A state/event-based model-checking approach for the analysis of abstract system properties.

Science of Computer Programming 76 (2011) 119–135

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A state/event-based model-checking approach for the analysis of abstract system properties

Maurice H. ter Beek[a,*], Alessandro Fantechi[a,b], Stefania Gnesi[a], Franco Mazzanti[a]

[a] *Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR Via G. Moruzzi 1, 56124 Pisa, Italy*

[b] *Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze Via S. Marta 3, 50139 Firenze, Italy*

ARTICLE INFO

Article history:
Received 8 June 2009
Accepted 14 July 2010
Available online 30 July 2010

Keywords:
Formal methods
UML
State machines
Temporal logic
Model checking
Automotive systems
Service-oriented computing

ABSTRACT

We present the UMC framework for the formal analysis of concurrent systems specified by collections of UML state machines. The formal model of a system is given by a doubly labelled transition system, and the logic used to specify its properties is the state-based and event-based logic UCTL. UMC is an *on-the-fly* analysis framework which allows the user to interactively explore a UML model, to visualize abstract behavioural slices of it and to perform local model checking of UCTL formulae. An automotive scenario from the service-oriented computing (SOC) domain is used as case study to illustrate our approach.

1. Introduction

Real-life systems are often modelled according to either a state-based or an event-based paradigm. While in the former case the system is characterized by states and state changes, in the latter case it is characterized by the events (actions) that can be performed to move from one state to another. Both are important paradigms for the specification of real-life systems and, as a result, formal methods ideally should cover both. Indeed, this trend is witnessed by the recent widespread use of modelling frameworks that allow both events and state changes to be specified. Examples include UML state machines [29,32].

In this paper we describe the UML model checker (UMC), a framework for the efficient verification of abstract system properties over UML models. In our case, a UML model is defined as a set of concurrently executing UML state machines. UML state machines describe the dynamic aspects of a system component's behaviour, enlightening both its state-based (*e.g.* values of object attributes) and its event-based (*i.e.* related to the executed actions) aspects. Both aspects are well represented by *doubly labelled transition systems* (L^2TSs), an extension of ordinary *labelled transition systems* (LTSs), introduced in [12] as a semantic model for concurrent systems. The UMC framework adopts L^2TSs as formal models of the system evolutions; a characteristic of these L^2TSs is that their states and transitions are labelled with sets of predicates and sets of events, respectively.

Since the amount of information that is useful to observe of these L^2TSs strictly depend on the level of abstraction at which one wants to reason, UMC provides appropriate abstraction mechanisms that allow one to select, or hide and possibly reshape, the information observable on the *ground* L^2TS (which represents the basic UML operational semantics of the system), and present it in the shape of a higher-level *abstract* L^2TS. The ground and the abstract L^2TSs, which both

[*] Corresponding author. Tel.: +39 050 3153471; fax: +39 0503152810.
E-mail addresses: terbeek@isti.cnr.it (M.H. ter Beek), fantechi@dsi.unifi.it (A. Fantechi), gnesi@isti.cnr.it (S. Gnesi), mazzanti@isti.cnr.it (F. Mazzanti).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.07.002

represent all the possible system behaviours (although at different levels of abstraction), can be model checked w.r.t. logic formulae expressed in UCTL. UCTL is a UML-oriented branching-time temporal logic which is able to express state predicates over system states, event predicates over single-step system evolutions, and combine these with temporal and boolean operators in the style of CTL [8] and ACTL [11,12].

In order to show the expressiveness of the logic and the potential of the UMC framework, we present a case study coming from an automotive scenario studied in the Service-Oriented Computing (SOC) domain. The goal of this case study is to verify *a priori*, *i.e.* before any implementation, certain design issues related to functional requirements.

The paper is organized as follows. After a background section (Section 2), Section 3 describes the semantics of UML state diagrams in terms of L^{2}TSs. In Section 4, we formally define the logic UCTL, after which Section 5 gives an overview of UMC's model-checking algorithm. In Section 6 we introduce the abstraction mechanisms over L^{2}TS provided by UMC. The selected case study illustrates the use of UMC in Section 7. Section 8 shows how abstraction can be applied to get a simplified view of the case study. Section 9 describes how UMC generates meaningful counterexamples. Finally, Section 10 describes UMC deployment and Section 11 draws some conclusions.

2. Background

We define some basic notations and terminology used throughout the paper.

Definition 1 (*LTS*). An LTS is a quadruple (Q, q_0, Evt, R), where:

- Q is a set of states
- $q_0 \in Q$ is the initial state
- Evt is a finite set of observable events (e ranges over Evt and η over 2^{Evt})
- $R \subseteq Q \times 2^{Evt} \times Q$ is the transition relation; instead of $(q, \eta, q') \in R$ we may also write $q \xrightarrow{\eta} q'$.

The main difference between this definition and the usual one is the transition labelling: we label them by sets of events rather than by single events.[1] This allows transitions from one state to another to represent sets of events without the need of intermediate states; this will turn out to be useful when modelling UML state machines. Another extension is to label states with atomic propositions, like the concept of an L^{2}TS [12], again extended as in Definition 1.

Definition 2 (*L^{2}TS*). An L^{2}TS is a quintuple (Q, q_0, Evt, R, AP, L), where:

- (Q, q_0, Evt, R) is an LTS
- AP is a set of atomic propositions (p ranges over AP)
- $L : Q \longrightarrow 2^{AP}$ is a function labelling each state with a subset of AP

The L^{2}TS thus obtained is similar to a so-called *Kripke transition system* [27], defined as an extension of a Kripke structure with a labelling over transitions.

Definition 3 (*Path*). Let (Q, q_0, Evt, R, AP, L) be an L^{2}TS and let $q \in Q$.

- σ is a *path* from q if $\sigma = q$ (empty path) or σ is a (possibly infinite) sequence $(q_1, \eta_1, q_2)(q_2, \eta_2, q_3) \cdots$, with $q_1 = q$ and $(q_i, \eta_i, q_{i+1}) \in R$ for all $i > 0$; σ is a *full path* if it cannot be further extended, *i.e.* σ is infinite or ends in a state with no outgoing transition; $fpath(q)$ denotes the set of all full paths from q.
- If $\sigma = (q_1, \eta_1, q_2)(q_2, \eta_2, q_3) \cdots$, then the ith state in σ, *i.e.* q_i, is denoted by $\sigma(i)$ and the ith set of events in σ, *i.e.* η_i, is denoted by $\sigma\{i\}$.
- The concatenation of paths σ and σ', denoted by $\sigma\,\sigma'$, is a partial operation defined iff σ is finite and its final state coincides with the initial state of σ'.

The usual notion of bisimulation equivalence can be straightforwardly extended to an L^{2}TS by taking into account equality of labelling of states, and considering the transitions labelled by sets of events.

Definition 4 (*Bisimulation*). Let $A_1 = (Q_1, q_{0_1}, Evt, \rightarrow_1, AP_1, L_1)$ and $A_2 = (Q_2, q_{0_2}, Evt, \rightarrow_2, AP_2, L_2)$ be two L^{2}TSs and let $q_1 \in Q_1$ and $q_2 \in Q_2$. We say that the two states q_1 and q_2 are *strongly equivalent* (or simply *equivalent*), denoted by $q_1 \sim q_2$, if there exists a *strong bisimulation* $\mathcal{B}$ relating q_1 and q_2. $\mathcal{B} \subseteq Q_1 \times Q_2$ is a *strong bisimulation* if for all $(q_1, q_2) \in \mathcal{B}$ and $\eta \in 2^{Evt}$:

(1) $L_1(q_1) = L_2(q_2)$,
(2) $q_1 \xrightarrow{\eta}_1 q_1'$ implies $\exists\, q_2' \in Q_2 : q_2 \xrightarrow{\eta}_2 q_2'$ and $(q_1', q_2') \in \mathcal{B}$, and
(3) $q_2 \xrightarrow{\eta}_2 q_2'$ implies $\exists\, q_1' \in Q_1 : q_1 \xrightarrow{\eta}_1 q_1'$ and $(q_1', q_2') \in \mathcal{B}$.

[1] Note that the standard LTS terminology uses the term *action* instead of *event*: here we prefer to use the term *event* since it is closer to the UML terminology.

We say that the two L^{2}TSs A_1 and A_2 are *equivalent*, denoted by $A_1 \sim A_2$, if there exists a strong bisimulation $\mathcal{B}$ such that $(q_{0_1}, q_{0_2}) \in \mathcal{B}$.

Analogously, the well-known notions of simulation preorder, weak (observational) equivalence[2] and (maximal) trace equivalence (for an encyclopedic treatment of these notions, consult [15]) can also be extended to an L^{2}TS.

3. The UMC framework and its L^{2}TS semantics for UML models

UMC is a framework for formal verification of the dynamic behaviour of UML models. In this section, we show how UMC (in particular versions 3.5–3.6) can be used to generate system models according to the UML paradigm; models which can be explored in their evolutions, abstracted, minimized and checked w.r.t. formal temporal properties expressed in the logic UCTL (see Section 4).

By "verification of UML models" we do not intend just the final *validation* step of a completed architectural design, but rather a formal support during all steps of the incremental design phase (*i.e.* when ongoing designs are still likely to be incomplete and, with a high probability, contain mistakes). Indeed, the UMC framework has been developed having in mind the requirements of a system designer as end user: (s)he intends to take advantage of formal approaches to achieve an early validation of the system requirements and an early detection of design errors. Therefore, the goals of the development of UMC have been:

- The possibility to manually explore the evolutions of a system and to generate a *summary* of its behaviour in terms of minimal abstract traces.
- The possibility to investigate abstract system properties by using a parametric branching-time temporal logic supported by an on-the-fly model checker.
- The possibility to obtain a clear explanation of the model-checking results.

According to the UML paradigm, a dynamic system can be seen as a set of evolving and communicating objects, where objects are class instances. The set of objects and classes which constitute a system can be described in UML by a structure diagram, while the dynamic behaviour of the objects can be described by associating a statechart diagram to their classes. Each object of the system will therefore behave like a state machine; it will have a set of local attributes, an event pool collecting the events that need to be processed, and the current status: since statecharts may contain parallel regions, the current status is given by a set of active states from those of the state diagram.

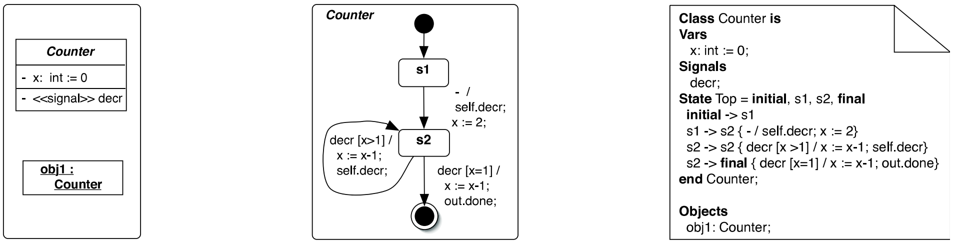
The details of an object's state machine behaviour are informally described by UML's semantics [29] (UML Superstructure 2.1.2, Sect. 15.3.12) as a traversal of a graph of state nodes interconnected by one or more joint transitions that are triggered by the dispatching of series of events. The so-called "run-to-completion" step defines the transition between two configurations of the state machine. The run-to-completion assumption states that an event can be taken from the pool and dispatched only when the previous event has been fully processed. During a run-to-completion step a sequence of activities can be executed; these include changing the value of some local attribute, sending a signal or call operation event to some object, (de)activating some node of the statechart, and removing an event from the state machine's events queue. The state machine modelled by an object (and described by a finite statechart) need not be finite, either because values of some infinite data type are used by an object attribute or because its events queue may grow unboundedly.

To transform this informal description of a run-to-completion step into a formal framework, and to model the concurrent evolution of state machines, some aspects that are (often intentionally) not precisely and uniquely defined by the UML standard have to be fixed. In this respect, UMC makes certain assumptions that, while compatible with the UML standard, are not necessarily the only possible choice. For instance, given a model constituted by more than one active object, a single system evolution is supposed to be constituted by a single evolution of a single state machine. This means that in UMC the concurrency among state machines is modelled through interleaving (even though priorities can be dynamically assigned to objects for greater flexibility). Moreover, the propagation of messages inside and among state machines is instantaneous and without duplication or message loss (this is an aspect intentionally left unspecified by the UML standard); the communication is direct and one-to-one (no broadcasts). We refer to the UMC documentation for all details on these aspects [25].

The graph modelling all possible evolutions of a UML model can be formally represented by an L^{2}TS (defined in Section 2). We show this by means of a small example, a counter described in UML as shown on the left side of Fig. 1.

Single_Counter is a very simple model constituted by only one class (*Counter*) and one object instance (*obj1*). Objects of class *Counter* have a local private integer attribute *x* (initialized at 0) and accept (internal) signals named *decr*. Their intended behaviour is described by the statechart shown in Fig. 1 (middle). As a first step, *Counter* sets its *x* attribute to 2 and generates the internal signal *decr*, *i.e.* a signal sent to itself. Then a loop is started in which the *decr* trigger enables a transition in which the *x* attribute is decremented and the internal signal *decr* is generated again (and this is repeated as long as *x* remains greater than 1). Finally, when *x* has become 1 a final step is performed (still triggered by the *decr* signal) which sets *x* to 0 and sends the *done* signal to an external *out* object (not modelled). The Single_Counter model can be textually encoded in UMC as a *Counter* class declaration, followed by an *obj1* object instantiation, as shown on the right side of Fig. 1.

[2] The unobservable action classically used to define weak equivalences is represented here by the empty set of events.



```
Class Counter is
Vars
  x: int := 0;
Signals
  decr;
State Top = initial, s1, s2, final
  initial -> s1
  s1 -> s2 { - / self.decr; x := 2}
  s2 -> s2 { decr [x >1] / x := x-1; self.decr}
  s2 -> final { decr [x=1] / x := x-1; out.done}
end Counter;

Objects
  obj1: Counter;
```

Fig. 1. Structure (left) and state (middle) diagrams and UMC encoding (right).

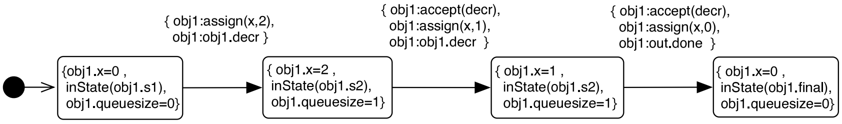


Fig. 2. The L^2TS associated with the Single_Counter model.

In UMC, classes define the structure and dynamic behaviour of the objects composing a system. Class declarations represent a template for the set of active and non-active objects of the system. In the case of active objects, the states and transitions associated with the class describe the dynamic behaviour of the corresponding objects, and a state machine (with its events queue) is associated with each such active object. Non-active objects play the role of *interfaces* towards the environment of the system and can only be the target of signals. In the end, a system is constituted by a static set of objects (*i.e.* no dynamic object creation) and, in order to exhibit a meaningful overall behaviour, it must be an *input-closed* system, *i.e.* it does not require the external environment to provide incoming signals or method calls: at least one active object must be defined to interact with the rest of the system, by sending signals or invoking methods. As illustrated in Fig. 1, a **Class** is constituted by its name, the list of events which trigger the transitions of its objects (asynchronous event **Signals** or synchronous call **Operations**), the list of attributes (**Vars**) local to its objects, the structure of its **States** (nodes of a statechart diagram), and the **Transitions** of its objects (edges of a statechart diagram).

All possible evolutions of this UMC model can be represented by an L^2TS, whose states are labelled with the structural properties of the system configurations (*e.g.* values held by the object attributes, names of the currently active substates in the statechart, size of the object events queue) and whose transitions are labelled with the events occurring during a system evolution step (*e.g.* removal of an event from a queue, update of a local attribute, sending of a signal or call operation event to some object). We refer to [25] for all the details on the syntax and semantics of UMC models. UMC associates the L^2TS depicted in Fig. 2 with the Single_Counter model.

Note that the atomic propositions that label the states of the L^2TS are actually predicates that encode the above mentioned structural properties, such as the current value of the object attributes (*obj1.x=0*), the current active substate (*inState(obj1.s2)*) and the size of the events queues (*obj1.queuesize=0*), and that the events labelling the transitions encode the activities occurring during an evolution step, like updating variable *x* (*obj1*:*assign(x,1)*), dispatching the *decr* signal (*obj1*:*accept(decr)*) and sending a signal (*obj1*:*out.done*).

Next we consider a model very similar to the previous one, but constituted by two instantiations of the Counter class instead of one, named *obj1* and *obj2*. Fig. 3 shows the L^2TS representing all possible model evolutions (information on the currently active substates and on the size of events queue is omitted).

An L^2TS in which all (supported) structural state properties and all executed activities are represented, is from now on called *ground* L^2TS. A ground L^2TS can be used as reference to express ground temporal properties like the property that "for all execution paths, attribute *x* of object *obj2* will eventually hold value 2, and after that it will eventually hold again value 0". However, it is often very useful to be able to reason not just on the ground L^2TS of a model, but on a more abstract version of it, in which many aspects not relevant for the properties one is interested to observe are omitted. For instance, the above property does not care about the values held by the attributes of object *obj1* at each system configuration, nor about all the activities performed by object *obj1* at each step of its evolution, nor of the size of event queues or the set of active states in each configuration. Similarly, we might be interested to observe only transition labels of a certain kind of events (*e.g.* specific signals) and not all the updates of the local attributes of the objects. To this aim, already in UMC version 3.5 a hiding mechanism was included, allowing the user to specify an "observation mode" of the system under analysis, in which the user can select the structural information or events (s)he is interested to observe, hiding the rest. For instance, once we set UMC to observe only the *x* attribute of *obj2* and the signal messages it generates, then we obtain the *abstract* L^2TS of the Two_Counters model shown in Fig. 4.

Abstraction of the ground L^2TS is very useful for the following two reasons:

(1) It allows the user to easily explore the system evolutions without being overwhelmed by a lot of irrelevant details (*e.g.* generating with UMC version 3.5 a graph like the one shown in Fig. 4). This really becomes a must when one wants

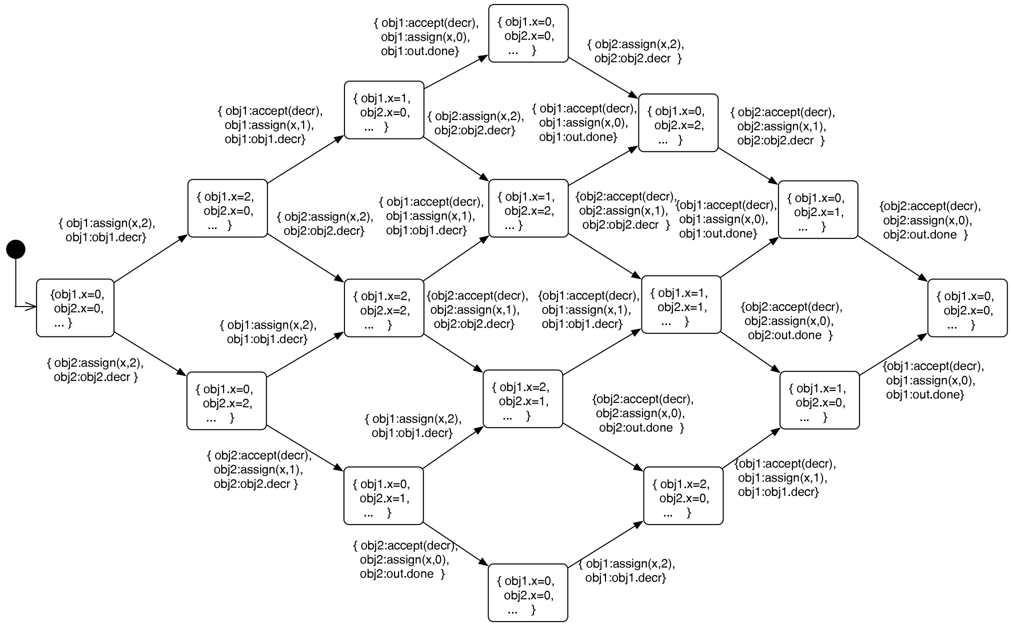

{ obj1:accept(decr), obj1:assign(x,0), obj1:out.done}
{ obj1.x=0, obj2.x=0, ... }
{ obj2:assign(x,2), obj2:obj2.decr }
{ obj1:accept(decr), obj1:assign(x,1), obj1:obj1.decr}
{ obj1.x=1, obj2.x=0, ... }
{ obj2:assign(x,2), obj2:obj2.decr}
{ obj1:accept(decr), obj1:assign(x,0), obj1:out.done}
{ obj1.x=0, obj2.x=2, ... }
{ obj2:accept(decr), obj2:assign(x,1), obj2:obj2.decr }
{ obj1:assign(x,2), obj1:obj1.decr}
{ obj1.x=2, obj2.x=0, ... }
{ obj2:assign(x,2), obj2:obj2.decr}
{ obj1:accept(decr), obj1:assign(x,1), obj1:obj1.decr}
{ obj1.x=1, obj2.x=2, ... }
{obj2:accept(decr), obj2:assign(x,1), obj2:obj2.decr }
{obj1:accept(decr), obj1:assign(x,0), obj1:out.done}
{ obj1.x=0, obj2.x=1, ... }
{obj2:accept(decr), obj2:assign(x,0), obj2:out.done }
{obj1.x=0, obj2.x=0, ... }
{ obj1.x=2, obj2.x=2, ... }
{obj2:accept(decr), obj2:assign(x,1), obj2:obj2.decr}
{obj1:accept(decr), obj1:assign(x,1), obj1:obj1.decr}
{ obj1.x=1, obj2.x=1, ... }
{obj2:accept(decr), obj2:assign(x,0), obj2:out.done }
{ obj1.x=0, obj2.x=0, ... }
{ obj1:assign(x,2), obj1:obj1.decr}
{ obj2:assign(x,2), obj2:obj2.decr }
{ obj1.x=0, obj2.x=2, ... }
{ obj1.x=2, obj2.x=1, ... }
{obj2:accept(decr), obj2:assign(x,0), obj2:out.done }
{ obj1.x=1, obj2.x=0, ... }
{obj1:accept(decr), obj1:assign(x,0), obj1:out.done}
{ obj1:assign(x,2), obj1:obj1.decr}
{ obj2:accept(decr), obj2:assign(x,1), obj2:obj2.decr }
{ obj1.x=0, obj2.x=1, ... }
{ obj1.x=2, obj2.x=0, ... }
{obj1:accept(decr), obj1:assign(x,1), obj1:obj1.decr}
{ obj2:accept(decr), obj2:assign(x,0), obj2:out.done }
{ obj1.x=0, obj2.x=0, ... }
{ obj1:assign(x,2), obj1:obj1.decr}

Fig. 3. The ground L^{2}TS associated with the Two_Counters model.

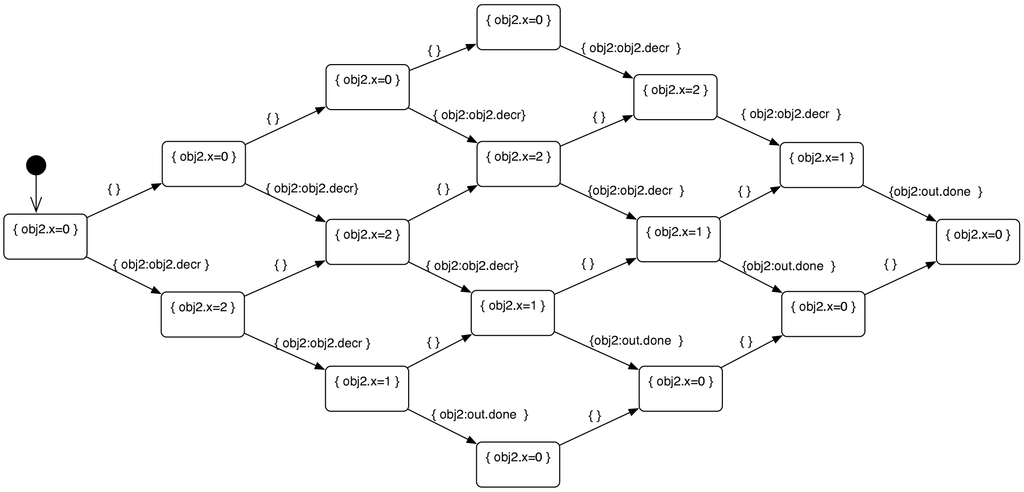


Fig. 4. The abstract L^{2}TS associated with the Two_Counters model.

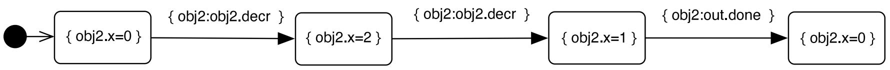


Fig. 5. An abstract minimized L^{2}TS of the Two_Counters model.

to explore the properties of complex models, which may consist of many objects with many attributes, while one actually wants to focus on the dynamic behaviour of just one component.

(2) It is possible to exploit L^{2}TS minimization tools, allowing the user to observe in a more compact form the actual system behaviour w.r.t. the observed properties. For instance, an abstract minimized (w.r.t. the *weak maximal trace equivalence*) version of the L^{2}TS representing all system evolutions of the Two_Counters is shown in Fig. 5. More on this is given in Section 8.

4. UCTL: A state/event-based temporal logic

In this section, we present the syntax and semantics of UCTL. This temporal logic, *state and event based*, allows one to reason on state properties as well as to describe the behaviour of systems that perform events during their lifetime. UCTL includes both the branching-time action-based logic ACTL and the branching-time state-based logic CTL [8].

UCTL is the logic used in UMC to reason over UML specifications and L^{2}TSs.

Definition 5 (*Event Formulae*)**.** Event formulae are built over $Evt \cup \{\tau\}$[3]:

$$\chi ::= tt \quad | \quad e \quad | \quad \tau \quad | \quad \neg\chi \quad | \quad \chi \wedge \chi'$$

Event formulae are thus simply boolean compositions of events $e \in Evt$. As usual, we will use *ff* to abbreviate $\neg tt$ and $\chi \vee \chi'$ to abbreviate $\neg(\neg\chi \wedge \neg\chi')$.

Definition 6 (*Event Formulae Semantics*)**.** The satisfaction relation $\models$ for event formulae of the form $\eta \models \chi$ is defined over sets of events as follows:

- $\eta \models tt$ holds always
- $\eta \models e$ iff $e \in \eta$
- $\eta \models \tau$ iff $\eta = \varnothing$
- $\eta \models \neg\chi$ iff not $\eta \models \chi$
- $\eta \models \chi \wedge \chi'$ iff $\eta \models \chi$ and $\eta \models \chi'$

Note that the semantics of event formulae requires that an event *e* matches exactly one event in η.

Definition 7 (*UCTL Syntax*)**.** The syntax of UCTL formulae is defined as:

$$\begin{aligned}(\textit{state formulae})\phi &::= true \quad | \quad p \quad | \quad \neg\phi \quad | \quad \phi \wedge \phi' \quad | \quad E\,\pi \quad | \quad A\,\pi\\ (\textit{path formulae})\pi &::= X_\chi \phi \quad | \quad \phi\ {}_\chi U_{\chi'}\ \phi' \quad | \quad \phi\ {}_\chi W_{\chi'}\ \phi'\end{aligned}$$

E and *A* are existential and universal *path quantifiers*. *X*, *U* and *W* are the *next*, *until* and *weak until* operators drawn from those firstly introduced in [11] and later elaborated in [26].[4] Intuitively, the next operator says that in the next state of the path, reached by an event satisfying χ, the formula ϕ holds. The until operator *U* says that ϕ' holds at some future state of the path reached by a last event satisfying χ, while ϕ holds from the current state until that state is reached and all events executed in the meanwhile along the path satisfy χ. The weak until operator *W* (also called *unless*) holds on a path either if the corresponding strong until operator holds or if for all states of the path the formula ϕ holds and all events of the path satisfy χ. In linear-time temporal logic (LTL), the formula $\phi\ W\ \psi$ can be derived from the until (U) and always (G) operators, as follows: $\phi\ U\ \psi \vee G\,\phi$. This way to derive the weak until operator from the until operator is not feasible in UCTL since disjunction or conjunction of path formulae is not expressible according to UCTL's syntax, and the same holds for any pure branching-time temporal logic. Starting from the syntax of UCTL, it is possible to derive both CTL and ACTL by removing the event or the state component, respectively.

The interpretation domain of the UCTL formulae are L^{2}TSs over the set of events *Evt* and the set of predicates *AP* (*i.e.* with its states labelled by subsets of *AP* and its transitions labelled by subsets of *Evt*). To define the semantics of UCTL, we use the notion of a *path* of an L^{2}TS as defined in Section 2. Recall that $fpath(q)$ denotes the set of all *full paths* from *q* while, for a path σ, its *i*th state is denoted by $\sigma(i)$ and its *i*th set of events by $\sigma\{i\}$.

Definition 8 (*UCTL Semantics*)**.** Let (Q, q_0, Evt, R, AP, L) be an L^{2}TS, let $q \in Q$, and let $\sigma \in fpath(q')$ for some $q' \in Q$. The satisfaction relation of the UCTL formulae is defined as follows:

- $q \models true$ holds always
- $q \models p$ iff $p \in L(q)$
- $q \models \neg\phi$ iff not $q \models \phi$
- $q \models \phi \wedge \phi'$ iff $q \models \phi$ and $q \models \phi'$
- $q \models E\,\pi$ iff $\exists\,\sigma \in fpath(q)\colon\ \sigma \models \pi$
- $q \models A\pi$ iff $\forall\,\sigma \in fpath(q)\colon\ \sigma \models \pi$
- $\sigma \models X_\chi\phi$ iff $\sigma\{1\} \models \chi$ and $\sigma(2) \models \phi$
- $\sigma \models \phi\ {}_\chi U_{\chi'}\ \phi'$ iff $\exists j \geq 1\colon \sigma(j) \models \phi, \sigma\{j\} \models \chi'$, and $\sigma(j+1) \models \phi'$ and $\forall\, 1 \leq i < j\colon \sigma(i) \models \phi$ and $\sigma\{i\} \models \chi$
- $\sigma \models \phi\ {}_\chi W_{\chi'}\ \phi'$ iff either $\sigma \models \phi\ {}_\chi U_{\chi'}\ \phi'$ or $\forall j \geq 1\colon \sigma(j) \models \phi$ and $\sigma\{j\} \models \chi$

Beyond the usual $\vee$ and $\Rightarrow$ boolean operators, other useful logic operators can now be derived as usual, such as:

- *false* stands for $\neg$ *true*
- $[\gamma]\,\phi$ stands for $\neg < \gamma > \neg\,\phi$
- $EF\phi$ stands for $E(true\ {}_{tt}U\phi)$
- $AF\phi$ stands for $A(true\ {}_{tt}U\phi)$
- $AG\,\phi$ stands for $\neg\, EF\, \neg\,\phi$
- $< \gamma > \phi$ stands for $EX_\gamma\ \phi$
- $E(\phi\ {}_\chi U\ \phi')$ stands for $\phi' \vee [E(\phi\ {}_\chi U_\chi\ \phi')]$
- EF_γ *true* stands for $E(true\ {}_{tt}\,U_\gamma\ true)$
- $E(\phi\ {}_\chi W\ \phi')$ stands for $\phi' \vee [E(\phi\ {}_\chi W_\chi\ \phi')]$

[3] In UCTL, the classical unobservable action τ is actually an event formula matching the *empty set* of transition labels.

[4] Contrary to ACTL, in UCTL the operator $X_\chi\phi$ can be derived as $false\ {}_{false}U_\chi\ \phi$.

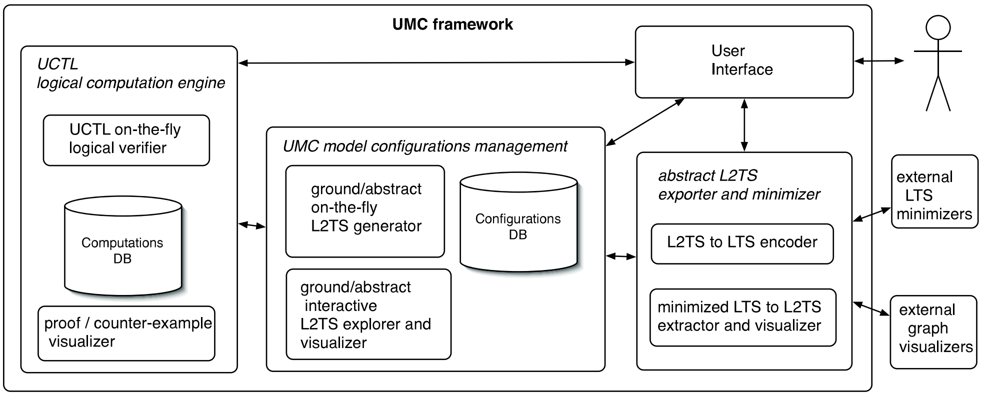


Fig. 6. The architecture of the UMC framework.

Operators $<\chi>\phi$ ("possibly") and $[\chi]\phi$ ("necessarily") are the diamond and box modalities of the Hennessy–Milner logic [19]. The meaning of $EF\phi$ is that ϕ must eventually be true in a possible future, that of $AF\phi$ is that ϕ must eventually be true in all possible futures, and that of $AG\phi$ is that ϕ must always be true in all possible futures.

The property "for all execution paths, attribute *x* of object *obj2* will eventually hold value 2, and after that it will eventually hold again value 0", informally introduced in Section 3, can be formalized in UCTL as:

$AF\ (obj2.x{=}2)\ AF\ (obj2.x{=}0)$

We can observe that this formula is satisfied by each of the L^{2}TSs of Figs. 3–5.

Other properties that can be checked on the Two_Counters model are:

$EX_{obj2:obj2.decr}\ true$. This should be read as: in the initial configuration, *obj2* can perform an evolution step by which it sends the signal *decr* to itself. This property is checked to be true for the system model described in each of the L^{2}TSs of Figs. 3–5.

$AG\ ((EX_{obj2:obj2.decr}\ true) \Rightarrow (obj2.x = 0))$. This should be read as: the event *obj2:obj2.decr* can only occur when the object attribute has value 0. This property is false in the different system views presented in Figs. 3–5.

The logic UCTL is *adequate* w.r.t. strong bisimulation equivalence on L^{2}TSs (see Section 2 for a definition) [2]. Adequacy [30] means that two L^{2}TSs A_1 and A_2 are strongly bisimilar if and only if $F_1 = F_2$, where $F_i = \{\ \psi \in UCTL : A_i \models \psi\ \}$ for $i = 1, 2$. In other words, adequacy implies that if there is a formula that is not satisfied by one of the L^{2}TSs but satisfied by the other L^{2}TS, then the two L^{2}TSs are not bisimilar, and – on the other hand – if two L^{2}TSs are not bisimilar, then there must exist a distinguishing formula.

5. The on-the-fly structure of the UMC framework

The UMC framework adopts an "on-the-fly" approach to generate the L^{2}TS from a UML specification, meaning that the L^{2}TS corresponding to the model is generated "on-demand", following either the interactive requests of a user while exploring the system, or the needs of the logical verification engine, or a user's minimization requests. The set of generated system configurations and (possibly) their immediate "next-step" evolutions are incrementally saved into a "Configurations" database and their abstract view is computed. The overall structure of the framework is shown in Fig. 6.

The minimization module of the tool is the part that less benefits from the on-the-fly approach since an exhaustive exploration of the state space is needed in order to correctly construct a minimized L^{2}TS. This minimization is implemented by first translating the L^{2}TS into an equivalent LTS, then applying standard (external) minimization algorithms to this LTS, and finally translating back the minimized LTS into the L^{2}TS format in order to visualize it (the logical verification engine always starts from the original L^{2}TS). The translation of the L^{2}TS into a plain LTS is achieved by translating each single transition according to the rules shown in Fig. 7. In practice, the state labels of the source and target states of the L^{2}TS are pushed inside the transition labels of the LTS, and a special outgoing transition is added to final L^{2}TS nodes. These rules guarantee that transitions labelled with τ ("tau"), which can be removed by the minimization process, really correspond to internal (and possibly unobservable) steps during which no observable event actually occurs and nothing changes in terms of observable properties of the states.

The logical verification engine, instead, is the part that best exploits the on-the-fly approach of the L^{2}TS generator. It maintains an archive of computation fragments; this is not only useful to avoid unnecessary duplications in the evaluation of subformulae, but necessary to deal with recursion in the evaluation of a formula, arising from the presence of loops in L^{2}TSs. Computation fragments are in the form ⟨`subformula`, `state`, `computationprogress`⟩ and associate each evaluation of a `subformula` in some `state` with its computation status (`computationprogress`), which in the end will be the final

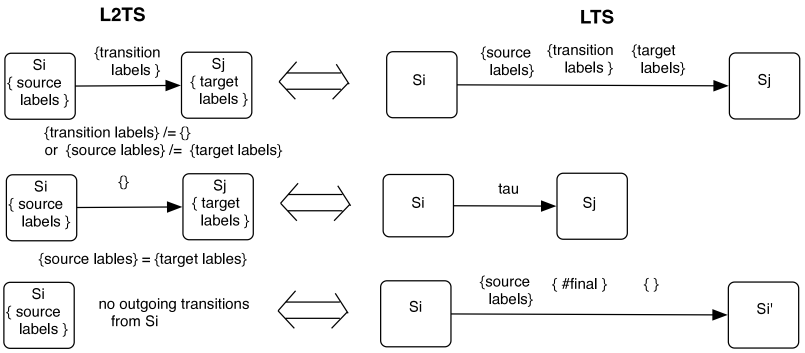


Fig. 7. Conversion rules between the L^{2}TS and LTS formats.

```
function ExistsNext (Form: in Formula; Current_State: in State;  Evaluation_Depth: in Natural)
         return Computation is
      a formula Form has the structure:  (Kind => Next,  Event,  NextForm)
      a computation fragment Comp has the structure: (Form, Current_State, Result, Subcomputations)
      a transition T has the structure:  (Source, Labels, Target)
      --
  Comp := Check_Computation (Form, Current_State);
   if  the computation has already been performed  then
      return Comp;
   else
      Comp := (Form, Current_State, In_Progress, Empty_Subcomputations):
   end if;
   if Evaluation_Depth  >  Current_Bound_Limit then
       Comp.Result :=  Aborted;
      return  Comp;
   end if;
   Tmp_Result := False;
   foreach T in outgoing transitions of Currrent_State loop
      if T.Labels satisfy Form.Event then
         SubComp := Evaluate (Form.NextForm, T.Target, Evaluation_Depth + 1);
         if SubComp.Result = True then
            Tmp_Result := True:
            Comp.Subcomputations :=  { SubComp };
            exit;
         elsif SubComp.Result  = False and Tmp_Result = False then
            Comp.Subcomputations :=  Comp.Subcomputations  + SubComp;
         elsif SubComp.Result  = Aborted  then
            Tmp_Result = Aborted;
            Comp.Subcomputations :=  Empty_Subcomputations;
         end if;
      end if;
   end loop;
   Comp.Result := Tmp_Result;
   Save Comp in the computations_DB;
   return  Comp;
end ExistsNext;
```

Fig. 8. The evaluation schema for the UCTL operator X.

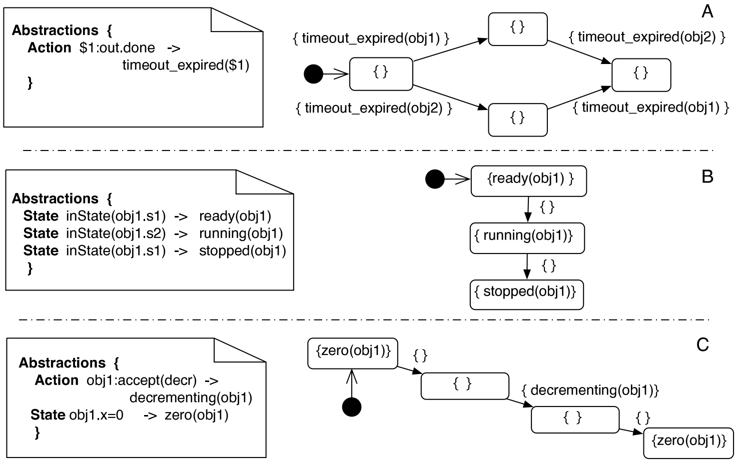
`True` or `False` value. With each computation fragment is also associated a set of subcomputations which serve to explain the final result.

In general, given a state of an L^{2}TS, the validity of a UCTL formula in that state is evaluated by analyzing the transitions allowed in that state, and by analyzing the validity of the needed subformulae in a subset of the next reachable states, all this in a recursive "depth-first" way.

In the case of infinite state spaces, the above approach may fail to produce a result even when a result could actually be deduced in a finite number of steps. This is a consequence of the algorithm's "depth-first" recursive structure. The solution taken to solve this problem is to adopt a bounded model-checking approach [6], *i.e.* the evaluation is started by assuming a certain value as a maximum depth of the evaluation. In this case, if the evaluation of a formula reaches a result within the requested depth, the result holds for the whole system; otherwise the maximum depth is increased and the evaluation is subsequently retried (preserving all useful subresults already found). This approach, initially introduced in UMC to address infinite state spaces, happens to be quite useful also for another reason: by setting a small initial maximum depth and a small automatic increment of this bound at each re-evaluation failure, once a result is finally found then we also have a reasonable (almost minimal) explanation for it. This is very useful also in the case of finite state machines.

It is beyond the scope of this paper to present detailed descriptions of the evaluation algorithms for the more complex cases of recursive operators, but in Fig. 8 we do show the schema of the algorithm for the evaluation of UCTL's next (X) operator. This is a particularly simple logic operator since it does not involve the complexity induced by recursion; it is however sufficient to give an idea of how the on-the-fly approach of the evaluation mechanism behaves.

The main advantage of the on-the-fly approach to model checking is that, depending on the formula, only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result [9].



```
Abstractions {
  Action $1:out.done ->
                timeout_expired($1)
 }
```

A
{ timeout_expired(obj1) }
{ timeout_expired(obj2) }
{ timeout_expired(obj2) }
{ timeout_expired(obj1) }
{ }

```
Abstractions {
  State inState(obj1.s1) -> ready(obj1)
  State inState(obj1.s2) -> running(obj1)
  State inState(obj1.s1) -> stopped(obj1)
 }
```

```
Abstractions {
  Action obj1:accept(decr) ->
            decrementing(obj1)
 State obj1.x=0   -> zero(obj1)
 }
```

Fig. 9. Three sets of abstraction rules and three abstract minimized L^{2}TSs.

6. Adding explicit abstraction mechanisms

Adopting a standard format for labels, it is possible to implement a logical verification engine that works on the abstract L^{2}TS independently of how it is generated from the underlying computational model.

In Section 3, we presented a hiding mechanism to be applied to the ground L^{2}TS to produce an abstract version. In UMC version 3.6 this mechanism was extended and made more flexible, allowing a user to specify an explicit set of *hiding* and *renaming* rules precisely defining the ground labels to be observed, possibly reshaping them to fit a standard format. This is done by adding an **Abstractions** section with a list of abstraction rules to the UMC encoding.

Abstraction rules translate state and transition labels of ground L^{2}TSs to state and transition labels of abstract L^{2}TSs, respectively, as follows (renaming):

$$\textbf{State}\quad \textit{ground\_state\_label} \;\rightarrow\; \textit{abstract\_state\_label(arg1, arg2, \dots)}$$

$$\textbf{Action}\quad \textit{ground\_action\_label} \;\rightarrow\; \textit{abstract\_event\_label(arg1, arg2, \dots)}$$

in which *arg1, arg2, . . .* is a list of zero or more arguments. The left-hand sides of these abstraction rules may include variable parameters, denoted by $\$i$, for $i \in \mathbb{N}$, that are matched by elements of the ground labels: the matching gives a value to the variable which can then be used on the right-hand sides. Note that those state and transition labels that are not replaced by their abstract counterparts may not be observed (hiding).

For instance, for the Two_Counters model of Fig. 3 three sets of abstraction rules and the resulting abstract minimized L^{2}TSs are given in Fig. 9. Note how the top abstraction rule uses a variable substitution ($1) to extract the information of interest (the executing object) from the ground label and move it into the abstract label.

With respect to the sets of abstractions shown in Fig. 9 (A), (B) and (C), we can verify that the following formulae **FA**, **FB** and **FC**, respectively, hold:

$$\begin{array}{ll} \textbf{FA} & (AF_{timeout\_expired(obj1)}\; true) \;\wedge\; (AF_{timeout\_expired(obj2)}\; true) \\ \textbf{FB} & AF\; AG\; stopped(obj1) \\ \textbf{FC} & AG\; ((< decrementing(obj1) >\; true) \Rightarrow \neg zero(obj1)) \end{array}$$

7. Automotive case study

A vehicle that leaves the assembly line today is equipped with a multitude of sensors and actuators providing the driver with services that assist in conducting the vehicle more safely, like vehicle stabilization systems. Driver assistance systems kick in automatically when the vehicle context renders it necessary, and more and more context is taken into account (*e.g.* road conditions, vehicle condition, driver condition, weather conditions, traffic conditions). In addition, the highly advanced mobile technology available nowadays allows car drivers telephone and Internet access within their vehicles, giving rise to a variety of new services for the automotive domain, such as handling based on information provided by other vehicles passing nearby or by location-based services in the surrounding. Some of these scenarios, like the one described in this section, have been used to validate the engineering approaches developed in the EU IST-FET Integrated Project Sensoria [35], which addresses Service-Oriented Computing by building, from first-principles, novel theories, methods and tools supporting the engineering of software systems for service-oriented overlay computers.

7.1. The on road assistance scenario

While a driver is on the road with her/his car, the vehicle's diagnostic system reports a low oil level. This triggers the in-vehicle diagnostic system to report a problem with the pressure of the cylinder heads, resulting in the car being no longer drivable, and to send this diagnostic data as well as the vehicle's GPS coordinates to the repair server. Based on the driver's preferences, the service discovery system identifies and selects an appropriate set of services (garage, tow truck and rental car) in the area. When the driver makes an appointment with the garage, the results of the in-vehicle diagnosis are automatically sent along, allowing the garage to identify the spare parts needed to repair the car. Similarly, when the driver orders a tow truck and a rental car, the vehicle's GPS coordinates are sent along. Obviously, the driver is required to deposit a security payment before being able to order any service. Finally, each service can be denied or cancelled, causing an appropriate compensation activity.

7.2. A UML specification

One of the techniques developed in Sensoria is a specific UML profile, called UML4SOA [23], for the design of abstract service-oriented architectures in terms of specialized activity diagrams. In [22], a UML4SOA model for the above automotive scenario is presented, identifying the components that logically play an active role, and illustrating their required relations and interactions. The scenario essentially involves the concept of a *Car* and that of external *Bank*, *Garage*, *TowTruck* and *RentalCar* services. A *Car* component can be further decomposed into subcomponents like the *Engine* (detecting the low oil level alert), the *GPS* (providing the vehicle coordinates), the *Reasoner/Discovery* system (searching in and selecting from a local database the best pack of assistance services), a *Vehicle Communication Gateway* (handling the telecommunications between the vehicle and the external services) and, finally, an *Orchestrator* (supporting the on-board *Road Assistance* service by orchestrating the various internal and external services to achieve its goals). In Fig. 10 we show the UML 2.0 activity diagram according to the UML4SOA profile describing the dynamic behaviour of the *Orchestrator*.

The complete automotive scenario can be encoded in UMC as a set of communicating UML state machines. The structure diagram of our automotive system is shown in Fig. 11. In Fig. 12 we show instead the overall structure of the `Car` and `Bank` statechart diagrams, which put in evidence how the various components of the scenario are mapped to top-level state machines of submachines. The reader can consult our complete specification online [1]. Of all the involved submachines we will show here only the structure of the `Car.Orchestrator` component (see Fig. 13) which acts as the logical kernel of our system.

Starting from the set of communicating UML state machines of the automotive scenario, the ground and the abstract L^2TSs could be generated according to the elements of the scenario one wishes to observe in order to make an analysis of the specification. Let us assume that the properties we are interested to observe (and check) are those related to the abstract behaviour of the various system components or subcomponents seen as service providers. In such a case, we could observe the system classifying the abstract events as (i) *request* of some service, (ii) events which correspond to a successful reply, *response*, from a service to a request, (iii) events which correspond to a negative reply, *fail*, from a service and (iv) events which correspond to the *cancelling* of a previously issued request. Similarly, we could be interested in observing as state predicates whether or not a certain service is in a status in which it is enabled to *accept a request*. In particular, the services we are interested to analyze are the *road_assistance* service as a whole, plus the *bankcharge*, *garage*, *towtruck*, and *rentalcar* services.

A fragment of the abstraction rules used to extract the abstract information we need from the ground L^2TS is shown in Fig. 14. Actually, we are not that interested in the details of labels of the ground L^2TS appearing on the left side of those rules (*i.e.* how the information is extracted from the concrete details of the UML implementation; we just suppose that whoever designed the model did the job correctly), but rather in the abstract view provided by their right sides (*i.e.* which abstract properties are we able to observe and reason on, as requested by the high-level system designer). Both the translation from the UML4SOA profile to a UMC state machine representation and the generation of abstraction rules like those used in our example should ideally be performed automatically. Indeed, we are currently working on the implementation of such a translation. In our case, however, we have proceeded with a manual encoding.

7.3. UMC verification

We now show how some abstract safety/liveness/orchestration properties related to the service-oriented view of the automotive system can actually be encoded as UCTL formulae and checked within the UMC framework.

F1. The *Car* component of the system is initially in the *accepting* status w.r.t. the *road_assistance* operation and it remains in that state until a corresponding *request(road_assistance)* event is accepted:

$$A\ (\ \mathit{accepting\_request(road\_assistance,car1)}\ {}_{tt}\ U_{\mathit{request(road\_assistance,car1)}}\ \mathit{true}\ )$$

F2. If the *Bank* component of the system accepts a *request(bankcharge,car1)* event, then it will always eventually send back a corresponding *response* or *fail* event, and meanwhile no further *request(bankcharge,car1)* events are accepted:

$$AG\ [\ \mathit{request(bankcharge,car1)}\ ]A\ (\ {}_{\neg \mathit{request(bankcharge,car1)}}\ U_{\mathit{response(bankcharge,car1)} \vee \mathit{fail(bankcharge,car1)}}\ \mathit{true}\ )$$

F3. If the *Car* component of the system receives a *response(garage,car1)* event from the *Garage* component, and subsequently receives a *fail(towtruck,car1)* event from the *TowTruck* component, then it will always eventually react by issuing a *revoke(garage,car1)* event:

$$AG\ [\ \mathit{response(garage,car1)}\ ]AF\ (\ [\ \mathit{fail(towtruck,car1)}\ ]\ AF_{\mathit{revoke(garage,car1)}}\ \mathit{true}\ )$$

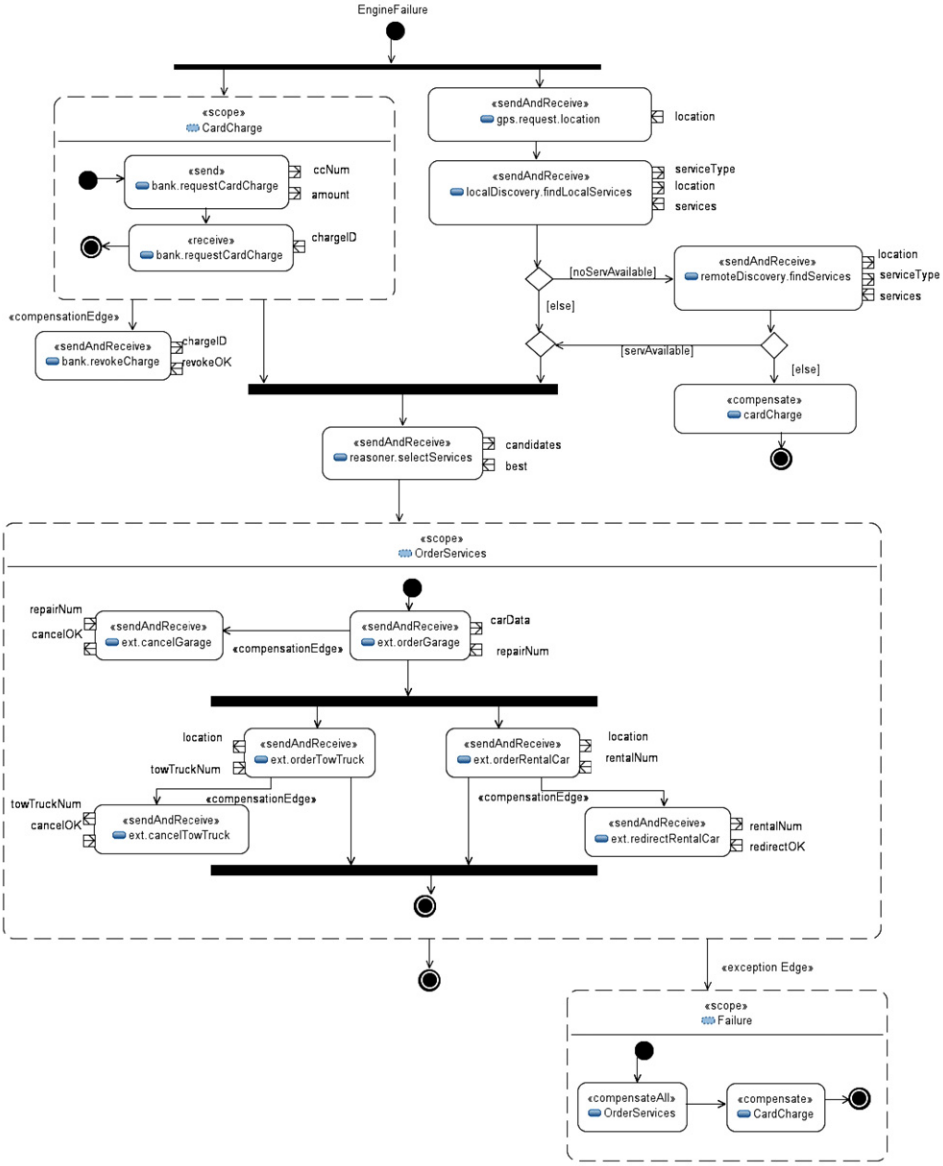


Fig. 10. Orchestration of services.

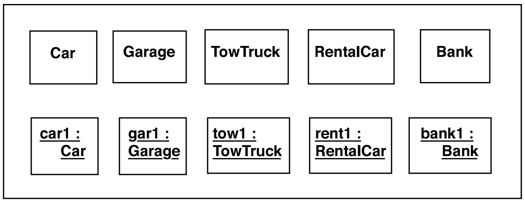


Fig. 11. Structure diagram of the automotive system.

F4. If the *Bank* component responds with a *fail(bankcharge,car1)* event to the *Car* component, then neither the *Garage*, nor the *RentalCar*, nor the *TowTruck* component will ever generate a successful *response* event:

$$AG\,[\,fail(bankcharge,car1)\,]\neg\, EF_{response(garage,car1)\vee response(rentalcar,car1)\vee response(towtruck,car1}\ true$$

F5. If the *Garage* component responds with a *fail(garage,car1)* event to the *Car* component, then the latter will always eventually recover from this failure by issuing a *revoke(bankcharge,car1)* event to the *Bank* component:

$$AG\,[\,fail(garage,car1)\,]\ AF_{revoke(bankcharge,car1)}\ true$$

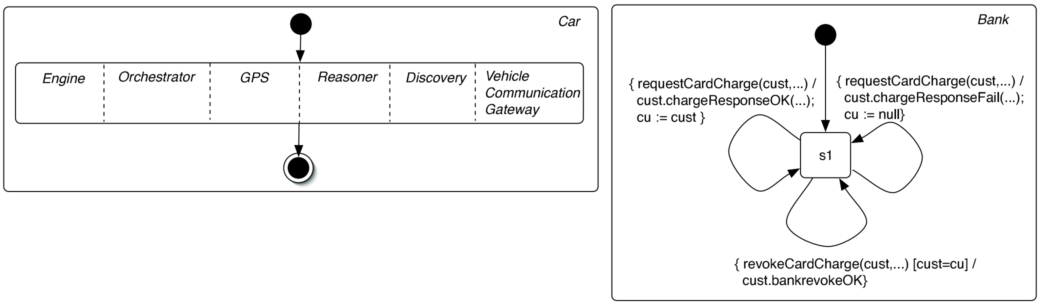

Car
Engine
Orchestrator
GPS
Reasoner
Discovery
Vehicle Communication Gateway
Bank
{ requestCardCharge(cust,...) / cust.chargeResponseOK(...); cu := cust }
{ requestCardCharge(cust,...) / cust.chargeResponseFail(...); cu := null}
s1
{ revokeCardCharge(cust,...) [cust=cu] / cust.bankrevokeOK}

Fig. 12. Overall structure of the `Car` and `Bank` statecharts.

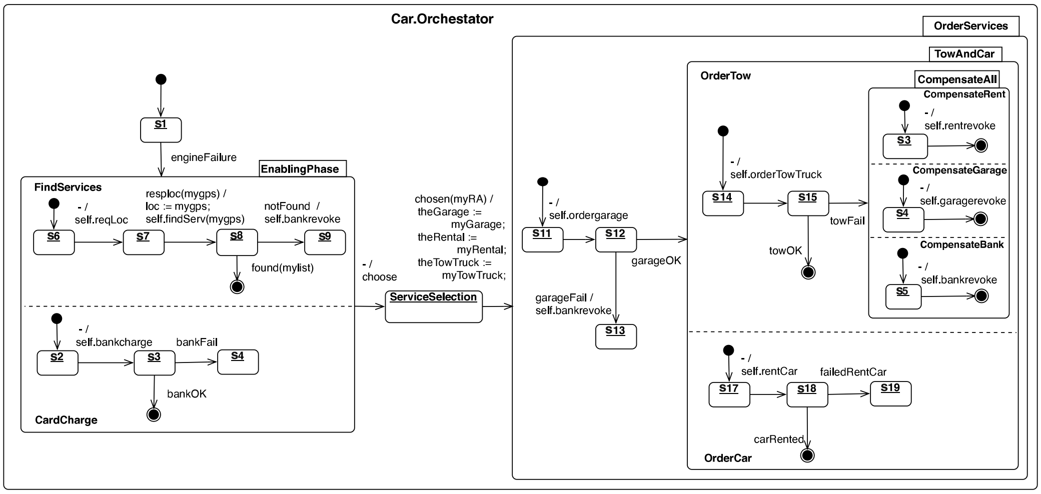


Fig. 13. The statechart diagram for the `Car.Orchestrator` component.

```
Abstractions {
  State   inState(car1.Orchestrator.o1)  ->  accepting_request(road_assistance)
  Action  car1:accept(engineFailure)  ->  request(road_assistance)
    ...
  State   inState(bank1.s1)  ->  accepting_request(bankcharge)
  Action  car1:bank1.requestCardCharge  ->  request(bankcharge)
  Action  bank1:car1.chargeResponseFail  ->  fail(bankcharge)
  Action  bank1:car1.chargeResponseOK  ->  response(bankcharge)
  State   inState(bank1.s1) and bank.cu=car1  ->  accepting_revoke(bankcharge)
   ...
  Action  car1:$1.requestGarage  ->  request(garage)
  Action  $1:car1.garageResponseFail  ->  fail(garage)
  Action  $1.car1.garageResponseOK  ->  response(garage)
  Action  car1:$1.revokeGarage  ->  revoke(garage)
  ... }
```

Fig. 14. A set of abstraction rules for the automotive scenario.

Finally, a particular property that does not hold for our model is the following.

F6. If the *Car* component of the system sends a *request(garage,car1)* event to the *Garage* component, then the latter will always eventually send back a corresponding *response(garage,car1)* event:

$$AG\ [\ request(garage,car1)\ ]\ AF_{response(garage,car1)}\ true$$

We model checked above properties over the abstract view of the automotive scenario. The results of these verifications are collected in Table 1.

Note that the only property *not* satisfied by the automotive scenario is **F6**. This is not surprising: the *Garage* component might be temporarily unable to successfully deal with the request, so it sends the *Car* component an unsuccessful response *fail(garage,car1)* rather than a successful *response(garage,car1)*. In this case, UMC can provide a counterexample, as discussed in Section 9.

It is worth making two observations on the above verifications. First note that only a few formulae actually exploit the on-the-fly approach of the framework (exploring just a small fragment of the state space). This is partly due to the structure

Table 1
Validation results.

Verified formula	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**
Validation result	True	True	True	True	True	False
Generated states	3	536	536	536	536	35

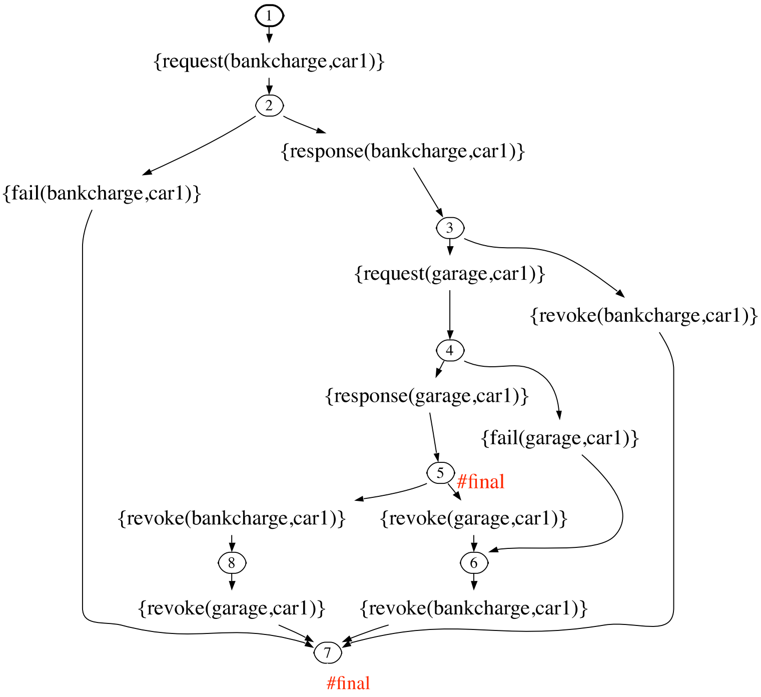


Fig. 15. An abstract minimized view of the automotive case study.

of the formulae, which are often of the form $AG\ldots$ (*i.e.* for all states of the state space it is true that ...), and partly due to the model's correctness: formulae of the form $AG\ldots$ actually require exploring the whole state space to possibly answer positively. During system development (*i.e.* for most of the development time), when the implementation might still contain several errors, the advantages of the on-the-fly approach are more evident.

Second, writing temporal logic formulae is an activity that is definitely not easy, even if one is allowed to reason at an abstract level and without knowing anything of the actual ground implementation (as is the case of UCTL), and often requires skills not usually found in industrial environments. It should however be noted that several of the above properties have rather standard translations in terms of temporal logic operators and abstract request/response events (*e.g.* availability, responsiveness of services or components). We could thus imagine a predefined taxonomy for properties specified in UCTL, much like the specification patterns repository for LTL, (A)CTL, *etc.* [36], that would also allow a designer not particularly skilled in formal methods to just select an appropriate predefined property and check it for the services or components of interest. This, in particular when associated with the other desirable automatic mechanisms for generating the system implementation and model abstractions from UML4SOA profiles, could really help the adoption of formal verification frameworks inside industrial software development cycles.

8. Abstract minimized traces

As already mentioned at the end of Section 3, UMC also allows the selection of a small set of abstract events of interest, starting from which it generates an abstract minimized (w.r.t. maximal trace equivalence) view of the system. This is an extremely powerful way for checking whether a system's behaviour matches the intended requirements, which moreover works well also in case the requirements themselves are not so clear or well formalized. For instance, in the automotive system of the previous section, selecting only the interactions of the *Bank* and *Garage* components with the *Car* component results in UMC producing the abstract minimized view depicted in Fig. 15.

Fig. 15 summarizes all possible system traces w.r.t. the selected set of events, thus providing a precise and complete understanding of the relation between the activities of charging a credit card and of reserving a garage. It is extremely easy to become confident of the correctness of the model by just looking at this figure, without being forced to identify *a priori* a *complete* set of requirements and formalize them in terms of logic formulae to be model checked separately. Unfortunately, however, this approach to system verification does also have several drawbacks:

- It is a computationally expensive approach: for very large models it might be too resource consuming to compute an abstract minimized view.
- If the L^2TS is not finite, then it is not even a matter of available computing resources: building the abstract minimized view is not possible.
- The abstract minimized view has completely lost the connection with the original ground L^2TS: if the system behaviour is not what one expected it to be, then there is no immediate way of exploring why certain evolutions are (not) possible in the ground L^2TS.
- The approach is not applicable for complex enough abstract minimized views, which can defy the intuition of who want to assess its correctness just by looking at the graph.

Note that trace minimization is used just as an aid to convey to the user a graphical summary of the system behaviour. The verification of a logic formula is in any case performed on the basis of the ground / abstract L^2TSs, which have exactly the same structure and differ only from the point of view of the labelling. Other types of minimizations could in principle be supported (*e.g.* strong/weak bisimulation) and could be used to verify a formula over a smaller system. Such a minimization is however in general computationally heavier than the direct on-the-fly verification on the original system.

9. Proofs and counterexamples

The generation of counterexamples has been always cited as the main advantage of model checking. Counterexamples (or witnesses) are usually returned by model checkers in the form of a computation path; however, only for certain kinds of formulae a computation path is able to explain completely the reason of satisfaction or missed satisfaction. Moreover, providing a useful counterexample for a given temporal logic formula is quite complex in the case of branching-time logics [10,18]. In the development of UMC, one of the goals was the ability to provide a clear explanation of the model-checking results. We aim therefore at providing meaningful counterexamples in the case of missed satisfaction and "proofs", rather than just witness paths, in the case of satisfaction of a formula. The problems that need to be solved for the generation of useful proofs / counterexamples are essentially three:

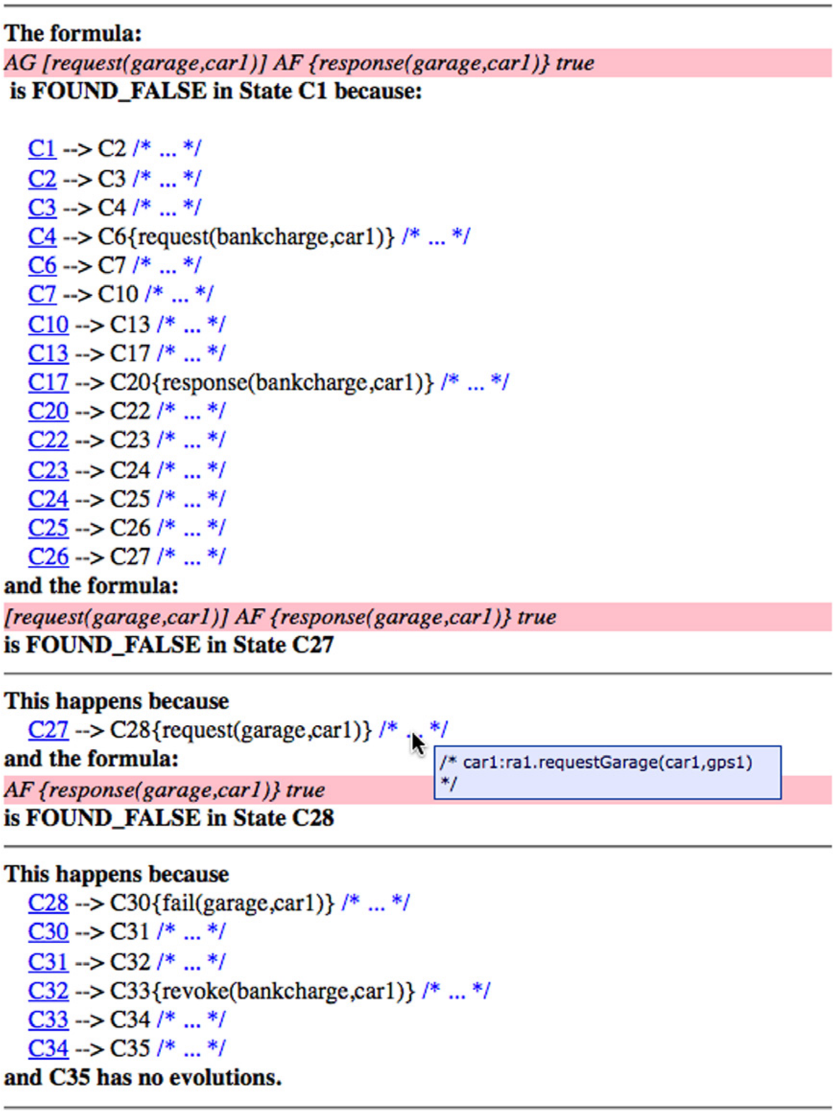
- The proof/counterexample is not (in general) based on one single execution path of the system, but it might be based on a subgraph of the abstract minimized L^2TS modelling the system.
- In general, not all states of the L^2TS that are needed by the proof/counterexample are *useful* from the point of view of the end user (designer).
- The information on the set of states of the L^2TS that is needed by the proof/counterexample is sometimes insufficient to produce usable feedback to the user: it may be necessary to provide feedback also on the subformula being evaluated when the L^2TS states are being explored.

For instance, consider a simple formula like (*AG predicate1*) $\vee$ (*AG predicate2*). If this formula does not hold, then its counterexample necessarily has the form of a pair of paths: one leading to a state in which *predicate1* does not hold and another leading to a state in which *predicate2* does not hold.

As a second example, consider the formula *EF predicate*. If this formula does not hold, then its counterexample coincides with the full state space of the system. It would however be completely pointless to provide the user with an exhaustive list of all the states for which the *predicate* does not hold. On the contrary, if the formula does hold, then the user might be interested in the sequence of steps that proves this.

Finally, as a third example, consider the formula *EF AG predicate*. If this formula holds for a certain system, then the user might be interested in the proof of the first part of this formula, *i.e.* an execution path which, starting from the initial state, leads to an intermediate state for which the subformula *AG predicate* holds. Once this intermediate state is identified, all the other states that are reachable from it belong to the proof of the subformula *AG predicate* and only add irrelevant noise and complexity to the original information. The *useful* part of the proof would be constituted by only a fragment of the full proof.

Now recall formula **F6** of Section 7, which does not hold for the automotive system. In Fig. 16, we show the counterexample produced by UMC, as it appears in the UMC web-based user interface. The node names mentioned in the counterexample are hyperlinks which, when followed, allow the user to observe all the concrete and abstract details of that system configuration. Furthermore, while abstract transition labels are always fully displayed on the right-hand side of the transitions, their corresponding underlying ground events (which are useful for understanding what exactly is happening in the evolutions of the ground model) are shown as dynamic tooltips appearing when the cursor is moved over the "`/*...*/`" regions. The explanation returned by UMC has indeed the form of a (partial) proof in the sense that not only the "witnessing" model fragment, but also the subformulae holding in the various substates, are put in evidence; moreover, only what are considered the *useful* parts of the explanation are shown.



```
The formula:
AG [request(garage,car1)] AF {response(garage,car1)} true
 is FOUND_FALSE in State C1 because:

   C1 --> C2 /* ... */
   C2 --> C3 /* ... */
   C3 --> C4 /* ... */
   C4 --> C6{request(bankcharge,car1)} /* ... */
   C6 --> C7 /* ... */
   C7 --> C10 /* ... */
   C10 --> C13 /* ... */
   C13 --> C17 /* ... */
   C17 --> C20{response(bankcharge,car1)} /* ... */
   C20 --> C22 /* ... */
   C22 --> C23 /* ... */
   C23 --> C24 /* ... */
   C24 --> C25 /* ... */
   C25 --> C26 /* ... */
   C26 --> C27 /* ... */
and the formula:
[request(garage,car1)] AF {response(garage,car1)} true
is FOUND_FALSE in State C27

This happens because
   C27 --> C28{request(garage,car1)} /* ... */
and the formula:                      /* car1:ra1.requestGarage(car1,gps1)
AF {response(garage,car1)} true       */
is FOUND_FALSE in State C28

This happens because
   C28 --> C30{fail(garage,car1)} /* ... */
   C30 --> C31 /* ... */
   C31 --> C32 /* ... */
   C32 --> C33{revoke(bankcharge,car1)} /* ... */
   C33 --> C34 /* ... */
   C34 --> C35 /* ... */
and C35 has no evolutions.
```

Fig. 16. Counterexample for formula **F6**.

10. UMC deployment

The UMC core consists of a command-line-oriented version of the model checker (*i.e.* the native UMC executable) which is a stand-alone program written in Ada and compiled for Windows/Linux/Solaris/MacOSX platforms. This core executable, once wrapped with a set of CGI scripts handled by a web server, becomes accessible via the web as an online web application [37]. In this way, it is easy to build a graphical HTML-oriented GUI and to integrate it with other external tools like graph visualizers and LTS minimizers.

UMC is routinely used for academic, research, and experimental purposes. Until now, the focus of development has been on the design of qualitative features one would desire for a verification tool, thus experimenting with various system modelling languages, logics, and user interfaces. It is currently not our purpose to transform UMC into a commercial development tool. For such a transformation we should take into consideration issues like strong code optimizations, scalability over massively large systems, exhaustive testing and validation issues—all issues outside our short-term plans and resources.

Improvements already under development include the automatic translation of UML4SOA designs to UMC models and the direct extraction of UMC models from standard UML-XMI models. Direct support of UML activity diagrams and the use of abstract interpretation techniques [7] to efficiently verify open systems are two other lines of research we plan to pursue in the future.

11. Discussion and conclusions

The roots of UCTL can be found in [16,17], where the logics called μ-ACTL$^+$ and μ-UCTL were first presented. These logics were already state/action based, but were defined as extensions of the full μ-calculus, so the until/unless operators were defined in terms of the fixpoint expressions. These logics (based on a very limited form of state predicates and basic actions) were already supported by the early prototypes of UMC (version 2.5), which however suffered from the computational complexity introduced by the full μ-calculus.

The logic UCTL in its current shape but in the *ground* version (*i.e.* without standard formatting of abstract labels) was presented for the first time in [2]. In this paper we give a more comprehensive presentation of the overall UMC framework, including the newly introduced on-the-fly ground / abstract exploration / verification mechanisms. The automotive case study used in Section 7 was presented for the first time in [3]; there UMC version 3.5 was used to check its properties, but the verification was conducted on its *ground* model without performing the abstraction of the L^{2}TS as described in Section 7. In [14,4] a specialized version of UCTL called SocL was presented; it is based on the current version of UCTL, but it is more specifically directed to service-oriented properties. In particular, in SocL we experimented for the first time the extension of UCTL with a formula parametrization mechanism, which is needed in the context of service-oriented systems to address the use of data correlation as a means to relate messages belonging to the same logical interaction.

Several studies have been done on model-checking UML-like state machines, based on translating the models into Promela for their subsequent verification using SPIN (hence using LTL). The earlier ones [24,31] were characterized by the support of extremely reduced subsets of UML statecharts (*e.g.* a single statechart, no local variables, no composite actions, no operation calls, no deferred events). More recently, new attempts have been presented (HUGO [21], PROCO [20]) which overcome most of the previous limits. Another approach, of which SMUL [13] is probably the best example, is based on translating UML statechart models into the input language of the open source NuSMV symbolic model checker. Other (commercial) approaches make use of translations into other frameworks like IF [28] or the VIS symbolic model checker [34]. Finally, a few attempts have been made to translate UML statecharts to Petri nets in order to apply the verification techniques developed for Petri nets [5,33]. However, the computational model of Petri nets is quite different from that of UML statecharts and achieving a reasonably complete and correct translation is quite difficult. Indeed, all the proposals we have seen make restrictive assumptions to achieve their goal, like not considering object attributes, local assignments, guards, deferred and non-deferred events in the various system substates, the higher priorities of more deeply nested transitions, and the higher priority of completion transitions over triggered transitions.

Our framework addresses roughly the same fragment of UML systems taken into consideration by the most recent (non-commercial) studies mentioned above, although we also support operation calls, an aspect which is rarely taken into consideration. Moreover, we strictly adhere to the standard UML semantics for concurrent states, an aspect from which it is common to find simplifying deviations (*e.g.* in [20,13]). Other functional and architectural differences w.r.t. those studies can be summarized as follows:

- The logic used to express system properties is a branching-time, state- and event-based temporal logic which is able to formalize both state-based properties in LTL-style and event-based properties in ACTL-style. We believe this kind of logic to be necessary to reason about the behaviour of UML systems, that are characterized by a rich structure of states (*e.g.* involving events queues and object attributes) together with a rich structure of events (messages can be sent, accepted, dispatched and discarded).
- The integration of the on-the-fly model checking of a logic formula, the interactive user-driven exploration of the state space, and the generation of abstract minimized behavioural slices of the system under analysis help the user to focus on the details of the behaviour of the system at the right level of abstraction. In particular, reasoning at different levels of abstraction on the same system is possible: for instance, we can verify an abstract (*e.g.* service-oriented) logic property, but then observe its counterexample at the *ground* level looking at all the possible details of the identified execution path. Moreover, once some defect in the design has been found we might be interested in verifying some formulae at the *ground* level. This is not to prove that an abstract high-level requirement holds, but to check whether a certain erroneous condition in the implementation occurs (*i.e.* using the model checker more as a debugger than as a system validator).
- Our approach is truly on-the-fly in the sense that no *a priori* bounds have to be established (*e.g.* on the events queues) before starting a verification. This allows the, sometimes partial, evaluation of possibly infinite systems. Clearly, for such systems no formula can be verified when an exploration of the full state space is needed. However, counterexamples can be found also for systems which by their nature do have the possibility to diverge.

Acknowledgements

The work presented in this paper has been partially funded by the EU project Sensoria (IST-2005-016004) and the Italian project D-ASAP.

We are grateful to Nora Koch from Cirquent GmbH and LMU Munich, Germany, for our joint work on the automotive case study.

We thank the anonymous reviewers for their many useful comments and suggestions that have improved the paper considerably.

References

[1] http://fmt.isti.cnr.it/umc/V3.6/EXAMPLES/00-automotive-SCP.umc.

[2] M.H. ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti, An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications, in: S. Leue, P. Merino (Eds.), Proceedings of the 12th International Workshop on Formal Methods for Industrial Critical Systems, FMICS'07, Berlin, Germany, in: LNCS, vol. 4916, Springer-Verlag, Berlin, 2008, pp. 133–148.

[3] M.H. ter Beek, S. Gnesi, N. Koch, F. Mazzanti, Formal verification of an automotive scenario in service-oriented computing, in: Proceedings of the 30th International Conference on Software Engineering, ICSE'08, Leipzig, Germany, ACM Press, New York, 2008, pp. 613–622.

[4] M.H. ter Beek, F. Mazzanti, S. Gnesi, CMC-UMC: a framework for the verification of abstract service-oriented properties, in: Proceedings of the 24th Annual ACM Symposium on Applied Computing, SAC'09, Honolulu, HI, USA, ACM Press, New York, 2009, pp. 2111–2117.

[5] S. Bernardi, S. Donatelli, J. Merseguer, From UML sequence diagrams and statecharts to analysable Petri net models, in: Proceedings of the 3rd International Workshop on Software and Performance, WOSP'02, Rome, Italy, ACM Press, New York, 2002, pp. 35–45.

[6] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: R. Cleaveland (Ed.), Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS'99, Amsterdam, The Netherlands, in: LNCS, vol. 1579, Springer-Verlag, Berlin, 1999, pp. 193–207.

[7] P. Cousot, R. Cousot, Temporal abstract interpretation, in: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'00, Boston, Massachusetts, USA, ACM Press, New York, 2000, pp. 12–25.

[8] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems 8 (2) (1986) 244–263.

[9] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 1999.

[10] E.M. Clarke, S. Jha, Y. Lu, H. Veith, Tree-like counterexamples in model checking, in: Proceedings of the 17th IEEE Symposium on Logic in Computer Science, LICS'02, Copenhagen, Denmark, IEEE Press, 2002, pp. 19–29.

[11] R. De Nicola, F.W. Vaandrager, Actions versus state based logics for transition systems, in: I. Guessarian (Ed.), Semantics of Systems of Concurrent Processes—Proceedings of the LITP Spring School on Theoretical Computer Science, La Roche Posay, France, in: LNCS, vol. 469, Springer-Verlag, Berlin, 1990, pp. 407–419.

[12] R. De Nicola, F.W. Vaandrager, Three logics for branching bisimulation, Journal of the ACM 42 (2) (1995) 458–487.

[13] J. Dubrovin, T. Junttila, Symbolic model checking of hierarchical UML state machines, in: J. Billington, Z. Duan, M. Koutny (Eds.), Proceedings of the 8th International Conference on Application of Concurrency to System Design, ACSD'08, Xian, China, IEEE Press, 2008, pp. 108–117.

[14] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, F. Tiezzi, A model checking approach for verifying COWS specifications, in: J.L. Fiadeiro, P. Inverardi (Eds.), Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering, FASE'08, Budapest, Hungary, in: LNCS, vol. 4961, Springer-Verlag, Berlin, 2008, pp. 230–245.

[15] R.J. van Glabbeek, The linear time-branching time spectrum, in: J.C.M. Baeten, J.W. Klop (Eds.), Proceedings of the 1st International Conference on Concurrency Theory, CONCUR'90, Amsterdam, The Netherlands, in: LNCS, vol. 458, Springer-Verlag, Berlin, 1990, pp. 278–297.

[16] S. Gnesi, F. Mazzanti, On the fly model checking of communicating UML state machines, in: Proceedings of the 2nd International Conference on Software Engineering Research, Management and Applications, SERA'04, Los Angeles, CA, USA, 2004, pp. 331–338.

[17] S. Gnesi, F. Mazzanti, A model checking verification environment for UML statecharts. Presented at the XLIII Annual Italian Conference AICA, Udine, 2005.

[18] A. Gurfinkel, M. Chechik, Proof-like counter-examples, in: H. Garavel, J. Hatcliff (Eds.), Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'03, Warsaw, Poland, in: LNCS, vol. 2619, Springer-Verlag, Berlin, 2003, pp. 160–175.

[19] M. Hennessy, R. Milner, Algebraic laws for nondeterminism and concurrency, Journal of the ACM 32 (1) (1985) 137–161.

[20] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, I. Porres, Model checking dynamic and hierarchical UML state machines, in: Proceedings of the 3rd International Workshop on Model Development, Validation and Verification, MoDeVa'06, Genoa, Italy, 2006, pp. 94–110.

[21] A. Knapp, S. Merz, Ch. Rauh, Model checking—timed UML state machines and collaborations, in: W. Damm, E.-R. Olderog (Eds.), Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'02, Oldenburg, Germany, in: LNCS, vol. 2469, Springer-Verlag, Berlin, 2002, pp. 395–414.

[22] N. Koch, D. Berndl, Requirements modelling and analysis of selected scenarios of the automotive case study. Sensoria Deliverable 8.2a, September 2007. Available via [35].

[23] N. Koch, P. Mayer, R. Heckel, L. Gönczy, C. Montangero, UML for service-oriented systems. Sensoria Deliverable 1.4a, September 2007. Available via [35].

[24] D. Latella, I. Majzik, M. Massink, Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker, Formal Aspects of Computing 711 (6) (1999) 637–664.

[25] F. Mazzanti, Designing UML models with UMC. Technical Report 2009-TR-043, Istituto di Scienza e Tecnologie dell'Informazione, CNR, 2009. Available via http://fmt.isti.cnr.it/umc/V3.6/UMC-Models-v36.pdf.

[26] R. Meolic, T. Kapus, Z. Brezocnik, ACTLW: An action-based computation tree logic with unless operator, Information Sciences 178 (6) (2008) 1542–1557.

[27] M. Müller-Olm, D.A. Schmidt, B. Steffen, Model-checking—a tutorial introduction, in: A. Cortesi, G. Filé (Eds.), Proceedings of the 6th International Symposium on Static Analysis, SAS'99, Venice, Italy, in: LNCS, vol. 1694, Springer-Verlag, Berlin, 1999, pp. 330–354.

[28] I. Obe, S. Graf, I. Ober, Validation of UML models via a mapping to communicating extended timed automata, in: S. Graf, L. Mounier (Eds.), Proceedings of the 11th International SPIN Workshop on Model Checking Software, SPIN'04, Barcelona, Spain, in: LNCS, vol. 2989, Springer-Verlag, Berlin, 2004, pp. 127–145.

[29] OMG, UML Superstructure 2.1.2. http://www.uml.org/.

[30] A. Pnueli, Linear and branching structures in the semantics and logics of reactive systems, in: W. Brauer (Ed.), Proceedings of the 12th International Colloquium on Automata, Languages and Programming, ICALP'85, Nafplion, Greece, in: LNCS, vol. 194, Springer-Verlag, Berlin, 1985, pp. 15–32.

[31] I. Porres, Modeling and analyzing software behavior in UML. Ph.D. thesis, Åbo Akademi University, Turku, 2001.

[32] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, Reading, MA, 1998.

[33] J.A. Saldhana, S.M. Shatz, Z. Hu, Formalization of object behavior and interactions from UML models, International Journal of Software Engineering and Knowledge Engineering 11 (6) (2001) 643–673.

[34] I. Schinz, T. Toben, C. Mrugalla, B. Westphal, The rhapsody UML verification environment, in: Proceedings of the 2nd International Conference on Software Engineering and Formal Methods, SEFM'04, Beijing, China, IEEE Press, 2004, pp. 174–183.

[35] EU IST-FET Integrated Project Sensoria. http://www.sensoria-ist.eu/.

[36] Spec patterns. http://patterns.projects.cis.ksu.edu/.

[37] UMC online web application. http://fmt.isti.cnr.it/umc/.