An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications*

Maurice H. ter Beek[1], A. Fantechi[1,2], S. Gnesi[1], and F. Mazzanti[1]

[1] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, Pisa, Italy
{maurice.terbeek,stefania.gnesi,franco.mazzanti}@isti.cnr.it
[2] Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy
fantechi@dsi.unifi.it

Abstract. In this paper we present an action/state-based logical framework for the analysis and verification of complex systems, which relies on the definition of doubly labelled transition systems. The defined temporal logic, called UCTL, combines the action paradigm—classically used to describe systems using labelled transition systems—with predicates that are true over states—as captured when using Kripke structures as semantic model. An efficient model checker for UCTL has been realized, exploiting an on-the-fly algorithm. We then show how to use UCTL, and its model checker, in the design phase of an asynchronous extension of the communication protocol SOAP, called aSOAP. For this purpose, we describe aSOAP as a set of communicating UML state machines, for which a semantics over doubly labelled transition systems has been provided.

1 Introduction

Complex 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 complex 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. An example are UML state diagrams, which are used more and more in industry to specify the behaviour of (software) systems, though often without caring much for their formal aspects. Also the specification of Service-Oriented Applications has seen several applications of UML state diagrams [25]. What is missing in order to use in full specification techniques that allow one to specify both events and state changes, is the availability of a formal framework in which desired properties can subsequently be proved over the specification, with the support of specific verification tools.

* This work has been partially funded by the EU project SENSORIA (IST-2005-016004) and by the Italian project TOCAI.IT.

In this paper, we aim to fill this gap by presenting the action/state-based temporal logic UCTL, which allows one to both specify the basic properties that a state should satisfy and to combine these basic predicates with advanced temporal operators dealing with the events performed. As was done for CTL and ACTL in the past, we consider a fragment of a doubly labelled temporal logic, interpreted over doubly labelled structures: This fragment is UCTL, which preserves the property shared by CTL and ACTL of having an explicit local model-checking algorithm in linear time. The semantic domain of UCTL is doubly labelled transition systems [11]. A prototypical on-the-fly model checker, called UMC, has been developed for UCTL: The tool allows the efficient verification of UCTL formulae that define action- and state-based properties.

In recent years, several logics that allow one to express both action-based and state-based properties have been introduced, for many different purposes. An event- and state-based temporal logic for Petri nets is given in [17]. In [16], a modal temporal logic without a fixed-point operator and interpreted over so-called Kripke modal transition systems (a modal version of doubly labelled transition systems) is defined. In [4,6], a state/event extension of LTL is presented, together with a model-checking framework whose formulae are interpreted over so-called labelled Kripke structures (essentially doubly labelled transition systems). Finally, in [5], this linear-time temporal logic is extended to a universal branching-time temporal logic. The latter logics are used extensively to verify software systems. The advantage of all such logics lies in the ease of expressiveness of properties that in pure action-based or pure state-based logics can be quite cumbersome to write down. Moreover, their use often results in a reduction of the state space, the memory use and the time spent for verification. Obviously, the real gain depends—as always—on the specific system under scrutiny.

To conclude, we present a case study that shows the use of UCTL and its model checker UMC in the design phase of aSOAP, which is an asynchronous extension of the web service communication protocol SOAP. Mobile communication networks typically are unstable, since terminal devices can dynamically change reachability status during their lifetime. In Service-Oriented Architectures, asynchronous service invocation is often the more suitable paradigm for the choreography and orchestration of their mobile components. Hence, there is a need for communication protocols that can manage asynchronous communication also in the presence of unstable network connections. Formal modelling and analysis of such protocols is a first step towards the successful implementation and evaluation of reliable Service-Oriented Applications. For this purpose, we describe aSOAP as a set of communicating UML state machines, for which a semantics over doubly labelled transition systems has been provided, express several behavioural properties on this UML model of aSOAP in UCTL and verify them with UMC.

The paper is organized as follows. Some preliminary definitions are given in Section 2. In Section 3, we present the syntax and semantics of UCTL, while in Section 4 we describe its model checker UMC. The case study illustrating their use is presented in Section 4. Finally, Section 5 concludes the paper.

2 Preliminaries

In this section, we define the basic notations and terminology used in the sequel.

Definition 1 (Labelled Transition System). *A* Labelled Transition System *(LTS for short) is a quadruple* (Q, q_0, Act, R)*, where:*

- Q *is a set of states;*
- $q_0 \in Q$ *is the initial state;*
- Act *is a finite set of observable events (actions) with* e *ranging over* Act*,* α *ranging over* 2^{Act} *and* ϵ *denoting the empty set;*
- $R \subseteq Q \times 2^{Act} \times Q$ *is the transition relation; instead of* $(q, \alpha, q') \in R$ *we may also write* $q \xrightarrow{\alpha} q'$*.*

Note that the main difference between this definition of LTSs and the classical one is the labelling of the transitions: we label transitions by sets of events rather than by single (un)observable events. This extension allows the transitions from one state to another to represent sets of actions without the need of intermediate states, which has proved to be useful when modelling, e.g., UML state diagrams.

Another extension is to label states with atomic propositions, like the concept of doubly labelled transition systems [11], again extended as in Definition 1.

Definition 2 (Doubly Labelled Transition System). *A* Doubly Labelled Transition System *(L^2TS for short) is a quintuple* (Q, q_0, Act, R, AP, L)*, where:*

- (Q, q_0, Act, R) *is an LTS;*
- AP *is a set of atomic propositions with* p *ranging over* AP*;* p *will typically have the form of an expression like* $VAR = value$*;*
- $L : Q \longrightarrow 2^{AP}$ *is a labelling function that associates a subset of* AP *to each state of the LTS.*

The L^{2}TSs thus obtained are very similar to so-called *Kripke transition systems* [19]. The latter are defined as an extension of Kripke structures by a labelling over transitions.

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

Definition 3 (Bisimulation). *Let* $A_1 = (Q_1, q_{0_1}, Act, \rightarrow_1, AP_1, L_1)$ *and* $A_2 = (Q_2, q_{0_2}, Act, \rightarrow_2, AP_2, L_2)$ *be two L^2TSs 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 $\mathcal{B}$ *that relates* 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* $\alpha \in 2^{Act}$*:*

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

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

The usual notions of simulation preorder or weak (observational) equivalence can be defined analogously.

LTSs and Kripke structures can be lifted to L^{2}TSs in a straightforward manner. An LTS $T = (Q, q_0, Act, R)$ can be lifted to an L^{2}TS A_T, on the same set of states and maintaining the same transition relation, in the following way: $A_T = (Q, q_0, Act, R, \varnothing, L)$, where for all $q \in Q$: $L(q) = \varnothing$.

A Kripke structure $K = (Q, q_0, R, AP, L)$ can be lifted to an L^{2}TS A_K, on the same set of states and maintaining the same labelling function, in the following way: $\mathcal{A}_K = (Q, q_0, \{\epsilon\}, R', AP, L)$, where for all $(q, q') \in R$: $(q, \epsilon, q') \in R'$.

3 The Action/State-Based Temporal Logic UCTL

In this section, we present the syntax and semantics of UCTL. This temporal logic, *action and state based*, allows one to reason on state properties as well as to describe the behaviour of systems that perform actions during their lifetime. UCTL includes both the branching-time action-based logic ACTL [10,11] and the branching-time state-based logic CTL [7].[1]

Before defining the syntax of UCTL, we introduce an auxiliary logic of events.

Definition 4 (Event formulae). *Let Act be a set of observable events. Then the language of event formulae on* $Act \cup \{\tau\}$ *is defined as follows:*

$$\chi ::= tt \mid e \mid \tau \mid \neg\chi \mid \chi \wedge \chi$$

Definition 5 (Event formulae semantics). *The satisfaction relation* $\models$ *for event formulae of the form* $\alpha \models \chi$ *is defined over sets of events as follows:*

$\alpha \models tt$ *holds always;*
$\alpha \models e$ *iff* $\alpha = \{e_1, \ldots, e_n\}$ *and there exists an* $i \in \{1, \ldots, n\}$ *such that* $e_i = e$;
$\alpha \models \tau$ *iff* $\alpha = \varnothing$;
$\alpha \models \neg\chi$ *iff not* $\alpha \models \chi$;
$\alpha \models \chi \wedge \chi'$ *iff* $\alpha \models \chi$ *and* $\alpha \models \chi'$.

As usual, *ff* abbreviates $\neg tt$ and $\chi \vee \chi'$ abbreviates $\neg(\neg\chi \wedge \neg\chi')$.

Definition 6 (Syntax of UCTL)

$$\phi ::= true \mid p \mid \neg\phi \mid \phi \wedge \phi' \mid A\pi \mid E\pi$$
$$\pi ::= X_\chi \phi \mid \phi\, {}_\chi U\, \phi' \mid \phi\, {}_\chi U_{\chi'}\, \phi' \mid \phi\, {}_\chi W\, \phi' \mid \phi\, {}_\chi W_{\chi'}\, \phi'$$

[1] Note that ACTL is also used to denote the universal fragment of CTL, originally called ∀CTL in [8]. For easy of writing, ∀CTL was changed to ACTL, thus generating a conflict with the previously introduced acronym ACTL for Action-based CTL.

State formulae *are ranged over by* ϕ, path formulae *are ranged over by* π*,* A *and* E *are* path quantifiers*, and* X*,* U *and* W *are the indexed* next*,* until *and* weak until *operators.*[2]

In linear-time temporal logic (LTL), the formula $\phi\ W\,\psi$ can be obtained by deriving it from the until (U) and the 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 applicable in UCTL since disjunction or conjunction of path formulae is not expressible according to the UCTL syntax, and the same holds for any pure branching-time temporal logic.

To define the semantics of UCTL, we need the notion of a path in an L^{2}TS.

Definition 7 (Path). *Let* $\mathcal{A} = (Q, q_0, Act, R, AP, L)$ *be an* L^2TS *and let* $q \in Q$*.*

- σ *is a* path *from* q *if* $\sigma = q$ *(the* empty path *from* q*) or* σ *is a (possibly infinite) sequence* $(q_0, \alpha_1, q_1)(q_1, \alpha_2, q_2)\cdots$ *with* $(q_{i-1}, \alpha_i, q_i) \in R$ *for all* $i > 0$*.*
- *The concatenation of paths* σ_1 *and* σ_2*, denoted by* $\sigma_1\sigma_2$*, is a partial operation, defined only if* σ_1 *is finite and its final state coincides with the first state of* σ_2*. Concatenation is associative and has identities:* $\sigma_1(\sigma_2\sigma_3) = (\sigma_1\sigma_2)\sigma_3$ *and if* q_0 *is the first state of* σ *and* q_n *is its final state, then* $q_0\sigma = \sigma q_n = \sigma$*.*
- *A path* σ *is said to be* maximal *if it is either an infinite sequence or it is a finite sequence whose final state has no successor states.*
- *The* length *of a path* σ *is denoted by* $|\sigma|$*. If* σ *is an infinite path, then* $|\sigma| = \omega$*. If* $\sigma = q$*, then* $|\sigma| = 0$*. If* $\sigma = (q_0, \alpha_1, q_1)(q_1, \alpha_2, q_2)\cdots(q_n, \alpha_{n+1}, q_{n+1})$*, for some* $n \geq 0$*, then* $|\sigma| = n + 1$*. Moreover, the* i^{th} *state in such a path, i.e.* q_i*, is denoted by* $\sigma(i)$*.*

Definition 8 (Semantics of UCTL). *The satisfaction relation for 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* $q \models \phi'$*;*
$q \models A\pi$ *iff* $\sigma \models \pi$ *for all paths* σ *such that* $\sigma(0) = q$*;*
$q \models E\pi$ *iff there exists a path* σ *with* $\sigma(0) = q$ *such that* $\sigma \models \pi$*;*
$\sigma \models X_\chi\phi$ *iff* $\sigma = (\sigma(0), \alpha_1, \sigma(1))\sigma'$*, and* $\alpha_1 \models \chi$*, and* $\sigma(1) \models \phi$*;*
$\sigma \models [\phi\ {}_\chi U\,\phi']$ *iff there exists a* $j \geq 0$ *such that* $\sigma(j) \models \phi'$ *and for all* $0 \leq i < j$*:* $\sigma = \sigma'(\sigma(i), \alpha_{i+1}, \sigma(i{+}1))\sigma''$ *implies* $\sigma(i) \models \phi$ *and* $\alpha_{i+1} = \epsilon$ *or* $\alpha_{i+1} \models \chi$*;*
$\sigma \models [\phi\ {}_\chi U_{\chi'}\,\phi']$ *iff there exists a* $j \geq 1$ *such that* $\sigma = \sigma'(\sigma(j-1), \alpha_j, \sigma(j))\sigma''$ *and* $\sigma(j) \models \phi'$ *and* $\sigma(j-1) \models \phi$ *and* $\alpha_j \models \chi'$*, and for all* $0 < i < j$*:* $\sigma = \sigma'_i(\sigma(i-1), \alpha_i, \sigma(i))\sigma''_i$ *implies* $\sigma(i-1) \models \phi$ *and* $\alpha_i = \epsilon$ *or* $\alpha_i \models \chi$*;*
$\sigma \models [\phi\ {}_\chi W\,\phi']$ *iff either there exists a* $j \geq 0$ *such that* $\sigma(j) \models \phi'$ *and for all* $0 \leq i < j$*:* $\sigma = \sigma'(\sigma(i), \alpha_{i+1}, \sigma(i{+}1))\sigma''$ *implies* $\sigma(i) \models \phi$ *and* $\alpha_{i+1} = \epsilon$ *or* $\alpha_{i+1} \models \chi$*; or for all* $i \geq 0$*:* $\sigma = \sigma'(\sigma(i), \alpha_{i+1}, \sigma(i+1))\sigma''$ *implies* $\sigma(i) \models \phi$ *and* $\alpha_{i+1} = \epsilon$ *or* $\alpha_{i+1} \models \chi$*;*

[2] Note that, differently from the original ACTL logic, in UCTL the operator $X_\chi\phi$ can be derived as $false\ {}_{false}U_\chi\ \phi$.

$\sigma \models [\phi\ {}_{\chi}W_{\chi'}\ \phi']$ *iff either there exists a* $j \geq 1$ *such that* $\sigma = \sigma'(\sigma(j{-}1), \alpha_j, \sigma(j))\sigma''$ *and* $\sigma(j) \models \phi'$ *and* $\sigma(j-1) \models \phi$ *and* $\alpha_j \models \chi'$*, and for all* $0 < i < j$*:* $\sigma = \sigma'_i(\sigma(i-1), \alpha_i, \sigma(i))\sigma''_i$ *implies* $\sigma(i-1) \models \phi$ *and* $\alpha_i = \epsilon$ *or* $\alpha_i \models \chi$*; or for all* $i > 0$*:* $\sigma = \sigma'_i(\sigma(i-1), \alpha_i, \sigma(i))\sigma''_i$ *implies* $\sigma(i-1) \models \phi$ *and* $\alpha_i = \epsilon$ *or* $\alpha_i \models \chi$*.*

It is now straightforward to obtain a set of derived operators for UCTL, such as:

$< \chi > \phi$ stands for $E[true\ {}_{\tau}U_{\chi}\ \phi]$;
$[\chi]\phi$ stands for $\neg < \chi > \neg\phi$;
$EF\phi$ stands for $E[true\ {}_{true}U\phi]$;
$AG\phi$ stands for $\neg EF\neg\phi$;

Operators $< \chi > \phi$ and $[\chi]\phi$ are the diamond and box modalities, respectively, of the Hennessy-Milner logic [15]. The meaning of $EF\phi$ is that ϕ must be true sometimes in a possible future; that of $AG\phi$ is that ϕ must be true always.

The logic UCTL is *adequate* with respect to strong bisimulation equivalence on L^{2}TSs. Adequacy [21] 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 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.

Proof (sketch). Let A_1 and A_2 be two L^{2}TSs. Note that neither the existential nor the universal next operator of a UCTL formula can distinguish the transition $q_1 \xrightarrow{a} q'_1$ in A_1 from the two transitions $q_2 \xrightarrow{a} q'_2$ and $q_2 \xrightarrow{a} q''_2$, with $\{q'_2, q''_2\}$ bisimilar to q'_1, which makes q_1 and q_2 bisimilar as well. The same can be said for the atomic predicates, since the labelling of bisimilar states is the same, as well as for the until operators, which just follow recursively the transition relation. Hence, if two L^{2}TSs are bisimilar, then no distinguishing formula can be found.

On the other hand, if two L^{2}TSs are not bisimilar, then applying recursively the definition of bisimulation to the pair of initial states $\{q_{0_1}, q_{0_2}\}$ of the two L^{2}TSs implies that we eventually end up in at least one pair of states $\{q_1, q_2\}$ with the following characteristics: A sequence $a_1, a_2, \ldots, a_n$ of actions leads from q_{0_1} and q_{0_2} to q_1 and q_2, respectively, and q_1 and q_2 can be differentiated either (but not necessarily exclusively) by their labelling or by an outgoing transition.

In the former case, there exists at least one predicate p such that $p \in L(q_1)$ but $p \notin L(q_2)$ (or vice versa), which means that $X_{a_1}X_{a_2}\cdots X_{a_n}p$ is a distinguishing formula for the two L^{2}TSs.

In the latter case, there exists at least one transition $q_1 \xrightarrow{a} q'_1$, for some action a and state q'_1, while there exists no transition labelled with a from q_2 to some state q'_2 (or vice versa), which means that $X_{a_1}X_{a_2}\cdots X_{a_n}X_a\ true$ is a distinguishing formula for the two L^{2}TSs. $\square$

Starting from the syntax of UCTL, it is possible to derive both CTL [7] and ACTL [10,11] by simply removing the action or the state component, respectively: Given a Kripke structure $K = (Q, q_0, R, AP, L)$ that has been lifted to an L^{2}TS $A_K = (Q, q_0, Act, R', AP, L)$, a CTL formula ϕ and a state $q \in Q$, it follows that

$$q \models_K \phi \ \text{ iff } \ q \models_{A_K} \phi',$$

where ϕ' is a UCTL formula which is syntactically identical to ϕ, apart from the fact that all occurrences of $X\psi'$ have been replaced by $X_{true}\psi'$ and all occurrences of $\psi U \psi'$ have been replaced by $\psi\ _{true}U\psi'$.

Given an LTS $T = (Q, q_0, Act \cup \tau, R)$ that has been lifted to an L^{2}TS $A_T = (Q, q_0, Act, R, AP, L)$, an ACTL formula ϕ and a state $q \in Q$, it follows that

$$q \models_T \phi \ \text{ iff } \ q \models_{A_T} \phi',$$

where ϕ' is a UCTL formula which is syntactically identical to ϕ, apart from the fact that all occurrences of $X_{true}\psi$ are replaced by $X_{\neg\tau}\psi$.[3]

4 The UCTL Model Checker UMC

We have developed an on-the-fly model checking tool for UCTL, called UMC [18].

The big 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 [2,12,24]. This type of model checking is also called local [9], in contrast to global model checking [7], in which the whole state space is explored to check the validity of a formula.

The basic idea behind UMC is that, given a state of an L^{2}TS, the validity of a UCTL formula on that state can be evaluated by analyzing the transitions allowed in that state, and by analyzing the validity of some subformula in only some of the next reachable states, all this in a recursive way. The following simplified schema gives an idea of the algorithmic structure of the evaluation process: `F` denotes the UCTL formula (or subformula) to be evaluated, `Start` denotes the state in which the (recursive) evaluation of `F` was started and `Current` denotes the current state in which the evaluation of `F` is being continued.[4]

[3] The original definition of ACTL [10] is based on a definition of LTSs in which a transition label can be a single (un)observable action. Hence, to be precise, we actually need to use here a different definition of lifting an LTS to an L^{2}TS, namely one in which τ-transitions are replaced by ϵ-transitions.

[4] The given schema can be extended to handle also max and min fixpoint operators, by replacing the single `Start` state with a vector of states according to the fixpoint nesting depth of the formula. UMC actually supports this extension, but with drawbacks in the level of optimizations performed.

```
Evaluate (F : Formula, Start : State, Current : State) is
  if we have already done this computation and its result is available,
  i.e. (⟨F, Start, Current⟩ → Result) has already been established then
      return the already known result;
  else if we were already computing the value of exactly this computation,
  i.e. (⟨F,Start,Current⟩ → inprogress) has already been established then
      return true or false depending on max or min fixed point semantics;
  else
      keep track that we started to compute the value of this computation,
      i.e. set (⟨F, Start, Current⟩ → inprogress);
      foreach subformula F′ and next state S′ to be computed do loop
         if $F \neq F$ (i.e. this is a syntactically nested subformula) then
            call Evaluate(F′, S, S′);
         else if $F = F$ (i.e. this is just a recursive evaluation of F) then
            call Evaluate(F, Start, S′);
         end
         if the result of the call suffices to establish the final result then
            exit from the loop;
         end
      end loop
      At this point we have in any case a final result. We keep track of the
      result of this compuation (e.g. set (⟨F, Start, Current⟩ → Result)).
      return the final result;
  end
end Evaluate;
```

This simplified schema can be extended with appropriate data-collection activities in order to be able to produce, in the end, also a clear and detailed explanation of the returned result.

In case of infinite state spaces, the above schema 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 "depth-first" recursive structure of the algorithm. The solution taken to solve this problem consists of adopting a bounded model-checking approach [3], i.e. the evaluation is started assuming a certain value as limit of the maximum depth of the evaluation. In this case, if a formula is given as result of the evaluation within the requested depth, then the result holds for the whole system; otherwise the maximum depth is increased and the evaluation is subsequently retried (preserving all useful subresults that were already found). This approach, initially introduced in UMC to overcome the problem of infinite state machines, happens to be quite useful also for another reason. By setting a small initial maximum depth and a small automatic increment of this limit at each re-evaluation failure, once we finally find a result then we have a reasonable (almost minimal) explanation for it, and this is very useful also in the case of finite states machines.

Given a formula F, the upper bound on the number of necessary computations steps, identifiable by the triple ⟨subformula, startstate, currentstate⟩ apparently tends to grow quadratically with respect to the number of system states. A linear complexity of the above model-checking algorithm can be achieved by performing several optimizations in the management of the "archive" of performed computations.[5] In particular, we consider the property that if $(\langle \texttt{F}, \texttt{State}, \texttt{State} \rangle \rightarrow \texttt{Result})$ is true, then for any other $\texttt{State}'$, $(\langle \texttt{F}, \texttt{State}', \texttt{State} \rangle \rightarrow \texttt{Result})$ is also true. Moreover, when $(\langle \texttt{F}, \texttt{State}, \texttt{State} \rangle \rightarrow \texttt{Result})$ is established, for all the recursive subcomputations of $\texttt{F}$ that have the form $(\langle \texttt{F}, \texttt{State}, \texttt{State}' \rangle \rightarrow \texttt{Result})$ it is also true that $(\langle \texttt{F}, \texttt{State}', \texttt{State}' \rangle \rightarrow \texttt{Result})$ can be considered to hold.

The development of UMC is still in progress and a prototypical version is being used internally at ISTI–CNR for academic and experimental purposes. Until now, the focus of the development has been on the design of the kind of qualitative features one would desire for such a tool, experimenting with various logics, system modelling languages and user interfaces. So far there has been no official public release of the tool, even if the current prototype can be experimented via a web interface at the address http://fmt.isti.cnr.it/umc/.

UMC verifies properties defined over a set of communicating UML state machines [22,20]. We used UML as particular formal method since it has become the de facto industrial standard for modelling and documenting software systems. The UML semantics associates a state machine to each object in a system design, while the system's behaviour is defined by the possible evolutions of the resulting set of state machines that may communicate by exchanging signals. All these possible system evolutions are formally represented as an L^2TS, in which the states represent the various system configurations and the transitions represent the possible evolutions of a system configuration. In this L^2TS, states are labelled with the observed structural properties of the system configurations (e.g. active substates of objects, values of object attributes, etc.), while transitions are labelled with the observed properties of the system evolutions (e.g. which is the evolving object, which are the executed acions, etc.).

5 aSOAP: A Case Study

The particular case study we describe here has as main objective to define a variant of SOAP [26] supporting asynchronous communications, driven step-by-step by the results of a formal analysis. This approach thus contrasts with the usual approach of performing analysis on an already specified protocol to verify its correctness. The development of aSOAP is ongoing joint work with Telecom Italia. Some initial modelling and verification results have been presented in [1].

The domain of the case study is the definition of a SOAP-based protocol supporting asynchronous interactions, i.e. interactions different from the usual

[5] In [1,13,14], a less restricted logic (μ-UCTL) was defined and used in previous versions of UMC. Essentially based on the full μ-calculus, μ-UCTL was still defined over doubly labelled structures, but in that case the system transitions were labelled with sequences of events rather than with sets. Moreover, since the model-checking algorithm lacked the necessary optimizations, the complexity of the evaluation of alternation-free formulae still had a quadratic complexity.

synchronous "request-response" interactions supported by the available SOAP implementations based on HTTP. For the following reasons, asynchronous interactions are highly relevant in the delivery of telecommunication services:

– a service logic is triggered/activated by events produced, in an asynchronous way, by the network/special resources, or must react to such events during the execution of a service instance;
– requests produced by a service logic to a network/special resource may result in long computations (e.g. the set-up of a call), which might also require the involvement of end users;
– some service logic components may not be reachable (e.g. the ones deployed on mobile terminals), e.g. due to the temporary absence of communication.

The final objective is thus to formally define aSOAP as a protocol that is able to address most of these situations. We consider the following requirements.

Backward compatibility
– aSOAP must be compatible with SOAP v1.2 on HTTP;
– aSOAP must have limited impact on clients, i.e. clients that need no support for asynchronous interactions must be usual SOAP clients, working in request-response mode, while clients that do need such support should introduce only very limited variations w.r.t. normal SOAP requests.

Reachability
– aSOAP must be able to deal with the unreachability of the servers (e.g. due to the lack of connectivity);
– aSOAP must be able to deal with the case in which a server cannot return a (provisional or final) response due to the lack of connectivity;
– aSOAP must be able to deal with the case in which a (provisional or final) response cannot be returned to a client due to the lack of connectivity.

Message Exchange Patterns
– aSOAP must be able to deal with requests that require the servers to perform some long-running computation (longer than the HTTP timeout) before producing any results;
– aSOAP must be able to deal with requests with multiple responses.

We envision aSOAP to operate in a Client-Server architecture with an additional web service Proxy placed in between the Client and Server side. This Proxy must guarantee that various attempts to contact either side are made in case of temporary unavailability of the respective side. Moreover, aSOAP requires that a Client, whenever it is willing to accept the possibility of an asynchronous response to its request, sends to the Proxy not only its request but also the URL at which it would like to receive the response. We consider this URL to be the address of a generic "SOAP listener" and we assume the application level to be equipped with a mechanism capable of receiving SOAP messages at this URL.

Before discussing some aspects of our formal specification of aSOAP, we first list the assumptions that are part of the design of aSOAP.

- The Proxy is always reachable by both the Client and the Server, whenever they have an active connection;
- If the Client is willing to accept an asynchronous response to its SOAP invocation, then it inserts in the SOAP header the URL of the SOAP listener where it wants to receive the response;
- The URL in the header of an asynchronous SOAP invocation is the address of a generic SOAP listener and the application level is equipped with a mechanism for receiving SOAP messages at this URL;
- Upon receiving an asynchronous SOAP invocation from the Client, the Proxy generates a request identifier ReqId that uniquely identifies the Client's SOAP invocation in further communications.

During several sessions between ISTI–CNR and Telecom Italia we discussed our design and developed our formalisation of aSOAP in detail. In order to facilitate the discussions about the behaviour of the various use case scenarios of aSOAP, we decided upon a separate message sequence chart for each such a scenario. Finally, all these scenarios were translated into an operational model, in which the following concrete modelling choices were adopted:

- All SOAP invocations are asynchronous, i.e. we abstract from the synchronous SOAP invocations that only serve to guarantee backward compatibility with SOAP v1.2;
- The URL in the header of a SOAP message is identified with the Client, i.e. each Client is seen as just a listener of asynchronous SOAP invocations;
- A system model is constituted by a Server (and its subthreads), a Proxy (and its subthreads) and a fixed (configurable) number of Clients;
- The Proxy and the Server may activate at most a fixed (configurable) number of parallel subthreads;
- With the Client or the Server unreachable, the Proxy attempts to contact them up to a configurable number of times;
- The Client issues a single SOAP invocation and then terminates.[6]

For a complete discussion on these modelling choices, we refer the reader to [1].

Specifying the formal model of aSOAP as a set of communicating UML state machines has allowed us to express behavioural properties of our aSOAP model in UCTL and to verify them with UMC. The reader can consult the full specification online [23]. In Figure 1, the activity of a Client, a Server thread and a Proxy thread are depicted.

The full specification contains also the definition of the statecharts for the classes Server and Proxy. Objects of these classes are very simple (they have just one state) and their role is simply to forward any incoming request to some available subthread, which will then perform all the relevant activities. The actual complexity of the systems which can be built with these components clearly depends on the number of Clients, Servers and Proxies that one wants

[6] In the future we do intend to consider Clients that perform a loop of SOAP invocations or issue several SOAP invocations before awaiting the deferred SOAP results.

to deploy, on the number of subthreads one assigns to each Server or Proxy and on the maximum number of times a Proxy thread may retry to contact a Server or a Client before it must give up.

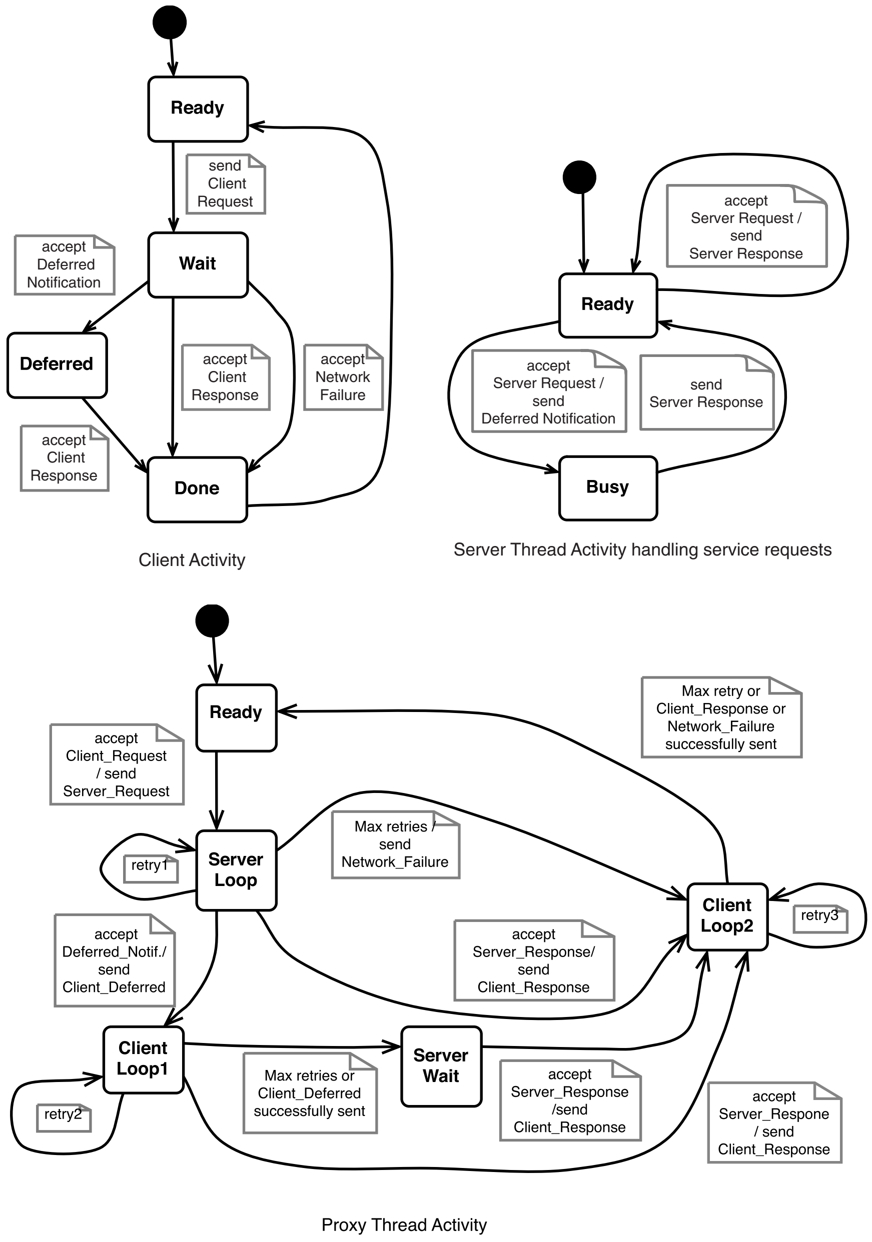
Ready
send Client Request
Wait
accept Deferred Notification
Deferred
accept Client Response
accept Network Failure
accept Client Response
Done
Client Activity
accept Server Request / send Server Response
Ready
accept Server Request / send Deferred Notification
send Server Response
Busy
Server Thread Activity handling service requests
Ready
Max retry or Client_Response or Network_Failure successfully sent
accept Client_Request / send Server_Request
Server Loop
retry1
Max retries / send Network_Failure
Client Loop2
retry3
accept Deferred_Notif./ send Client_Deferred
accept Server_Response/ send Client_Response
Client Loop1
retry2
Max retries or Client_Deferred successfully sent
Server Wait
accept Server_Response /send Client_Response
accept Server_Respone / send Client_Response
Proxy Thread Activity

Fig. 1. Activity of a Client, a Server thread and a Proxy thread

The minimal system composed of 1 Client, 1 Proxy and 1 Server (the latter two both with only 1 subthread) and 1 as maximum number of retries, clearly is an example of a small system with only 118 states and 245 transitions. A more complex system can be deployed by using 2 Clients, 1 Server and 1 Proxy (the latter two with 2 subthreads each), and with up to 2 communication attempts. Such a system contains 96,481 states and 367,172 transitions. Finally, a system composed of 3 Clients, 1 Server and 1 Proxy (the latter two with 3 subthreads each) instead is too complex to be able to explicitly measure its size (far more than 600,000 states and well over 1,000,000 transitions).

5.1 Verification of UCTL Formulae with UMC

In this section, we show the verification with UMC of several behavioural properties expressed in UCTL over our model of the aSOAP protocol. These properties demonstrate the logic's flexibility in dealing with both action- and state-based properties. A different set of behavioural properties is verified in [1]. Property 1:

From every system state a Proxy thread can reach its initial state 'Ready'

can be shown to hold by using UMC to verify the state-based UCTL formula

$$AG\ EF\ \mathit{PT1.state} = \mathit{Ready},$$

in which *PT1* is a Proxy Thread. This is different for Clients, since Property 2:

A Client C1 may reach a deadlock, i.e.
there exists a system state from which C1 cannot evolve

can be shown to hold by using UMC to verify the action-based UCTL formula

$$EF\ (\neg EF < \mathit{C1}\colon > \mathit{true}),$$

in which *C1*: is satisfied by any system evolution in which object *C1* is the one that evolves. This outcome is not so bad as it might seem, because Property 3:

Whenever a Client C1 reaches a deadlock,
then C1 is either in state 'Wait' or in state 'Deferred'

can be shown to hold by using UMC to verify the action/state-based UCTL formula

$$AG\ (\,(\neg EF < \mathit{C1}\colon > \mathit{true}) \Rightarrow (\mathit{C1.state} = \mathit{Wait} \vee \mathit{C1.state} = \mathit{Deferred})\,).$$

The time needed to verify any of the above formulae in the minimal system mentioned above (with 118 states and 245 transitions) is negligible. The situation is different for the system with 2 Clients specified above (with 96,481 states and 367,172 transitions). In spite of its higher complexity, the evaluation of the formula $EF\ (\neg EF < C1: > true)$ (corresponding to Property 2) is still almost immediate, since it requires the analysis of only 1998 states, while the remaining formulae require the analysis of all system states and therefore their evaluation requires about one minute (using a state of the art portable computer).

Finally, none of the above formulae can be verified over more complex systems. However, formulae whose evaluation require the analysis of just a small fraction of all the system states can be evaluated also for such complex systems. Consider, e.g., the formula

$$EF\ EG\ ((PT1.state = ClientLoop2) \wedge < PT1: > (PT1.state = Ready)),$$

which states that there exists an infinite path along which object *PT1* always remains in the *ClientLoop2* state, although the object itself always has the possibility of immediately returning to the state *Ready* in just one step. This formula can be proved to hold also in case of a complex system composed of 3 Clients, 1 Server and 1 Proxy (but each with 3 subthreads), which has more than $600,000$ states and over $1,000,000$ transitions. It takes just a few seconds, during which only $92,536$ system states are analysed. Clearly it is simply a form of unfairness in the scheduling that prevents the Proxy thread to complete its execution cycle.

6 Conclusions

In this paper, we have presented the action/state-based temporal logic UCTL and its on-the-fly model checker UMC. The need to define an action/state-based logic stems from the fact that in order to verify concurrent (software) systems, it is quite often necessary to specify both state information and the evolution in time by actions (events). As a result, semantic models should take both views into account. The L^2TSs that are at the basis of UCTL are one such semantic model. Given a state of an L^2TS, UMC evaluates the validity of a UCTL formula on that state on the fly by analyzing the transitions allowed in that state, and by analyzing the validity of some subformula in only some of the next reachable states, all this in a recursive way. Some clever "archiving" of performed computations allows UMC to evaluate UCTL formulae in linear time.

UML is a graphical modelling language for object-oriented software (systems). UML models can be used to visualize, specify, build and document several aspects—or views—of such systems. The UML semantics associates to each active object a state machine, and the system's behaviour is defined by the possible evolutions of these communicating state machines. All possible system evolutions can be formally represented as an L^2TS in which the states represent the system configurations and the transitions represent the possible evolutions of a system configuration. As a result, UCTL can be used to express properties of the dynamic behaviour of complex systems described as UML state diagrams. The ability to state structural properties of system configurations (state attributes and predicates) and not just actions (events), opens the door to the modelling and verification of structural properties of parallel systems. Examples include topological issues, state invariants and mobility issues.

Acknowledgements. We thank Corrado Moiso of Telecom Italia for having provided us with the case study mentioned in this paper and for his work on aSOAP.

References

1. ter Beek, M.H., Gnesi, S., Mazzanti, F., Moiso, C.: Formal Modelling and Verification of an Asynchronous Extension of SOAP. In: Bernstein, A., Gschwind, T., Zimmermann, W. (eds.) Proceedings of the 4th IEEE European Conference on Web Services (ECOWS 2006), Zurich, Switzerland, pp. 287–296. IEEE Computer Society, Los Alamitos, CA (2006)
2. Bhat, G., Cleaveland, R., Grumberg, O.: Efficient On-the-Fly Model Checking for CTL*. In: Proceedings of the 10th IEEE Symposium on Logics in Computer Science (LICS 1995), San Diego, CA, USA, pp. 388–397. IEEE Computer Society, Los Alamitos, CA (1995)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) ETAPS 1999 and TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
4. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-Based Software Model Checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
5. Chaki, S., Clarke, E.M., Grumberg, O., Ouaknine, J., Sharygina, N., Touili, T., Veith, H.: State/Event Software Verification for Branching-Time Specifications. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 53–69. Springer, Heidelberg (2005)
6. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: Concurrent software verification with states, events, and deadlocks. Formal Aspects of Computing 17(4), 461–483 (2005)
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986)
8. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Transaction on Programming Languages and Systems 16(5), 1512–1542 (1994)
9. Cleaveland, R.: Tableau-Based Model Checking in the Propositional μ-Calculus. Acta Informatica 27(8), 725–747 (1989)
10. De Nicola, R., Vaandrager, F.W.: Actions versus State based Logics for Transition Systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
11. De Nicola, R., Vaandrager, F.W.: Three Logics for Branching Bisimulation. Journal of the ACM 42(2), 458–487 (1995)
12. Fernandez, J.-C., Jard, C., Jéron, T., Viho, C.: Using On-The-Fly Verification Techniques for the Generation of test Suites. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 348–359. Springer, Heidelberg (1996)
13. Gnesi, S., Mazzanti, F.: On the fly model checking of communicating UML State Machines. In: Dosch, W., Lee, R.Y., Wu, C. (eds.) SERA 2004. LNCS, vol. 3647, pp. 331–338. Springer, Heidelberg (2006)
14. Gnesi, S., Mazzanti, F.: A Model Checking Verification Environment for UML Statecharts. In: XLIII Annual Italian Conference AICA, Udine (2005)
15. Hennessy, M., Milner, R.: Algebraic Laws for Nondeterminism and Concurrency. Journal of the ACM 32(1), 137–161 (1985)

16. Huth, M., Jagadeesan, R., Schmidt, D.A.: Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001)
17. Kindler, E., Vesper, T.: ESTL: A Temporal Logic for Events and States. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 365–384. Springer, Heidelberg (1998)
18. Mazzanti, F.: UMC User Guide v3.3. Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR (2006)
19. Müller-Olm, M., Schmidt, D.A., Steffen, B.: Model-Checking—A Tutorial Introduction. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 330–354. Springer, Heidelberg (1999)
20. OMG (Object Management Group), UML (Unified Modeling Language), `http://www.uml.org/`
21. Pnueli, A.: Linear and Branching Structures in the Semantics and Logics of Reactive Systems. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 15–32. Springer, Heidelberg (1985)
22. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading, MA (1998)
23. Specification of aSOAP, `http://fmt.isti.cnr.it/umc/examples/aSOAP.umc`
24. Stirling, C., Walker, D.: Local Model Checking in the Modal μ-Calculus. In: Díaz, J., Orejas, F. (eds.) Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT 1989), Barcelona, Spain, vol. 354, pp. 369–383. Springer, Berlin (1989)
25. Wirsing, M., Clark, A., Gilmore, S., Hölzl, M., Knapp, A., Koch, N., Schroeder, A.: Semantic-Based Development of Service-Oriented Systems. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 24–45. Springer, Heidelberg (2006)
26. W3C (WWW Consortium), Latest SOAP versions, `http://www.w3.org/TR/soap/`