Implementation of the Conformance Relation for Incremental Development of Behavioural Models

Hong-Viet Luong, Thomas Lambolais, and Anne-Lise Courbis

Laboratoire LGI2P, École des Mines d'Alès
Site EERIE, Parc Scientifique Georges Besse
30 035 Nîmes cedex 1, France
{Hong-Viet.Luong,Thomas.Lambolais,Anne-Lise.Courbis}@ema.fr

Abstract. In this paper, we show how to implement the conformance relation on transition systems. The computability of this relation relies on the composition of two operators: the reduction relation whose computability has been proven in our previous work, and the merge function of acceptance graphs associated with transition systems under comparison. It is formally demonstrated, and illustrated through a case study whose analysis is performed by a JAVA prototype we have developed. This research work is developed in order to be applied in a larger context: our goal is to support modelers to develop UML state machine through an incremental modelling method which is able to guarantee that model upgrading does not introduce inconsistencies. Hence, these works lead to a semantics for the specialisation relation between UML State Machines.

1 Introduction

Our area of interest concerns the build-up of UML behavioural models (state machines) according to an incremental approach. Such an approach is natural as it is based on a widely held mental process, because on the one hand, specifying a complex system requires a hierarchical and iterative approach, and on the other hand, initial specification cannot be considered as complete and must be updated all along the modelling process. The problem we address is to compare a model obtained at a given stage with models derived from previous stages: how is it possible to guarantee that updates do not introduce inconsistencies and preserve already modelled functionalities?

Few works deal with this problem in the context of UML modelling, despite the fact that the UML standard [1] is increasingly used in the industry. There are some UML simulation frameworks supporting behavioural model observation, but such intuitive and experience-based evaluations cannot be considered as systematic and reliable. Our goal is therefore to develop formal comparison techniques and tools ensuring consistency between behavioural models, taking into account the distinctive features of incremental modelling. As this problem is not much addressed in the literature [2,3], our first approach is to study tried and tested solutions of domains closely related to state machine modelling. Solutions have been found in works dealing with Labelled Transition Systems (LTS) and precisely on conformance, extension and reduction relations as well as may, must and testing preorders. We have demonstrated in a previous work [4]

K. Czarnecki et al. (Eds.): MoDELS 2008, LNCS 5301, pp. 356–370, 2008.

the extension and reduction relation computability. In this paper, we focus on the computability of the conformance relation. We demonstrate that it can be implemented. The proof is based on the composition of the reduction relation with a merge function applied on the acceptance graphs associated with the compared LTS. Applying this result on UML state machines [5,6] is not trivial and requires a long-term analysis, especially to deal with the complete standard. However, we illustrate the extension relation on a case study modelled both in LTS and its corresponding simplified UML model.

The paper is organised in three parts. At first, we give an overview of the main relations allowing LTS to be compared, and we give definitions useful to understand the demonstration of our theorem. The second part focuses on the conformance computability. For this purpose, we give the definition of acceptance graphs and their merging by reformulating existing works. In part three, we present the JAVA demonstrator we have developed to implement the conformance relation, and we give results obtained on a case study. Lastly, we make a conclusion and present our future work.

2 Definitions and Analysis of Existing Relations

In this part, we give fundamental definitions about Labelled Transition Systems (LTS) [7] and refusal sets [8] to understand the relations we have studied. We present the concept of acceptance sets [9] and some associated results. Before defining these concepts, we informally present existing relations allowing models to be compared at the different stages of the modelling, which should lead us to consider the problem of implementing the conformance relation.

2.1 State of the Art of Relations to Compare Behavioural Models

In the context of incremental development, existing relations defined to analyse and compare LTSs are $\mathbf{conf}$, $\mathbf{red}$, $\mathbf{ext}$ [10,11], as well as *may*, *must* and *testing* preorders [12].

The conformance relation has been formalised by Brinksma and Tretmans [8,11] to represent the notion of conformance between implementations and specifications (or between protocols and services) in telecommunication networks. Initial informal notion was proposed by the ISO standard ISO 9646 [13]. This standard specifies a general methodology for testing the conformance of products to OSI specifications which the products are claimed to implement. Tretmans formal definition translates the property stating that an implementation conforms to its specification, if any test that the specification *must* accept, *must* also be accepted by the implementation. Stated otherwise, any test that the implementation *may* refuse, *may* also be refused by the specification. These notions of "must accept" and "may refuse" tests, or experiments, are very similar to may and must preorders defined by Hennessy [9], except that Hennessy also takes into account the case of divergent processes. Divergent processes are those able to perform infinite internal transition sequences.

It appears that the conformance relation is not transitive. It is well suited to compare implementations with their specifications, but not to be used in refinement sequences. Combined with the notion of trace inclusion, it leads to extension and reduction relations. An extension guarantees the conformance relation with larger traces, whereas a

reduction does the same but with fewer traces. Extension and reduction relations are, then, transitive relations. Moreover, for incremental construction of processes, the extension relation has the following interesting property:

$$S_2 \ \mathbf{ext}\ S_1 \Rightarrow (\forall I.\ I\ \mathbf{conf}\ S_2 \Rightarrow I\ \mathbf{conf}\ S_1) \tag{1}$$

This is the desired property for refinement construction, S_2 being a refinement of S_1. The $\mathbf{ext}$ relation is not the largest refinement relation [10], but it can be used for such purposes. As far as we know, the computability of these relations had not been demonstrated yet. Our first work has been to study a technique for implementing them. We demonstrated through a theorem that $\mathbf{red}$ and $\mathbf{ext}$ can be implemented as simulations between acceptance graphs [4].

The conformance relation and its variant (ioconf, ioco, ...) have been used in test generation and conformance testing tools [11]. However, no implementation of the conformance relation between LTS has ever been proposed. In section 3, we shall present a result which enables us to propose an efficient implementation of this conformance relation.

2.2 Formal Definitions of Conformance Relations

A LTS [7] is a graph consisting of states linked by labelled transitions. It models behavioural specifications as well as implementations.

Definition 1 (Labelled Transition Systems). *A LTS $P = (S, Act, \rightarrow, s_0)$ is a tuple consisting of:*

– a non-empty finite set S of states;
– a set Act of actions;
– a transition relation $\rightarrow \subseteq S \times Act \times S$;
– an initial state $s_0 \in S$.

$Act = L \cup \{\tau\}$ where τ represents any internal, unobservable actions, and L is the set of observable actions.

Before presenting the definitions of conformance relation, we give some usual notations:

$$\begin{aligned}
s \xrightarrow{a} s' &=_{def} (s, a, s') \in \rightarrow \\
s \xrightarrow{a_1 \cdots a_2} s' &=_{def} \exists s_0, \ldots, s_n.\ s = s_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n = s' \\
s \xrightarrow{a_1 \cdots a_2} &=_{def} \exists s'.\ s \xrightarrow{a_1 \cdots a_n} s' \\
s \stackrel{\epsilon}{\Longrightarrow} s' &=_{def} s = s' \text{or } s \xrightarrow{\tau \cdots \tau} s' \\
s \stackrel{a}{\Longrightarrow} s' &=_{def} \exists s_1, s_2.\ s \stackrel{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\epsilon}{\Longrightarrow} s' \\
s \stackrel{a_1 \cdots a_2}{\Longrightarrow} s' &=_{def} \exists s_0, \ldots, s_n.\ s = s_0 \stackrel{a_1}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} s_n = s' \\
s \stackrel{\sigma}{\Longrightarrow} &=_{def} \exists s'.\ s \stackrel{\sigma}{\Longrightarrow} s' \\
s \text{ after } \sigma &=_{def} \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}
\end{aligned}$$

$$P \text{ after } \sigma =_{def} s_0 \text{ after } \sigma$$
$$\text{Traces : } Tr(P) =_{def} \{\sigma \in L^* \mid s_0 \stackrel{\sigma}{\Longrightarrow}\}$$
$$Out(p) =_{def} \{a \in L \mid p \stackrel{a}{\rightarrow}\}$$
$$Out(p, \sigma) =_{def} \cup_{p' \in p \text{ after } \sigma} Out(p')$$
$$Out(P, \sigma) =_{def} Out(s_0, \sigma)$$
$$D(s, a) =_{def} \{s' \mid s \stackrel{a}{\rightarrow} s'\}$$

Definition 2 (Refusal set). *$Ref(P, \sigma)$, the refusal set of P after trace σ, is defined by:*

$$Ref(P, \sigma) =_{def} \left\{X \mid \exists p \in P \textit{ after } \sigma.\ p \stackrel{e}{\not\Longrightarrow}, \forall e \in X\right\}$$

The refusal set is a set $Ref(P, \sigma) \subset \mathcal{P}(L)$. If $\sigma \notin Tr(P)$, $Ref(P, \sigma) = \emptyset$. The conformance relation is defined in the following way:

Definition 3 (Conformance relation $\mathbf{conf}$). *Let P and Q be two LTS,*

$$Q \ \mathbf{conf}\ P \textit{ if } \forall \sigma \in Tr(P).\ Ref(Q, \sigma) \subseteq Ref(P, \sigma).$$

Extension and reduction are defined as extending or reducing traces, while preserving the conformance.

Definition 4 (Reduction relation $\mathbf{red}$). *Let P and Q be two LTS,*

$$Q \ \mathbf{red}\ P \textit{ if } Tr(Q) \subseteq Tr(P) \textit{ and } Q \ \mathbf{conf}\ P.$$

Definition 5 (Extension relation $\mathbf{ext}$). *Let P and Q be two LTS,*

$$Q \ \mathbf{ext}\ P \textit{ if } Tr(P) \subseteq Tr(Q) \textit{ and } Q \ \mathbf{conf}\ P.$$

The relation $\mathbf{conf}$ is not a preorder relation: $\mathbf{conf}$ has not the transitivity property. But $\mathbf{red}$ and $\mathbf{ext}$ are reflexive and transitive.

2.3 Acceptance Sets

In this section, we present a definition of acceptance sets and their relation with refusal sets. This notion will be used in the next section to build up acceptance graphs associated to LTS.

Definition 6 (Acceptance set). *The acceptance set of P after σ is defined by:*

$$Acc(P, \sigma) = \{X \mid \exists p' \in P \textit{ after } \sigma.\ X = Out(p', \epsilon)\}$$

The acceptance set represents the "sets of possible actions" of a process after a trace. Intuitively, the inclusion of acceptance set allows us to check whether a process is more deterministic than another.

Definition 7 (Set of sets inclusion). *Let $A, B \subseteq 2^{Act}$. $A \subset\subset B$ if:*

$$\forall S \in A.\ \exists S' \in B.\ S' \subseteq S.$$

The following theorem has been stated in [4].

Theorem 1. $\forall \sigma \in Tr(Q).\ Acc(P, \sigma) \subset\subset Acc(Q, \sigma) \Leftrightarrow Ref(P, \sigma) \subseteq Ref(Q, \sigma)$.

3 The Conformance Relation Computability

In this section, we present the definition of merging acceptance graphs [14] and demonstrate that the **conf** relation can be calculated through the merging acceptance graphs. Let us recall the definition of acceptance graphs.

3.1 Acceptance Graphs

An acceptance graph is the deterministic transition system corresponding to a LTS, where states are associated to their acceptance sets. Examples of acceptance graphs automatically generated by our JAVA prototype from LTSs are given in section 4.2, Figure 5.

Definition 8 (Acceptance graph). *$\mathcal{A}(P) = \langle T, Act, \rightarrow_T, t_0 \rangle$ of the LTS P is a tuple where:*

- *T is the set of states. $T = \{Q \in 2^S \mid Q = Q^\epsilon\}$;*
- *$\rightarrow_T$ is the set of transitions;*
- *For $t \in T$, we define the acceptance set $t.acc = \{X \mid X = Out(q, \epsilon) \wedge q \in t\}$;*
- *For $A \in t_1.acc$, $a \in A \Rightarrow \exists t_2 \in T$ such that $t_1 \xrightarrow{a}_T t_2$;*
- *$t_0 = (\{s_0\})^\epsilon$.*

In this definition, the ϵ-closure is defined as follows:

Definition 9 (ϵ-closure). *The ϵ-closure of a set of states Q is:*

$$Q^\epsilon = \{p \mid \exists q \in Q.\ q \stackrel{\epsilon}{\Longrightarrow} p\}$$

This definition of acceptance graphs is similar to acceptance graphs of [12], except that acceptance sets do not take into account divergence states. It is also similar to the definition of acceptance graphs of Khendek [14], but Khendek uses a different definition of acceptance sets which is:

$$\begin{aligned} Acc(P, \sigma) &= \{Out(P, \sigma) - X \mid X \in Ref(P, \sigma)\} \\ &= \{X \mid \exists p' \in P \stackrel{\sigma}{\Longrightarrow} .\ Out(p') \subseteq X \subseteq Out(P, \sigma)\} \end{aligned}$$

The algorithm of acceptance graph construction introduced by [12] can be adapted to the construction of acceptance graphs as defined in this paper.

3.2 Merging Acceptance Graphs

The definition of merging acceptance graphs is introduced by Khendek [14]. The merging operation is defined as follows:

Definition 10 (Merge). *Let P and Q be two LTS, and $\mathcal{A}(P)$ and $\mathcal{A}(Q)$ their acceptance graphs:*

$$\mathcal{A}(P) = (T_1, Act, \rightarrow_{T_1}, t_{1_0})$$
$$\mathcal{A}(Q) = (T_2, Act, \rightarrow_{T_2}, t_{2_0})$$

The merging graph of the two above acceptance graphs is defined as follows:
$\text{Merge}(\mathcal{A}(P), \mathcal{A}(Q)) = (T_3, Act, \rightarrow_{T_3}, \langle t_{1_0}, t_{2_0} \rangle)$

1. $T_3 = T_1 \times T_2 \cup T_1 \cup T_2$
2. *For each state* $t_{3_i} \in T_3$,
   if $t_{3_i} = \langle t_{1_i}, t_{2_j} \rangle$ *then* $t_{3_i}.acc = \{X_1 \cup X_2 \mid X_1 \in t_{1_i}.acc \wedge X_2 \in t_{2_j}.acc\}$,
   if $t_{3_i} \in T_x$ *then* $t_{3_i}.acc = t_{x_i}.acc$ *where* $x = 1, 2$.
3. *For each* $\langle t_{1_j}, t_{2_k} \rangle \in T_3$
   - $\langle t_{1_j}, t_{2_k} \rangle \xrightarrow{a}_{T_3} \langle t_{1_l}, t_{2_m} \rangle$ *if* $t_{1_j} \xrightarrow{a}_{T_1} t_{1_l} \wedge t_{2_k} \xrightarrow{a}_{T_2} t_{2_m}$
   - $\langle t_{1_j}, t_{2_k} \rangle \xrightarrow{a}_{T_3} \langle t_{1_0}, t_{2_0} \rangle$ *if* $(t_{1_j} \xrightarrow{a}_{T_1} t_{1_0} \wedge t_{2_k} \overset{a}{\not\rightarrow}_{T_2}) \vee (t_{1_j} \overset{a}{\not\rightarrow}_{T_1} \wedge t_{2_k} \xrightarrow{a}_{T_2} t_{2_0})$
   - $\langle t_{1_j}, t_{2_k} \rangle \xrightarrow{a}_{T_3} t_{1_l}$ *if* $t_{1_j} \xrightarrow{a}_{T_1} t_{1_0}, t_{1_l} \neq t_{1_0} \wedge t_{2_k} \overset{a}{\not\rightarrow}_{T_2}$
   - $\langle t_{1_j}, t_{2_k} \rangle \xrightarrow{a}_{T_3} t_{2_m}$ *if* $t_{2_k} \xrightarrow{a}_{T_2} t_{2_m}, t_{2_m} \neq t_{2_0} \wedge t_{1_j} \overset{a}{\not\rightarrow}_{T_1}$
4. *For each state* $t_{x_j} \in T_3$ *where* $x = 1, 2$
   - $t_{x_j} \xrightarrow{a}_{T_3} \langle t_{1_0}, t_{2_0} \rangle$ *if* $t_{x_j} \xrightarrow{a}_{T_x} t_{x_0}$
   - $t_{xj} \xrightarrow{a}_{T_3} t_{x_l}$ *if* $t_{x_j} \xrightarrow{a}_{T_x} t_{x_l}, t_{x_l} \neq t_{x_0}$

The operation Merge has some interesting properties. It is commutative and associative. Moreover, for any acceptance graph, there exists one LTS [14]. We write $\mathrm{Merge}(P, Q)$ the LTS of $\mathrm{Merge}(\mathcal{A}(P), \mathcal{A}(Q))$. The operation Merge guarantees the extension relation between the LTS of merging acceptance graphs and the initial LTS. In addition, the merging graph is always the least common cyclic extension of two initial graphs.

Proposition 1. *Let* P, Q *be two LTS.*

1. $\mathrm{Merge}(P, Q) = \mathrm{Merge}(Q, P)$
2. $\mathrm{Merge}(P, Q)$ **ext** $P \wedge \mathrm{Merge}(P, Q)$ **ext** Q.

3.3 Demonstration of the Conformance Relation Computability

This section gives the demonstration of the conformance relation computability through a reduction relation applied on the merging of acceptance graphs.

Theorem 2. *Let* P *and* Q *be two LTS,*

$$Q \ \mathbf{conf} \ P \Longleftrightarrow Q \ \mathbf{red} \ \mathrm{Merge}(P, Q).$$

Proof.

1. Let us prove ($\Rightarrow$): Q **conf** $P \Rightarrow Q$ **red** $\mathrm{Merge}(P, Q)$

1.1. $\mathrm{Merge}(P, Q)$ **ext** $Q \Rightarrow Tr(Q) \subseteq Tr(\mathrm{Merge}(P, Q))$

1.2. Following the definition of merging graphs:

$$\begin{aligned} &\forall \sigma \in Tr(Q) \wedge \sigma \notin Tr(P).\ Acc(\mathrm{Merge}(P, Q), \sigma) = Acc(Q, \sigma) \\ \Rightarrow\ &\forall \sigma \in Tr(Q) \wedge \sigma \notin Tr(P).\ Acc(Q, \sigma) \subset\subset Acc(\mathrm{Merge}(P, Q), \sigma) \end{aligned} \tag{2}$$

With definition 3 and theorem 1, from Q **conf** P we know that:

$\forall \sigma \in Tr(Q) \cap Tr(P).\ Acc(Q, \sigma) \subset\subset Acc(P, \sigma)$,

Moreover, following definition 10:

$\forall \sigma \in Tr(Q) \cap Tr(P).$
$Acc(\mathrm{Merge}(P, Q)) = \{X_1 \cup X_2 \mid X_1 \in Acc(P, \sigma) \wedge X_2 \in Acc(Q, \sigma)\}$

It can be written as follows:

$$\begin{aligned}
&\forall \sigma \in Tr(Q) \cap Tr(P).\\
&\quad \forall X_2 \in Acc(Q,\sigma).\ \exists X' \in Acc(P,\sigma).\ X' \subseteq X_2\\
&\quad \Rightarrow \forall X_2 \in Acc(Q,\sigma).\ \exists X' \in Acc(P,\sigma).\ X' \cup X_2 \subseteq X_2\\
&\quad \Rightarrow \forall X_2 \in Acc(Q,\sigma).\ \exists X \in Acc(\mathrm{Merge}(P,Q),\sigma), X = X' \cup X_2.\ X \subseteq X_2\\
&\quad \Rightarrow Acc(Q,\sigma) \subset\subset Acc(\mathrm{Merge}(P,Q),\sigma) \qquad (3)
\end{aligned}$$

With (2) and (3), we have: $\forall \sigma \in Tr(Q).\ Acc(Q,\sigma) \subset\subset Acc(\mathrm{Merge}(P,Q),\sigma)$. So, with condition $Tr(Q) \subseteq Tr(\mathrm{Merge}(P,Q))$ proven in 1.1., we have:

Q **conf** $P \Rightarrow Q$ **red** $\mathrm{Merge}(P,Q)$

2. We prove ($\Leftarrow$): Q **red** $\mathrm{Merge}(P,Q) \Rightarrow Q$ **conf** P

Since $\mathbf{conf} = \mathbf{red} \circ \mathbf{ext}$, where $\circ$ is the composition between binary relations [10], we have:

Q **red** $\mathrm{Merge}(P,Q) \wedge \mathrm{Merge}(P,Q)$ **ext** $P \Rightarrow Q$ **conf** P. □

In [4], we have demonstrated that

$$Q \ \mathbf{red}\ P \Longleftrightarrow \mathcal{A}(Q) \lessapprox \mathcal{A}(P),$$

where $\lessapprox$ is a simulation relation defined over acceptance graphs. Hence, theorem 2 leads to

Corollary 1. *Let P and Q be two LTS,*

$$Q \ \mathbf{conf}\ P \Longleftrightarrow \mathcal{A}(Q) \lessapprox \mathrm{Merge}(\mathcal{A}(P), \mathcal{A}(Q)).$$

This theorem and corollary allow us to implement the conformance relation between LTS. The tool we have developed strictly follows this result: it computes first acceptance graphs, then Merge, and then checks whether the reduction is satisfied or not. An example is presented in the next section.

4 Implementation and Results

This part gives an overview of the JAVA prototype we have developed to implement the conformance relation. We present a case study modelling a phone and the different models that may be set up during the incremental modelling approach. The conformance relation is computed to compare an implementation with a given specification. Intermediate steps and results of these computations are given to illustrate the application of theorem 2.

The relations being demonstrated on LTS, computation is obviously performed on LTS. Nevertheless, for every step of modelling, we present the UML state machine associated with LTS for two reasons: using UML model is more widespread than LTS, and this approach gives a quick outline of our future work consisting in applying results of LTS comparison on UML models.

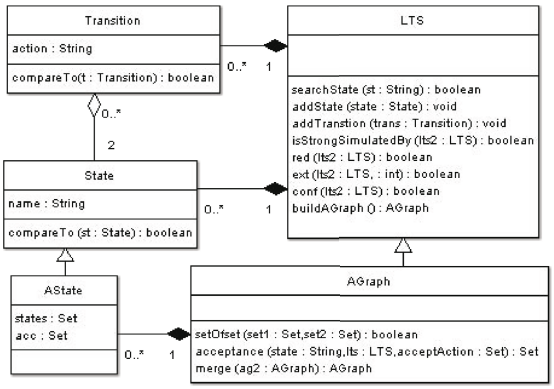


Fig. 1. The class diagram of the prototype

4.1 Implementation of the Conformance Relation

The JAVA prototype we have developed follows the computation approach of theorem 2. Consequently, the main implemented classes are *LTS* and *AGraph* (cf. Figure 1):

- *LTS* class implements a LTS as a set of states derived from the *State* class and transitions derived from the *Transition* class. As demonstrated in [4], relations *red* and *ext* have been implemented in terms of bisimulation (function *isStrongSimulatedBy*).
- *AGraph* class implements the acceptance graph associated with a LTS. It is itself a LTS, the state of which belongs to the *AState* class defined as a subclass of *State* class. The attribute *states* of the class *AState* defines the list of its associated states in the LTS. This attribute allows the relationship between acceptance graph nodes and LTS nodes to be established. Another fundamental attribute associated with a *AState* node is *acc* which is its acceptance set defined has a set of sets of actions.

In order to follow theorem 2, the conformance relation (function *conf*) has been implemented into three steps: build up of acceptance graphs associated with the reference LTS (the specification) and the LTS under analysis (the implementation), their merging into a new acceptance graph and at last, computation of the reduction relation between the acceptance graph of the reference LTS and the merging acceptance graph.

The following sections point out the results obtained by the JAVA prototype and present in details intermediate graphs automatically build up and analysed to demonstrate the conformance.

4.2 Case 1: Modelling a Simple Phone in Two Steps

This case study illustrates how to check the conformance relation between a phone and its specification defined from a user point of view. The phone is modelled to interact with a user, and *simulates* network activities (such as incoming calls) like internal treatments. A UML class diagram presenting phone classes and interfaces is shown in Figure 2. The specification provided interface (class *User_actions*) is composed of

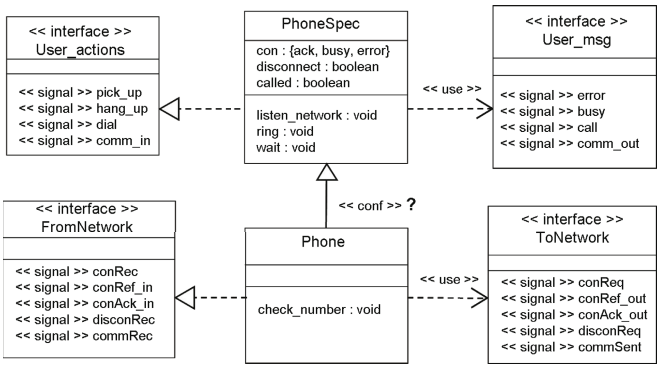


Fig. 2. Phone classes, with provided and required interfaces

signals sent by users: *hang_up*, *pick_up*, *dial* and *comm_in* (incoming communication). The specification required interface (class *User_msg*) is composed of signals sent by the phone: *error*, *busy*, *call* and *comm_out*.

The *Phone* is connected to a network. It uses and provides the same interfaces as its specification, plus network interfaces. Its provided interface is composed of signals sent by the network: connection received, incoming connection refused, incoming connection acknowledge, disconnection received and communication received. Its required interface is composed of signals sent to the network: connection request, outgoing connection refused, outgoing connection acknowledge, disconnection request and communication sent.

In UML, we distinguish between *actions* and *events*. There is no such distinction in LTS for which any transition label is an action. UML events can be signal events, call events, time events, change events and complete events. If the behaviour has to be observed according to a specific point of view, there is no simple means in UML to hide some events and actions. We will consider that UML change events, time events and complete events are internal. We also need to explicitly consider as internal some actions and signal events. UML tags can be used for such purpose.

Figure 3.a represents a UML state machine $SM_{PhoneSpec}$ of the phone specification. There are two functionalities: the user is called (right part of Figure 3.a) or the user is calling (left part of Figure 3.a).

We have not yet fully formalised the UML transformation into LTS, nevertheless main transformation rules can be expressed as follows:

- Change events, time events or complete events are modelled by the silent action τ;
- Call events, signal events and actions are translated into LTS actions;
- Any UML transition labelled by event and action (event / action), is split into two LTS transitions, separated by a state ($s_0 \xrightarrow{e} s_1 \xrightarrow{a} s_2$), where e is an action label which denotes the UML event, and a is an action label which denotes the UML action; if the UML action is not visible, it may be translated into a simple transition $s_0 \xrightarrow{e} s_1$;
- Non visible UML signal events, call events, and actions are modelled by the LTS internal action τ;

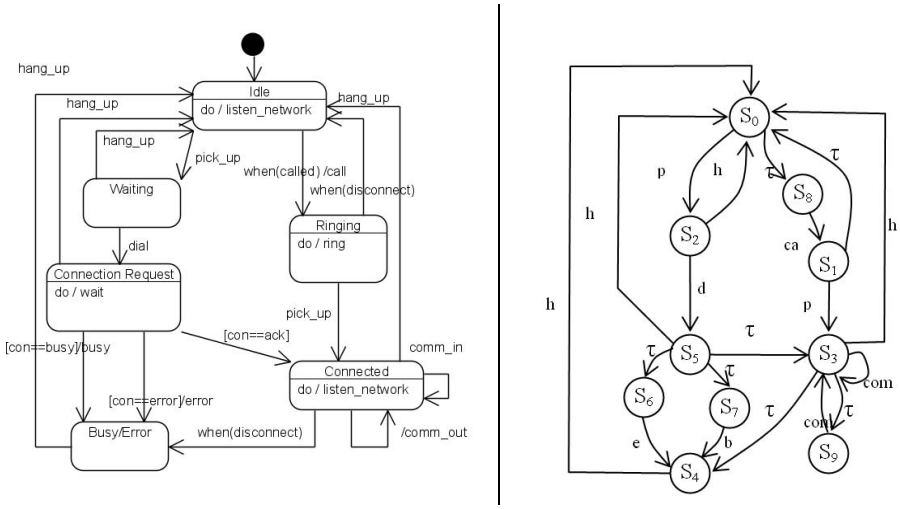


Fig. 3. (a) $SM_{PhoneSpec}$, *PhoneSpec* state machine. (b) $LTS_{PhoneSpec}$, associated LTS.

- UML activities inside states are considered like internal treatments and do not appear in LTS;
- UML hierarchical states are flattened and global transition from this state are distributed to every sub-states.

Note that the UML state machines we consider do not take into account signal or method parameters, neither other variables. Hence, UML guards are not taken into account in corresponding LTS, and systematically lead to nondeterministic transitions.

In order to have legible LTS, actions are named by the first letters of UML labels. UML signals which are both required and provided (such as *comm_in* and *comm_out*) are translated into a single LTS action (*com*). Figure 3.b shows $LTS_{PhoneSpec}$, the LTS associated with the state machine $SM_{PhoneSpec}$.

Let us consider the modelling of the phone state machine (Figure 4.a). Internal activities of $SM_{PhoneSpec}$ are refined: new network incoming and outgoing signals are used to build up new transitions or new sub-machines. For example, in the state *Idle* of $SM_{PhoneSpec}$, the *listen_network* activity and the associated change event $when(called)$, which only aims at detecting incoming calls, is replaced in SM_{Phone} by a transition triggered by the incoming signal event *conRec*.

The question is to verify if there is a conformance relation between the phone and its specification. It is automatically computed by our tool whose main intermediate results are given in details in the following figures. The first step performed by the JAVA prototype is to build up acceptance graphs associated with the two compared LTS and their merge.

The second step consists in verifying the simulation relation between the acceptance graph of LTS_{Phone} (Figure 5) and the merge (Figure 6).

This relation is computed by the JAVA prototype by verifying the simulation relation between the two graphs and the inclusion of acceptance sets. More precisely, two

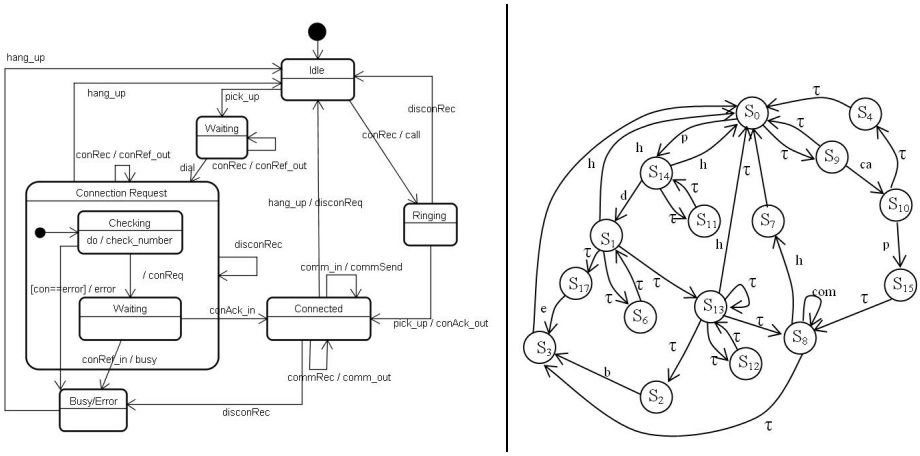
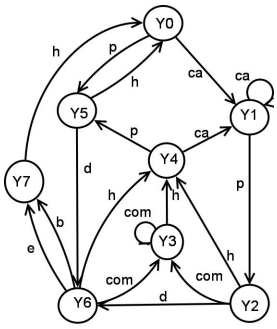


Fig. 4. (a) SM_{Phone}, the state machine of *Phone* class. (b) LTS_{Phone}, associated LTS.



AState	Acceptance set
y_0	$\{\{ca, p\}\}$
y_1	$\{\{ca, p\}\}$
y_2	$\{\{com, h\}, \{d, h\}, \{h\}, \{com\}\}$
y_3	$\{\{com, h\}, \{h\}, \{com\}\}$
y_4	$\{\{ca, p\}\}$
y_5	$\{\{d, h\}\}$
y_6	$\{\{h\}, \{com, h\}, \{e\}, \{b\},$ $\{com, e, b, h\}, \{com, b, h\}, \{com\}\}$
y_7	$\{\{h\}\}$

Fig. 5. $\mathcal{A}(LTS_{Phone})$ graph and its acceptance sets

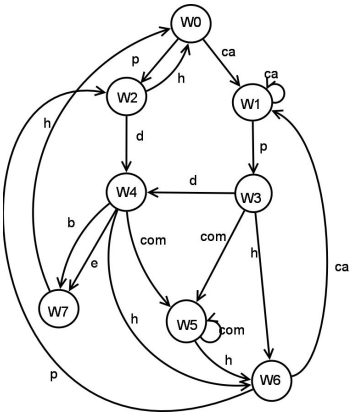
properties are automatically computed. The first one is that there exists, for each node of $\mathcal{A}(LTS_{Phone})$, a node of $\mathcal{A}(\text{Merge}(LTS_{PhoneSpec}, LTS_{Phone}))$ which simulates it.

The result of this first property is expressed by a set of pairs representing simulation relationships. In this case, it is: $\{(y_0, w_0), (y_1, w_1), (y_2, w_3), (y_3, w_5), (y_4, w_6), (y_5, w_2), (y_6, w_6), (y_7, w_7)\}$. The second property is that the acceptance set of each node of $\mathcal{A}(LTS_{Phone})$ is included in the acceptance set of its associated node in $\mathcal{A}(\text{Merge}(LTS_{PhoneSpec}, LTS_{Phone}))$. This property is checked by analysing acceptance sets given in tables of Figure 5 and Figure 6 for each pair belonging to the simulation relation. The conformance relation is therefore verified and the phone is guaranteed to conform to its specification.

Next section deals with an extended specification of the phone.

4.3 Case 2: Modelling a Phone with Double Call

We are interested in modelling a phone able to accept a second call while the user is on the phone. The class *DoubleCallSpec* (Figure 7) is a specialisation of the *PhoneSpec*



AState	Acceptance set
w_0	$\{\{ca, p\}\}$
w_1	$\{\{ca, p\}\}$
w_2	$\{\{d, h\}\}$
w_3	$\{\{d, h\}, \{h\}, \{com\}, \{com, h\}, \{d, com, h\}\}$
w_4	$\{\{e, h\}, \{com, e\}, \{e\}, \{com\}, \{b\},$ $\{com, e, b, h\}, \{com, b\}, \{b, h\}, \{e, b\},$ $\{com, h\}, \{h\}, \{com, b, h\}, \{com, e, h\}\}$
w_5	$\{\{h\}, \{com\}, \{com, h\}\}$
w_6	$\{\{ca, p\}\}$
w_7	$\{\{h\}\}$

Fig. 6. $\mathcal{A}(\mathrm{Merge}(LTS_{PhoneSpec}, LTS_{Phone}))$ graph and its acceptance sets

class. It inherits *PhoneSpec* interfaces, and has its own required interface defining signals accepting or rejecting/stopping the second call. Figure 8.a represents the state machine of this new specification, called $SM_{DoubleCallSpec}$. The difference with the first specification is on state $Connected$ since a $call$ transition may occur on this state. Figure 8.b gives the corresponding LTS, named $LTS_{DoubleCall}$, obtained by the same informal transformation rules as in the previous case. The conformance relation between the phone and the double call specification has been computed according to the same steps as it has been shown in previous section.

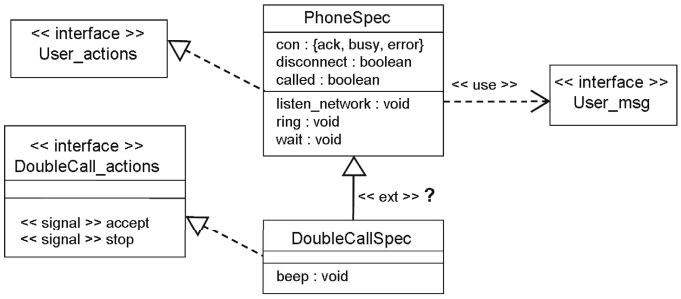
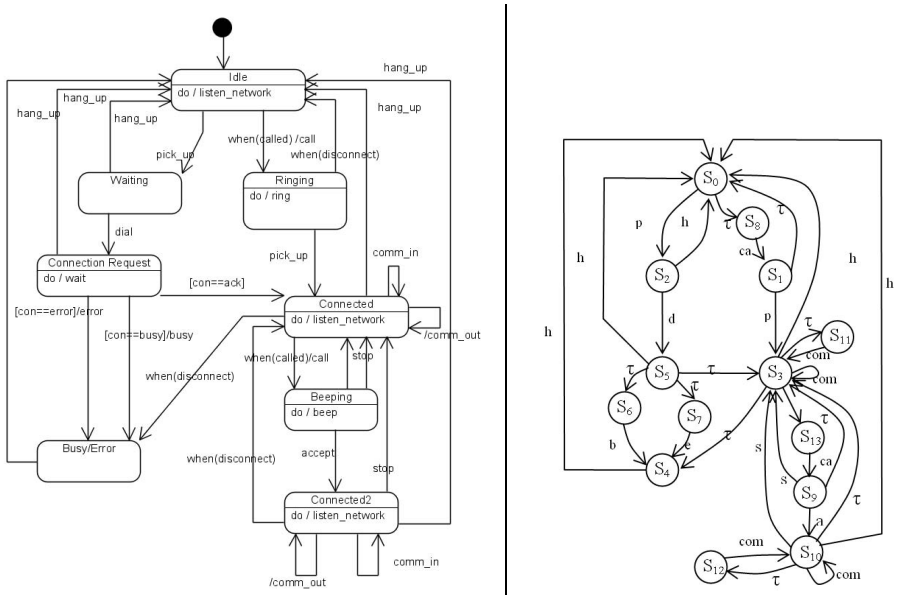


Fig. 7. Double call specification class, with provided and required interfaces

Again, the conformance relation is guaranteed. Moreover, there is an extension relation between the double call specification and the initial specification. Next modelling step consists in setting up the state machine of the implementation (*DoubleCall*) and verifying its conformance with *DoubleCallSpec*. Since the extension relation is a refinement relation (see property (1) in section 2.1), if *DoubleCall* conforms its specification, then it surely conforms to the initial phone specification. These results are summarised in Figure 9.



(a) $SM_{DoubleCallSpec}$, state machine (b) $LTS_{DoubleCallSpec}$, associated LTS

Fig. 8. Telephone with a second call

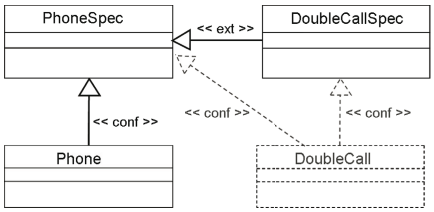


Fig. 9. Relations between phone specifications and their implementations

This case study shows two steps of the development of a state machine, with the second step including a new functionality, at two levels of abstraction (specification and implementation levels). The developed state machine is associated to a single class. Extensions of this class (specialisation or implementation) are then related to extensions of the state machine. For larger systems, we could consider developing state machines describing the behaviour of the whole system, and not only associated to a single class of the system. This would correspond to a *top-down* approach. Then, at a further development step, this would imply a decomposition of the whole state machine into several parts, involving communication between these parts. This approach of refinement through decomposition will constitute a research perspective. In the other direction, according to a *bottom-up* approach, the development of several state machines associated to different classes of the same system will lead us to consider the question of communication and resulting UML architecture involving these machines.

5 Conclusion and Future Work

Currently, the development of behavioural models following an incremental approach is hardly achievable. UML tools supporting state machines lack evaluation procedures, which could allow users to develop models step by step, and modify (change or extend) them according to evaluation results. We believe that a pragmatic development approach, such as ones adopted for program development, could be applied to behavioural modelling. Such an approach follows a simple repetitive cycle: construction, evaluation, correction. The evaluation means that we propose rely on model comparison. In previous works, we have presented how to compute extension and reduction relations and point out the use of the former one to verify model refinement. However, it is not adapted to verify, at the end of an incremental modelling process, if an implementation conforms to a specification.

In this article, we propose an implementation technique for conformance evaluation. The conformance relation defined over labelled transition systems, had never been implemented before. The technique we propose is based on model transformations into acceptance graphs and model merging. We have developed a tool which supports these steps on LTS models. We have proposed informal translation rules from UML State Machines to LTS. Hence, the conformance relation between LTS is a proposal for a semantics of UML state machine specialisation relation.

Our short term objectives are to automate the UML translation into LTS and integrate other state machine features such as concurrent sub-states and pseudo-states (e.g. choice, join and history) in order to be able to compare a large representative panel of state machines. We plan to integrate the extension and conformance relations into a UML CASE tool supporting incremental modelling processes. A complementary area of interest consists in comparing UML state machines with sequence diagram specifications. Such diagrams are largely used in first development steps. This will assist the modeler from early specification steps to detailed behavioural models, taking into account several modelling diagrams. A promising area of investigation, complementary to evaluation techniques, will consist in assisting the user during model construction with development schemes and patterns which would have been proved to preserve refinement or implementation relations.

References

1. OMG: Unified Modeling Language Specification. Object Management Group (2007)
2. Boiten, E., Bujorianu, M.: Exploring UML refinement through unification. In: Jürjens, J., Rumpe, B., France, R., Fernandez, E. (eds.) Critical Systems Development with UML - Proceedings of the UML 2003 workshop. Number TUM-I0323, Technische Universität München, pp. 47–62 (September 2003)
3. Beeck, M.V.D.: Behaviour specifications: Equivalence and refinement notions. Techreport 24/00-I, Universität Münster (November 2000)
4. Luong, H.V., Lambolais, T., Courbis, A.L.: Implementation of extension and reduction relations for incremental development of behavioural models. Technical Report RR-006-, EMA, Laboratoire LGI2P, École des mines d'Al (2008)

5. Lambolais, T., Gout, O.: Using conformance relations to help the development of state-machines. In: ISSRE 2004, International Symposium on Software Reliability Engineering (November 2004)
6. Gout, O.: Développement incrémental de spécifications orientées objets. PhD thesis, École de Mines d'Alès (2006)
7. Milner, R.: Communicating and Mobile Systems: The π Calculus. Cambridge University Press, Cambridge (1999)
8. Brinksma, E., Scollo, G.: Formal Notions of Implementation and Conformance in LOTOS. Technical Report INF-86-13, Dept. of Informatics, Twente University of Technology (1986)
9. Hennessy, M.: Algebraic theory of processes. The Foundations Of Computing. MIT Press, Cambridge (1988) ISBN:0-262-08171-7
10. Leduc, G.: Conformance relation, associated equivalence, and minimum canonical tester in lotos. In: PSTV XI. North-Holland, Amsterdam (1991)
11. Tretmans, J.: Testing concurrent systems: A formal approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664. Springer, Heidelberg (1999)
12. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. Formal Aspects of Computing 3 (1992)
13. ISO/IEC 9646-1: Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts (1991)
14. Khendek, F., Bochmann, G.V.: Merging behavior specifications. Formal Methods in System Design 6, 259–293 (1993)