boolean which indicates the power status of the camera.
- `idleTime` is an integer representing the duration (in seconds for example) of inactivity before automatically entering in idle mode.

5.1 First step

At this stage, the `Photographer` can only call three methods. This first step is very abstract and just models the basic usage/service of a digital camera, typically:

$$Switching\ on \rightarrow Shooting \rightarrow Shooting \rightarrow \ldots \rightarrow Switching\ off$$

A ProSM of this service is proposed in figure 7, with invariants and a timed transition in addition. We call this ProSM $PCam$.

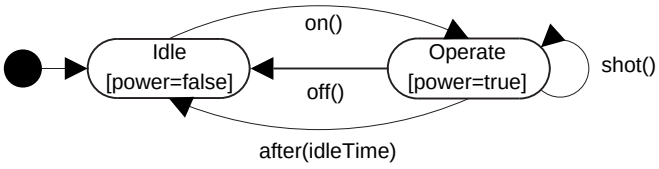


Figure 7. First ProSM **for generic video camera:** $PCam$

5.2 Second step

In this second stage, we precise the definition of the service with new methods and attributes. the resulting class

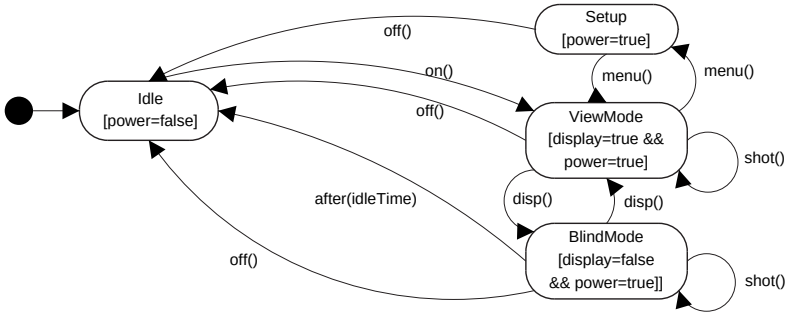


Figure 8. Flat $PCam'$.

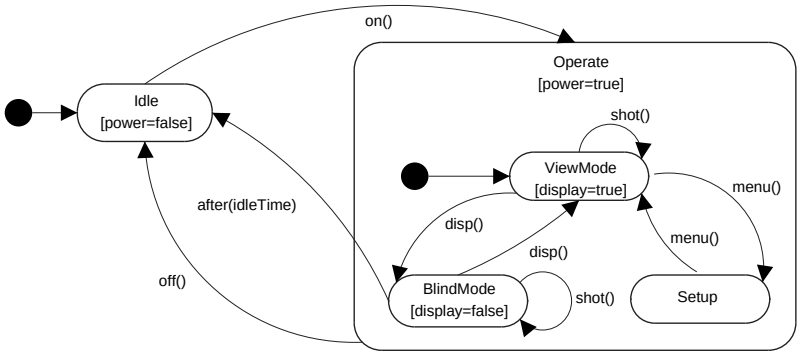


Figure 9. Second Refined ProSM **for generic video camera.**

diagram of the interface is slightly different (see figure 10). We add:

- The attribute `display`, a boolean used to differentiate view mode and blind mode;
- The method `disp()`, to access and exit the setup mode of the camera;
- The method `menu()`, used to activate or deactivate display.

The ProSM $PCam'$ associated is exposed in figure 9. The new `menu()` and `disp()` methods are used to refine the service when the digital camera is operating. The refinement is done by adding nested states and transitions in the `Operate` state.

5.3 Conformance evaluation

First we have to transform $PCam'$ into a flat ProSM. The flat ProSM equivalent to $PCam'$ is given in figure 8.

After that, we have to compute refusal sets of $PCam$ and $PCam'$ on the event traces of $PCam$. To do this we have to determine the traces set of $PCam$. This set is given as a grammar-like expression :

$$\mathrm{EvTraces}(PCam) = \mathrm{Prefix}((\texttt{on}(); \texttt{shot}()^*; \texttt{off}())^*)$$

where $\mathrm{Prefix}(\sigma)), \sigma \in E_v$ is the set of the prefixes of σ trace. Now we can examine refusal sets:

$$\begin{array}{rl} & \mathrm{ref}(PCam', \{\texttt{on}()\}) = \{\texttt{on}(), \texttt{after}(\texttt{idleTime})\} \\ & \mathrm{ref}(PCam, \{\texttt{on}()\}) = \{\texttt{on}()\} \\ \text{so} & \mathrm{ref}(PCam', \{\texttt{on}()\}) \not\subseteq \mathrm{ref}(PCam, \{\texttt{on}()\}) \\ \text{and} & PCam' \not\mathrel{eio} PCam \end{array}$$

As a result, $PCam'$ is not event conform with $PCam$. The reason is that from step1 to step2, the timed transition labeled `after(idleTime)` has been moved from the composite state `Operate` to one of its nested states `BlindMode`. The aim of this move is to model the fact that the auto power-off mode can only be activated when the display is off. Consequently, a `Photographer` designed to use a "step1" `GenericDigitalCamera` may be *deadlocked* in the `ViewMode` because he cannot know how to transit to `BlindMode`. In a finalized product, this kind of modeling error would cause a battery discharge. Generally, when refining, changing the source state of a transition to a nested state of the original source is a wrong refinement. This kind of error is detected with our relation.

6. Conclusion and future work

ProSMs differ sensibly from UML usual State Machines in the sense that they are more abstract (actions and activi-

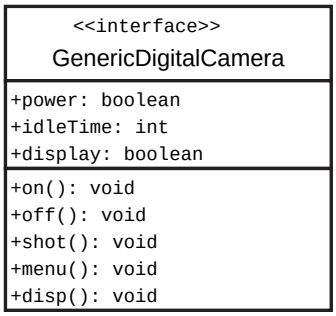


Figure 10. `GenericDigitalCamera` **service: new class diagram.**

ties are not detailed any more) but also that all the described behaviors are observable. Internal transition triggers cannot be used any more, since ProSMs describe the offered services and not a way it could be implemented. In order to define a conformance relation over ProSMs, abstraction is a real advantage, but the absence of internal transitions led us to adopt another point of view of the refinement notion.

For the first stages, we consider fully observable abstract non deterministic ProSMs: the refinement steps concern the reduction of nondeterminism. Indeed, for such stages, nondeterminism is a very convenient means to formally specify a behavior which is not entirely known. Hence, like previous conformance relations, the relation we defined over ProSMs reduces nondeterminism.

There is also a practical advantage of dealing with ProSMs rather than directly with State Machines. The ProSM language is very close to FTS (Fair Transition Systems). FTS are defined with state invariants and logical conditions to trigger transitions. Since several studies and tools have been proposed for FTS [5], we can reasonnabily hope to realize a bridge between UML ProSMs and FTS at short time.

The main differences between ProSMs and FTS consist in the internal transition which can be directly used in FTS, and in the fact that FTS cannot be externally nondeterministic. But refinement relations defined over FTS make a special use of the internal transition. It appears that a first way to translate ProSMs nondeterminism could be to use FTS internal transition.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] O. Gout and T. Lambolais. Construction incrmentale de modles comportementaux UML. In J. Julliand, editor, *AFADL'04, Approches Formelles dans l'Assistance au Dveloppement des Logiciels*, pages 29–42, Besanon, June 2004. Translated as a Research Report ref.[RR05/001].

[3] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts. The Statemate Approach*. Computing McGraw-Hill, 1998. ISBN 0-07-026205-5.

[4] ISO/IEC-JTC1/SC21. Conformance Testing Methodology and Framework, April 1989. DIS 9646.

[5] O. Kouchnarenko and A. Lanoix. Refinement and Verification of Synchronized Component-based Systems — Extended version of the FM'03 article. Research Report, INRIA, June 2003.

[6] T. Lambolais and O. Gout. Using Conformance Relations to Help the Development of State-Machines. In IEEE, editor, *ISSRE'04, International Symposium on Software Reliability Engineering*, Saint-Malo, November 2004.

[7] V. Mencl. Specifying Component Behavior with Port State Machines. In F. de Boer and M. Bonsangue, editors, *Proceedings of the Workshop on the Compositional Verification of UML Models (CVUML, Oct 21, 2003, part of UML 2003)*, volume 101C, pages 129–153. Electronic Notes in Theoretical Computer Science, Nov 2004.

[8] OMG. OMG Unified Modeling Language Specification. OMG formal/03-03-01, March 2003.

[9] OMG. OMG Unified Modeling Language Specification. OMG Adopted Specification ptc/03-08-02, August 2003.

[10] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherland, 1992.

[11] J. Tretmans. Repetitive Quiescence in Implementation and Testing (Extended Abstract). In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme*, number Nr. 315 in GMD-Studien, pages 23–37, St. Augu, 1997. GI/ITG-Fachgespräch, GMD.

[12] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Irelan.