LOVER: Light-weight fOrmal Verification of adaptivE systems at Run time

Amir Molzam Sharifloo[1] and Paola Spoletini[2]

[1]Dipartimento di Elettronica e Informazione, Politecnico di Milano,
P.zza Leonardo da Vinci 32, 20133 Milano, Italy
[2]Università dell'Insubria
via Ravasi, 2, 21100 - Varese (Italy)
molzam@elet.polimi.it, paola.spoletini@uninsubria.it

Abstract. Adaptive systems are able to modify their behaviors to respond to significant changes at run time such as component failures. In many cases, run-time adaptation is simply replacing a piece of system with a new one without interrupting the system operation. In terms of component-based systems, an adaptation may be defined as replacing a system component with a new version at run time. However, updating a system with new components requires the assurance that the new configuration will fully satisfy the expected requirements. Formal verification has been widely used to guarantee that a system specification satisfies a set of properties. However, applying verification techniques at run time for any potential change can be very expensive and sometimes unfeasible. In this paper, we present a methodology, called LOVER, for the lightweight verification of component-based adaptive systems. LOVER provides a new process model supported with formalisms, verification algorithms and tool to verify a significant subset of CTL properties.

1 Introduction

Adaptive systems have been deeply studied over the last decade as means for developing dependable software applications, always more flexible and dynamic [4]. Examples of such systems are service-oriented applications [10], active sensor networks [12] and smart grids [1]. Due to the increasingly use of such systems, a lot of research has been carried out to develop techniques that support adaptation [4, 8]. Run-time adaptation is required if a system is not able to cope with the unpredicted changes occurring at run time. For example, a service hired from an external component fails during the system operation or stops supporting a set of requirements. To react to such failures, a new component, discovered at run time, may be plugged to the system. Similarly, new components with higher level of QoS may become available while the system is operating, which may be preferred to the currently used component. However, since there has been no information about the behavior of the new component at design time, it is necessary to reason about its impact and negative side effects on the overall system behavior at run time.

To avoid any violations, it is necessary to guarantee that the overall system properties will be satisfied in case of applying an adaptation. This could be assured by formally verifying the new system specification, which is obtained by integrating the specification of the new component, against the properties. Intuitively, it is an extra work and overhead due to the fact that the major part of the specification does not change. Moreover, model checking a large specification at run time is quite difficult because of the time and resource limitations. If we could reuse the verification results of the invariant part for future verifications, this would significantly save the time and resource usage at run time.

The existing approaches to verifying adaptive systems mainly focus on the specification and verification of adaptation process [19, 2]. These approaches assume that there is a complete knowledge about the system and environment behavior at design time, so they are able to reason about the properties of the whole interaction model. However, this is not the case in many realistic examples, in which the information about the behavior of some components and the environment are obtained only at run time. This is why run-time verification techniques come into play to monitor and check that the running system does not violate the specification and the properties [11, 14]. Although these approaches are less expensive than model-checking techniques but still they are not complete, and do not guarantee the satisfaction of the properties.

The contribution of this paper is a methodology, called LOVER[1], to efficiently verify that a set of adaptations will lead to the satisfaction of the overall system properties. More specifically, our approach allows the designer to verify the system at design time, even if some components are unspecified. Our model checking algorithm then verifies if the requirements hold and produces a set of constraints for the unspecified components, if needed. Once the components are specified at runtime they can be verified in isolation against this new set of constraints, without checking again the entire system.

Our approach is different from the assume-guarantee approaches in which a set of assumptions on the environment of a component is made that guarantees the satisfaction of the desired properties [13, 6, 9]. Instead we address the run-time model checking of incomplete or changing specifications that comprise dynamic components evolving at run time. We focus on component-based adaptive systems represented by an extension of Labeled Transition Systems (LTS) and verification algorithms for *qualitative* Computational Tree Logic (qCTL), CTL without the next operator. Moreover, we provide a tool support for the verification algorithm, and a formalism to specify the constraints.

The remainder of the paper is organized as follows. Section 2 intuitively motivates the research problem through a running example. Section 3 presents LOVER process model, the formalisms, and the verification algorithms. Experimental results are reported in Section 4. Section 5 discusses the state of the art in verifying adaptive systems, and finally, Section 6 concludes the paper and gives some hints for the future work.

[1] $\underline{\text{L}}$ight-weight f$\underline{\text{o}}$rmal $\underline{\text{v}}$erification of adaptiv$\underline{\text{e}}$ systems at $\underline{\text{r}}$un time

2 The Running Example

In this section, we introduce the running example that is used through the paper. *Secure Information Retrieval* (SIR) is an information system that receives requests, in form of questions, from the clients and responds to them via encrypted messages. The system behavior, in terms of the interactions among the components, is illustrated in Figure 1.

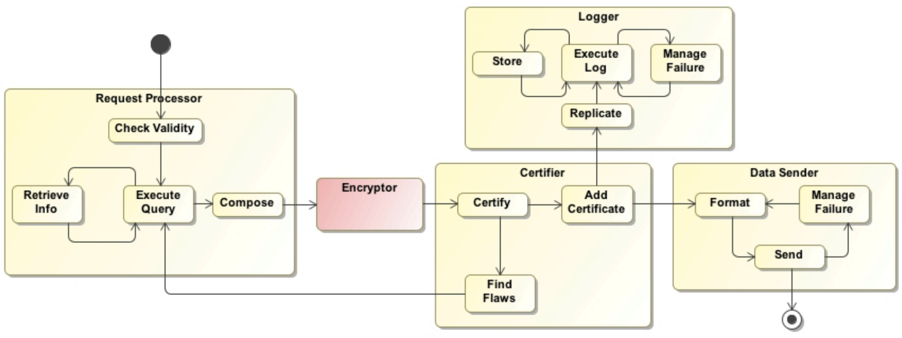

Fig. 1. The activity flow of the Secure Information Retrieval system

A request received from a client is processed by *Request Processor* component. First, the validity of the request is checked and then the requested information is retrieved by querying on different data centers. The results are composed as a message to be sent to the client. This message is encrypted by an *Encryptor* component. The system is designed in such a way that is able to dynamically change the encryption method depending on the level of the requested security and performance. Hence, Encryptor can be rebound to different components at run time with respect to the context. The encrypted message is checked against a set of security standards by a *Certifier* component. The certified message is logged and sent to the client. For security and reliability reasons, the following set of properties shall be guaranteed by the system.

Security property: any message shall be encrypted before being sent out over the network;
Reliability property: the system shall recover from any failure.

Note that the satisfiability of these properties strongly depends on how the encryption is performed and the details of this module are unknown at design time. Indeed, even if an encryption module is selected and the verification is accomplished at design time, this binding may change for different reasons at run time and may require a new verification phase to re-assure the properties. The SIR system is only an example of many component-based systems whose properties depend on dynamic components, which may be bound or changed at run time. Such systems require a continuous verification process that should be as light-weight as possible to avoid intolerable overheads.

3 The LOVER Framework

Differently from traditional model checking approaches, LOVER deals with incomplete models, where a set of components are unspecified at design time and are known only at run time. Obviously, the classical techniques could be applied by checking the system every time the bindings (unspecified at design time) are resolved or changed. Indeed, the time and space required for the verification could be considerable, and since some bindings are resolved only while the system is operating, the total overhead in resolving them should be kept as small as possible.

To overcome these limitations, we propose LOVER, which is a two-phase approach, that allows the designer to verify the incomplete system specification at design time and generates a set of constraints for the unspecified components. Those constraints are verified at run time whenever the component specifications become available. An overall view of LOVER is given in Figure 2. At design time, the incomplete system is described as a particular kind of LTS, where some states are transparent w.r.t. the labels. This model is then checked against a desired qCTL property. The result of the verification could be "yes", "no" or "conditionally yes". The last option gives the set of constraints that has to be satisfied by the unspecified components such that the whole system satisfies the given property. These constraints are expressed in path-qCTL, an extension of qCTL that allows the specification of properties also over finite paths. The constraints are verified by a path-qCTL model checker, which can be obtained by a simple extension to any CTL model checker, such as NuSMV [5].

In this section, we first introduce the novel formalisms and briefly recall qCTL. Then we present the core of LOVER: the model checking algorithm for incomplete models, and a sketch the proof of its equivalence with the traditional solution. We then conclude by showing how to check path-qCTL properties on a component specification expressed as a variation of LTS.

3.1 Incompletely Labeled Transition System

An Incompletely Labeled Transition System (ILTS) is a labelled transition system (LTS) in which the set of states is partitioned in R, the set of *regular* states, and T, the set of *transparent* states, that are special states that can represent more complex components and are considered as unknown. Formally, an ILTS is specified as a tuple $\langle S, s_0, \rightarrow, L\rangle$ over the alphabet A of atomic propositions, where

- S is a set of states, which is partitioned in two sets: R (Regular) and T (Transparent) , i.e., $S = R \cup T$ and $R \cap T = \varnothing$;

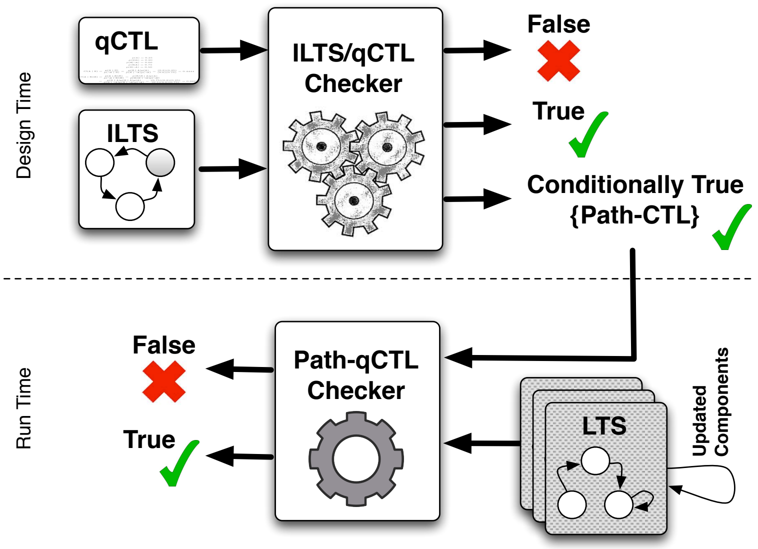

Fig. 2. The LOVER Framework

- s_0 is the initial state;
- $\rightarrow \subseteq S \times S$ represents the transitions between states;
- $L : R \rightarrow \wp(A)$ is the labeling function that associates a subset of atomic propositions to each regular state.

The transparent states represent unknown components that, once specified, can be modeled using a special kind of LTS, namely LTS with single final state, i.e., a tuple $\langle S, s_0, s_F, \rightarrow, L \rangle$, where $s_F \in S$ is the final state. The initial state and the final state represent the unique entering and exiting points in and from the component, respectively. ILTS can be used to model dynamic systems in which some components are unspecified at design time or may change at run time. In other words, there is a big part of system specification that is known at design time, but there are some components that may be left undefined or may be dynamically replaced with other components. Figure 3 shows the ILTS of the motivating example, which is driven from the activity flow, presented in Section 2. Transparent state 5 represents the unavailable specification of Encryptor. The other states are labeled regarding the three message attributes: encrypted, failed, and sent.

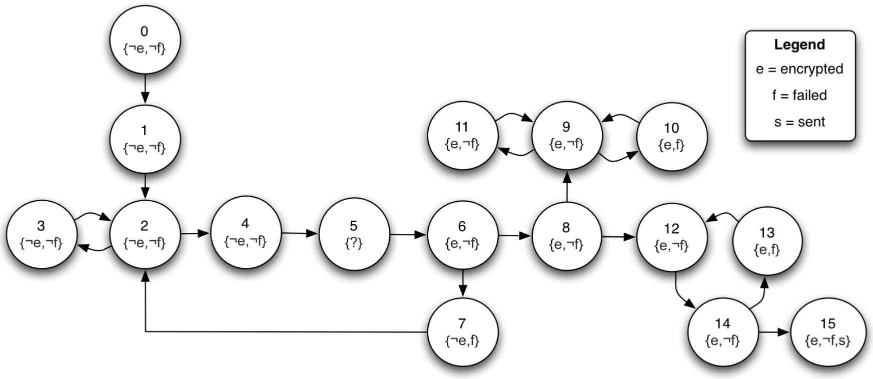

Fig. 3. The ILTS of the Secure Information Retrieval system

3.2 Qualitative CTL and Path-Qualitative CTL

Qualitative CTL (qCTL) is a proper subset of CTL that excludes metrics by neglecting the operators EX and AX. Hence, the syntax of the language becomes:

$$\phi \rightarrow \ \phi \wedge \phi \mid \neg\phi \mid E\ \phi\ U\ \phi \mid E\ G\ \phi \mid p$$

where $p \in AP$, EU and EG are the CTL operators whose semantics is briefly recalled below.

CTL is classically defined on a state of LTS $M = \langle S, s_0, L\rangle$ ($M,\ s \models\ \varphi$ means that φ holds in a state s of the LTS M) as follows:

- $M,\ s \models\ p\ \Leftrightarrow\ p \in L(s)$;
- $M,\ s \models\ \neg\varphi\ \Leftrightarrow\ M,\ s \not\models \varphi$
- $M,\ s \models \varphi_1 \wedge \varphi_2\ \Leftrightarrow\ M,\ s \models \varphi_1$ and $M,\ s \models\ \varphi_2$;
- $M,\ s \models\ E\varphi_1 \cup \varphi_2 \Leftrightarrow$ if there exists a path π starting from s such that $\exists s_k \in \pi \mid M,\ s_k \models \varphi_2$ and $\forall s_i \in \pi$ with $i < k$, $M,\ s_i \models \varphi_1$;
- $M,\ s \models\ EG\ \varphi \Leftrightarrow$ if there exists an infinite path π starting from s such that $\forall s_i \in \pi$, $M,\ s_i \models \varphi$.

Notice that the classical boolean connectives ($\vee$, $\Rightarrow$ and $\Leftrightarrow$) and the temporal operators AU, AG, EF, and AF can be derived from the above sets of operators. As an example, let us consider the security and reliability properties presented in Section 2, using the set of atomic propositions $AP = \{s, e, f\}$, the meaning of which was explained above. The property "All messages are encrypted before being sent out over the network" can be expressed as $A(\neg sUe)$, meaning that there is no sending until the encryption is performed. The reliability property ("The system eventually recovers from any failure") can be instead expressed as $\neg EFEGf$, meaning that there does not exist a path in which eventually there will be a path in which there is a failure forever.

We formally define path-qCTL by adding a temporal operator to qCTL that allows the designer to predicate also on finite sequences of events. Path-qCTL will be used to describe the constraints that has to be guaranteed by the transparent components to assure the requirements validity. The syntax of the language is formally defined as follows:

$$\phi \rightarrow\ \phi \wedge \phi \mid \neg\phi \mid E\ \phi\ U\ \phi \mid E\ G\ \phi \mid E_p\ G\ \phi \mid p$$

where $p \in AP$, EU and EG are the CTL operators (the above set of derivable operators is still derivable)), $E_P G$ is a fresh temporal operator, that indicates that the arguments, on which it is applied, holds at least in a possible scenario starting from the present until the end of the system behavior, i.e, the final state.

We can define the semantics of path-qCTL on M, a labelled transition system with a unique final state s_F, as defined above. If φ is a formula $M,\ s \models\ \phi$ means that ϕ holds in a state s of the LTS M. Omitting the qCTL operators, we just need to define the semantics of E_pG as follows

$M,\ s \models\ E_pG\ \phi \Leftrightarrow$ if there exists a path π, starting from s and ending in the final state s_F of M, such that, for all s_i in π, $M,\ s_i \models \phi$.

3.3 qCTL model checking of incomplete models

The core of LOVER is the qCTL model checking algorithm for incomplete models, described as ILTS. The basic idea is to modify the traditional explicit CTL model checking [3] in order to deal with transparent states. The algorithm takes as inputs a qCTL property and an ILTS. If the ILTS is a regular LTS, it behaves as the traditional approach on regular LTS, while if the ILTS contains transparent states, it computes the set of path-qCTL formulae that shall be guaranteed by the components modeled as transparent states.

More precisely, the algorithm works as follows. First, the qCTL formula is parsed and its parsing tree is derived. As usual, the leaves of the tree are propositions and the inner nodes are boolean and temporal operators. Similarly to CTL model checking, a bottom-up approach is applied to the tree to calculate the satisfactory states for each sub-formula, starting from the leaves of the tree. For each node of the tree, the set of the states in which the sub-formula holds is calculated by applying Algorithm 1.

Algorithm 1 is invoked for every subtree of the parsing tree, starting from the leaves. The algorithm takes as inputs a subtree T of the parsing tree (possibly the parsing tree itself), the formula φ, and the ILTS M on which the original formula is evaluated. The tree T is a binary tree, where a node representing a unary operator has a single son, while a node representing a binary operator has two sons. We use $T.S$ to refer to the set of states in M that satisfy the formula represented by the current subtree, $T.left$ and $T.right$ to refer to the left and the right subtrees of the current tree (when the root is a binary operator), and $T.son$ to refer to the subtree of the current tree (when the root is a unary operator). The

Algorithm 1 Node evaluation

```
**evaluate**($\varphi, T, M$){
 $X = \varnothing$
 **switch** ($\varphi$){
   **case** $\varphi \in AP$ :
     **for all** $s \in M.S$ { $constr(\varphi, s) = \varnothing;$ }
     **for all** $s \in M.S$ {
        **if** ($s \in M.R$ && $p \in L(s)$) {
          $X = X \cup \{s\};$
       }**elseif**($s \in M.T$){
           $X = X \cup \{s\};$
           $constr(\varphi, s) = constr(\varphi, s) \cup \{(\Theta p, s)\};$ }}
   **case** $\varphi = \neg\varphi_1$ :
     **for all** $s \in M.R - T.son.R${
        $X = X \cup \{s\};$ }
     **for all** $s \in (T.son.S \cap M.T) \vee (s \in T.son.R \wedge constr(\varphi_1, s) \neq \varnothing)${
        $X = X \cup \{s\};$
        $constr(\varphi, s) = buildNeg(constr(\varphi_1, s));$ }
 **case** $\varphi = \varphi_1 \wedge \varphi_2$ :
     **for all** $s_1 \in T.left.S${
        **for all** $s_2 \in T.right.S${
           **if** ($s_1 = s_2$){
            $X = X \cup \{s_1\};$
            **if**($constr(\varphi_1, s_1) \neq \varnothing \vee constr(\varphi_2, s_1) \neq \varnothing$){
              $constr(\varphi, s) = ANDCombine(constr(\varphi_1, s_1), constr(\varphi_2, s_1));$ }}}}
   **case** $\varphi = E\varphi_1 U \varphi_2$ :
     **for all** $s_2 \in T.right.S${
        $X = X \cup s_2$
        **if**($s_2 \in T.right.S$){$constr(\varphi, s_2) = resolveRightUntil(\varphi_2, s_2)$}
        $X' = \varnothing;$
        **while**($X'! = X$){
          $X' = X;$
          **for all** $s_1 \in T.left.S${
            **if**($\exists s' \in X | (s_1, s') \in M.Transitions$)
              $X = X \cup \{s_1\}$
              $\pi = buildPath(s_1, T.right.S)$
              {$constr(\varphi, s_1) = resolveLeftIUntil(constr(\varphi_1, s_1), \pi);$ }}}}
   **case** $\varphi = EG\varphi_1$ :
     $S' = \varnothing;$
     **for all** $s \in M.T${
        $S' = S' \cup \{\{s\}\}; X = X \cup \{s\};$
        $constr(\varphi, s) = resolveOutSCC(constr(\varphi_1, s);$ }
     **for all** $sub_S \in \wp(T.son.S)${
        **if**($sub_S$ is a scc){
          $S' = S' \cup \{sub_S\}; X = X \cup sub_S;$
          **for all** $s \in sub_S${
            $constr(\varphi, s) = resolveInSCC(constr(\varphi_1, s), subS);$ }}}
     **for all** $sub \in S' \cup M.T${
        $X' = sub$
          $X'' = \varnothing;$
          **while**($X''! = X'$){
            $X'' = X';$
            **for all** $s_1 \in T.son.S${
              **if**($\exists s' \in X' | (s_1, s') \in M.Transitions$)
                $X' = X' \cup \{s_1\}$
                $\pi = buildPath(s_1, T.right.S)$
                $constr(\varphi, s_1) = resolvePathGlobally(constr(\varphi_1, s_1), \pi);$ }}
        $X = X \cup X';$ }
 }
$T.S = X;$
}
```

elements of the ILTS M are referred as $M.S$ (states), $M.R$ (regular states), $M.T$ (transparent states), $M.Transitions$ (transition relation), and $M.L$ (labeling function).

The algorithm uses the set X (initialized in line 2) as a local set to store the elements that satisfy φ. Moreover, the set of constraints that are needed to satisfy the formula φ in a transparent state s are saved in a matrix $constr$. Each element $constr(\varphi, s)$ is a set of constraints in the form $[(\psi_1, state_1), \ldots, (\psi_n, state_n)]$, meaning that the formula φ holds in s if the path-qCTL formula ψ_1 holds in $state_1$, ..., and the path-qCTL formula ψ_n holds in $state_n$. For example, $constr(EGa, s) = \{[(EGa, s)], [(E_pGa, s), (EGa, s')]\}$ means that the formula EGa holds in the transparent state s either if the formula itself holds in the correspondent component or if the formula E_pGa holds in the correspondent component and EGa holds in the component represented by the transparent state s'. Roughly speaking, the elements of the set are conjunctions and the set is seen as a disjunction of such conjunctions. The evaluation algorithm is based on a switch on the value of the most external operator in φ (line 3). Considering the grammar of qCTL, there are five different cases: atomic proposition (lines 4–11), negated formulae (lines 12–17), conjunctions (lines18–24), EU formulae (lines 25–36), and EG formulae (lines 37–58).

If φ is an atomic proposition and T is a leaf, the value of $constr(\varphi, s)$ is initialized for all s. Note that this is the only case in which $constr(\varphi, s)$ is based on the value of the sub-formulae. Then, all the regular states labeled with φ are added to the set of states X in which the formula holds (lines 7-8). Moreover all the transparent states are added to X (line 10), together with an update of the correspondent $constr$ slot. In particular, for each transparent state s, the constraint Θp is added to $constr(\varphi, s)$(line 11). The symbol Θ represents a still non-identified path-qCTL operator, of which the kind will be resolved in the rest of the algorithm. The operator Θ indicates that a propositional formula, that is apparently evaluated on a state, will be evaluated on a component. If the propositional formula is inside a temporal formula, Θ will be resolved by the semantics of the outer operators.

If T is a subtree of which the root is a $\neg$ operator, i.e., φ is a formula of the form $\neg\varphi_1$, all the regular states that are not in the set of states in which φ_1 holds are added to the set X of states in which φ holds (line 13-14). The transparent states are always added to the set of states in which a formula holds together with a set of constraints (that however could also be unsatisfiable). Thus, every transparent state s is added to X. Moreover, the regular states in which the formula φ_1 conditionally holds are added to X. For both these kinds of states, the correspondent slot $constr(\varphi, s)$ is updated through the function $buildNeg(constr(\varphi_1, s))$ (lines 15-17). This function basically considers the "negation" of the set of constraints for φ_1 in s. At this stage, $\neg\Theta p$ is changed to $\Theta\neg p$, since the constraint comes from an untimed sub-formula. Note that the set represents a disjunction of constraints, while each element in square bracket represents a conjunction of constraints and this has to be considered in negating the set. For example the negation of $constr(EGa, s)$ considered above is $\{[(\neg EGa, s), (\neg E_pGa, s)], [(\neg EGa, s), (\neg EGa, s')]\}$.

When φ is a formula of the form $\varphi_1 \wedge \varphi_2$ and T is a subtree of which the root is a $\wedge$ operator, all the states that are both in the set of states in which φ_1 and φ_2 hold are added to the set X of states in which φ holds (line 19-23). If the added state contains a constraint w.r.t. the considered subformula, the correspondent constraint is built using the function $ANDCombine$ $(constr(\varphi_1, s_1)$, $constr(\varphi_2, s_1))$ (lines 23-24). This function basically considers the "conjunction" of the two sets, by simplifying the elements on the same state in the same constraint. At this stage, the conjunction of the elements Θp and $\Theta p'$ is considered as $\Theta(p \wedge p')$, because both the constraints come from an untimed formula. For example, if $\varphi = EGa \wedge EaUb$, $constr(EGa, s)$ is defined as shown above and $constr(EaUb, s) = \{[(EaUb, s)], [(E_pGa, s), (EaUb, s')]\}$, then $constr(EGa \wedge EaUb, s)$ becomes $\{[(EaUb, s), (EGa, s)], [(E_pGa, s), (EGa, s), (EaUb, s')], [(EaUb, s), (E_pGa, s), (EGa, s')], [(E_pGa, s), (EaUb, s'), (EGa, s')]\}$

If T is a subtree of which the root is an EU operator and φ is a formula of the form $E\varphi_1 U \varphi_2$, the procedure is in two steps. First, all the states that are in the set of states in which φ_2 holds (T.right.S) are added to the set X of states in which φ holds. (line 26-27). If the added state s is transparent, the constraint of s for φ is updated using the function $resolveRightUntil(\varphi_2, s)$. This function transforms the elements of the form (x, s) that appears in $constr(\varphi_2, s)$ into $(E\varphi_1 Ux, s)$. Note that the algorithm only changes the constraints connected to the current states and not the others on adjacent states of a constrained sequence. At this stage, if x has the form Θp or contains a Θ, the operator Θ is deleted. Second, X is updated by using φ_1 (lines 29-36). More precisely, we update X by adding in it the states, in which φ_1 holds (condition in line 32) and from which it is possible to reach a state in X (condition in line 33). The idea is that φ_1 holds in such states (these states can be either regular or transparent) and from them it is possible to reach directly the states in X, i.e., the states in which φ holds. For each added state, the path π that connects it to a state in the set in which φ_2 holds is computed (line 35). The path π is used to enrich the set of constraints that make φ hold in it. For this purpose, the algorithm uses the function $resolveLeftIUntil(constr(\varphi_1, s), \pi)$. This function adds to $constr(\varphi, s)$ a constraint composed by the conjunction of all the constraints x that makes φ_1 true in the transparent states of π (except the last one), after updating them in E_pGx. Again, if the original constraints contain Θ, the operator Θ is deleted.

Finally, if T is a subtree of which the root is an EG operator, i.e., φ is a formula of the form $EG\varphi_1$, all the transparent states are added to the set X of the states in which φ holds. Moreover, these states are added as singleton to the set S' that contains all the sets that represent strongly connected components, in which φ_1 always holds. Since, the added states are transparent, the correspondent set of constraints is updated using the function $resolveOutSCC(constr(\varphi_1, s))$ (lines 39-41). This function adds the constraint $EG\varphi_1$ to each of these states. Then as in the classical explicit model checking algorithm, for all the non-elementary possible subset in which φ_1 holds, if the subset is a strongly connected component, the set of the subset is added to S' and the states to X. If there exist transparent states in the added subset, their constraints are updated with the function $resolveInSCC(constr(\varphi_1, s), subS)$ (lines42-46). This function, for all the states in the subsets, adds a conjunction that includes for each state the constraint E_pGx, where x is the constraint that makes φ_1 hold in that state. Obviously, if the components only contain regular states, this constraint is empty. As the last step, analogously to what is done for operator EU, X is updated by using φ_1 and S' (lines 47-57). More precisely, starting from each strongly connected components in S', the set of the states in which φ_1 (condition in line 53) holds and from which it is possible to reach a state in which φ holds (condition in line 54) is added to X. Once a transparent node is added, the path π that connects it to the strongly connected component in which φ_1 holds is computed (line 56), and using π (that contains also the considered strongly connected component), the set of constraints that makes φ hold in it, is updated using $resolvePathGlobally(constr(\varphi_1, s_1), \pi)$. This function works analogously to function $resolveLeftIUntil(constr(\varphi_1, s), \pi)$. In all the functions considered for this case, the operator Θp is automatically deleted.

After the evaluation algorithm is performed on the whole parsing tree from the leaves to the root, if the set of the states, that satisfy the root, contains the initial state of M, then the property φ holds constrained to $const(\varphi, s_0)$. If there is still an unresolved Θ in this set of constraints, it means that the initial state is a transparent state and that the property φ is untimed. In this case the untimed property that follows Θ has to hold in the initial state of the component representing the transparent state.

Sketching the correctness of qCTL algorithm for incomplete models Here we informally describe the correctness of our algorithm by showing the equivalence between the classical checking of qCTL and the two-stage checking performed by LOVER. Our "proof" technique is based on the semantics of qCTL and path-qCTL. Basically, we show that checking a qCTL property φ on an ILTS with Algorithm 1 and imposing the obtained path-qCTL formulae to the components that are bound to the transparent states in the ILTS is equivalent to check the same property φ with the traditional qCTL algorithm on an LTS, obtained by substituting the transparent states in the original ILTS with the components bound to them.

Consider an LTS M and an ILTS M', obtained by removing k independent LTSs M_i^T (with $1 \leq i \leq k$) - starting from s_0^i with final state s_F^i - from M and replacing each of them with a transparent state s_i^T. An example, with $k = 2$, is shown in Figure 4, where the LTS M_1^T and M_2^T in M are abstracted through s_1^T and s_2^T in M'. A path π of M is called *compatible* with a path π' of M' if and only if π contains exactly the same (and in the same order) regular states of π' and, instead of the transparent states of π', it contains one of the possible paths that cross the graph obtained by substituting the transparent states with the actual components.

We want to show that proving a qCTL formula φ on M is equivalent to proving φ on M' using the LOVER approach.

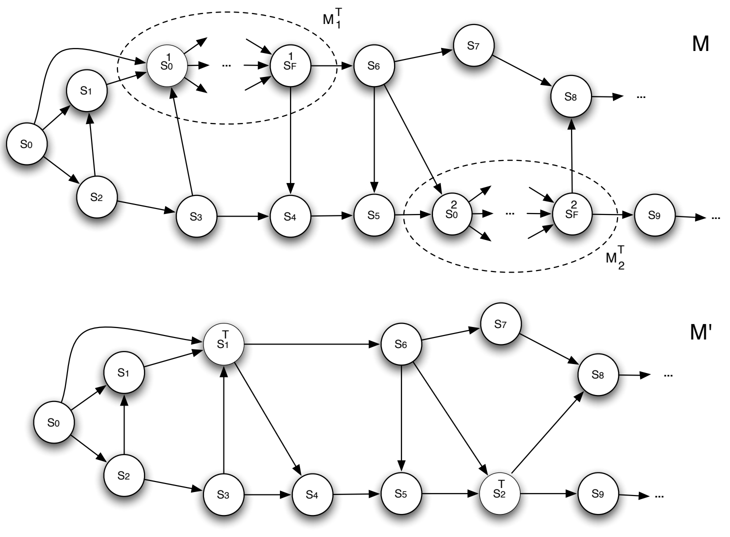

Fig. 4. An example of LTL and its corresponding ILT.

Let us start by considering formulae of the form $E\varphi_1 U \varphi_2$. Checking the validity of this formula corresponds to check if $M,\ s_0 \models\ E\varphi_1 U \varphi_2$ holds, i.e., if there exists a path π starting from the initial state s_0 such that $\exists s_k \in \pi \mid M,\ s_k \models \varphi_2$ and $\forall s_i \in \pi$ with $i < k$, $M,\ s_j \models \varphi_1$. To show the correctness of Algorithm 1 is enough to show that, given a generic path π' in M', it satisfies $E\varphi_1 U \varphi_2$ and the components corresponding to the transparent states in M' satisfy the constraints obtained by LOVER if and only if there exists a path π of M, compatible with π', that satisfies $E\varphi_1 U \varphi_2$.

A generic path π' in M' can be as follows:

1. π' does not contain any transparent state s_i^T;
2. the last state of π' is a transparent state;
3. π' contains transparent states, but the last state is not transparent;
4. π' contains transparent states, including the last position.

Obviously, case (4) is a generalization of cases (2) and (3), but since they are more intuitive, we will treat them separately (even if the proof for these cases are included in the proof for case (4)).

The first case is naive. Since there is no transparent state, Algorithm 1 behaves exactly as the classical model checking. The second case corresponds to π' containing only a transparent state at the end. Our algorithm will produce

"yes" only if for all s_x in π' (excluded the last $s_{|\pi'|}$) $M', s_x \models \varphi_1$, exactly as required by the classical model checking algorithm. Moreover our algorithm will impose that $E\varphi_1 U \varphi_2$ holds in the component corresponding to $s_{|\pi'|}$ and this will happen only if exists a path π in M compatible with π' that satisfies $E\varphi_1 \cup \varphi_2$. The third case considers a path π' that contains a number of transparent states, but not at the end. Our algorithm will produce "yes" only if for all non-transient state s_x in π' (excluded the last $s_{|\pi'|}$) $M', s_x \models \varphi_1$, and $M', s_{|\pi'|} \models \varphi_2$. Moreover our algorithm will impose that $E_p G\varphi_1$ holds in the component corresponding to the transparent state of π'. All these requirements are satisfied if there exists a path π in M compatible with π' that satisfies $E\varphi_1 U \varphi_2$.
The last case is the most general case and corresponds to π' containing a number of transparent states, including the end. Our algorithm on such a path would first label the state with φ, using only φ_2. Among all the possible constraints that the labeling imposes, for the proof, we are only interested to the sets that include all the states through the end of π'[2]. So, if $\pi' = s_0, s_1, ..., s_n$ and the sequence of transparent states in it is $[s'_1, \ldots, s'_m]$, for all $0 \leq i \leq n-1$, the set $constr(\varphi_2, s_i)$ can contain the constraint $[(sub_{\varphi_2}, s'_j)$, $(sub_{\varphi_2}, s'_{j+1})$, $\ldots$, $(sub_{\varphi_2}, s'_{m-1})$, $(\varphi_2, s'_m)]$, where s'_j is the first transparent state after s_i in π' and sub_{φ_2} is a subcondition needed to make φ_2 true in the current state. Moreover in s_n, $constr(\varphi_2, s_n)$ contains the constraint $[(\varphi_2, s_n)]$, where s_n is exactly the last transparent state s'_m. When our algorithm starts the labeling using also φ_1, each of the above constraints can be used to compute $constr(E\varphi_1 U \varphi_2, s_0)$, adding constraints of the form $[(E_p G\varphi_1, s'_1)$, $\ldots$, $(E_p G\varphi_1, s'_{j-1})$, $(E(\varphi_1 U sub_{\varphi_2}), s'_j)$, $(sub_{\varphi_2}, s'_{j+1})$, $\ldots$, $(sub_{\varphi_2}, s'_{m-1})$, $(\varphi_2, s'_m)]$. Moreover, if such a constraint exists, the algorithm checks that all the regular states before the j-th transparent state satisfy φ_1 and all the regular states after the j-th transparent state satisfy sub_{φ_2}. A compatible path π satisfies $E\varphi_1 \cup \varphi_2$ if and only if it satisfies one of the previous constraints.

An analogous reasoning can be applied to $EG\ \varphi$, while the atomic proposition case and the boolean connectors need to be treated differently. When $\varphi \in AP$, φ holds in M if s_0 is labeled with φ. If s_0 is a regular state, then our algorithm will check exactly the same. If instead s_0 is included in an LTS substituted with a transparent state, the algorithm will come up with the constraint that in this component, that is exactly the same condition checked by the classical algorithm. Moreover, our algorithm deals with the boolean connectors as the classical one, only modifying the previously obtained constraints according to the connector semantics.

Notice that this is not a formal proof, but is an informal reasoning to show the equivalence of the two approaches.

3.4 Path-CTL model checking

To verify a path-qCTL property on an ILTS with unique final state, we need to observe that ILTS with a unique final state is a particular case of LTS and that

[2] We are looking at the satisfiability using the whole path; all the subpaths are considered separately as one of the possible four mentioned scenarios.

the final state does not influence the verification of classical qCTL properties. Hence, since path-qCTL is qCTL with the extra temporal operator E_pG, we can readapt the classical CTL algorithm to deal with this new operator. Algorithm 2 shows a fragment of an evaluation function to deal with formulae φ of the form $E_pG\varphi_1$. The fragment uses the same notation and structure of Algorithm 1. The idea is that, starting from an LTS with final state M, the algorithm builds M' by delating the states where φ_1 does not hold and the transitions as a consequence (line 3). Then, a state s of M' is added to the set of states in which φ holds (line 7) if the final state s_F belongs to M' (line 4) and there exists at least a path from s to s_F in M' (line 6). This check can be done easily with a breadth-first search in $O(|M'.S|)$, making the overall evaluation $O(|M'.S|^2)$.

Algorithm 2 Checking formulae of the form $E_pG\varphi$

```
1: case($\varphi = E_pG\varphi_1$) :
2:   $S' = \{s \in M.S | s \in T.son.S\}$;
3:   $M' = M|_{S \leftarrow S'}$;
4:   if $s_F \in S'$ {
5:     for all $s \in S'$ {
6:       if $SearchPaths(s, s_F, M')$ {
7:         $X = X \cup \{s\}$; }}}
```

4 Experimental Results

In this section, we present the applicability and scalability of the proposed approach in practice.

4.1 Tool Support and Applicability

We have developed a prototype tool to verify ILTSs against properties expressed in qCTL according to the algorithm presented in Section 3. The inputs of the tool are two files, which contain the ILTS and the qCTL property. The tool is capable to verify the property and report the output as a set of solutions[3]. Solutions are path-CTL properties that constrain the transparent states. The tool is also able to verify LTSs against properties expressed in path-CTL.

To demonstrate the applicability of our approach, we used the tool to verify the ILTS of the running example against the properties (presented in Section 2). Regarding the global security property $A(\neg s \cup e)$, the model checker returns two solutions that constrain a possible specification of the transparent state (state 5). The first solution is $\{S_5 \models A(\neg s \cup e)\}$, which means that the global property shall hold also in the specification. The second solution is $\{S_5 \models A_pG\neg s\}$.

[3] The tool is available online: `https://sites.google.com/site/amirsharifloo/tool-lover`

This property enforces the paths between the start and the end states of the specification to be labeled with $\neg s$.

Applying the verification algorithm to the second property returns only one solution: $\{S_5 \models \neg EFEGf\}$. Therefore, any component that is bound at run time to play the role of Encryptor shall satisfy this path-qCTL property.

4.2 Scalability

To see how our approach scales up with respect to the number of regular and transparent states, we performed a scalability experiment. To do so, we generated different models by concatenating the running example. Concatenation here means to produce a new ILTS by simply connecting the last state (state 15) of the ILTS to the first state of another copy of the ILTS. For example, the first concatenation results in an ILTS with 30 states in which two states are transparent. This way we generated larger models and applied the tool to verify the properties.

Figure 5 illustrates the result, which is obtained by running the experiment 100 times and computing the average. The result shows that the verification time of both properties exponentially grow. However, the verification time of the nested property grows faster as the number of states increases. The machine we used for the experiments had the following characteristics: OS = Mac, CPU=2.4 GHz Core 2 Duo, and RAM=4 GB.

Although in general the verification cost of the algorithm exponentially grows with respect to the number of transparent states, the specification topology is a key parameter that can significantly affect the total amount of the computation. Note that this is the verification time required at design-time. Obviously, it is more than a simple verification performed by an LTS model checker, since the algorithm calculates constraints, considering the combinations. Moreover, our tool is a prototype and the result can be improved by applying further optimizations. Despite such overhead at design-time, the verification cost of verifying unspecified components at run time is always less than the model-checking of the whole specification, and that is the main advantage of applying LOVER in practice. This is due to the fact that the verification at run-time phase is performed on the specifications of the components, which are much smaller than the entire one. Moreover, the constraints can be checked in parallel in order to speed up the verification.

5 Related Work

There have been a set of approaches to formally specify adaptive systems and apply model checking techniques to verify their properties at design time [16]. To formally specify adaptive behaviors, Zhang et al. [19, 20] introduce A-LTL (an extension of LTL). A-LTL adds an operator Adapt-operator which eases describing the properties that hold in the initial program and the adapted program. They also present a modular verification algorithm to verify an adaptive

Transparent	State	$A(\neg s \cup e)$	$\neg E \Diamond E \Box f$
1	15	0.105637	0.079811
4	60	0.76972	0.702177
7	105	3.156306	5.841801
10	150	8.659444	24.611509
13	195	19.197839	70.304578
16	240	36.051059	161.264
19	285	59.829017	326.778

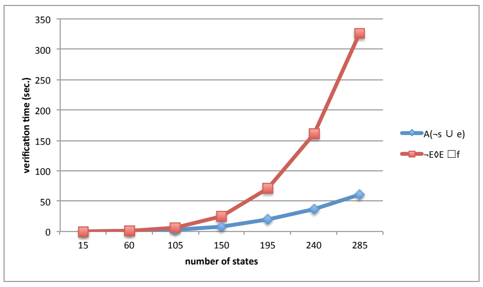

Fig. 5. The verification time for the properties (The table provides the precise values shown in the diagram.)

system against the formulae expressed in A-LTL [21]. The system is represented as a state machine in which the states present the system configurations and transitions are adaptation actions.

Adler et al. [2] propose an approach to modularly design and model adaptive embedded systems such that the system specification is suitable for verification analysis. The approach distinguishes between the part of the system that supports the functionality and the part that manages the adaptation, and focuses on specifying the adaptation behavior in order to verify the stability property of the adaptation process. Theorem proving techniques e.g. Isabelle/HOL are employed to verify the properties. The approach is extended in [15] to verify system properties with respect to environment constraints. To this end, the interaction among the system and the environment is modeled and is verified that the system properties are guaranteed assuming a maximal environment. This approach assumes that all the environmental behaviors can be predetermined in advanced so the verification of the properties are performed at design time. Although applying modular techniques reduces the verification costs, the approach assumes that the whole knowledge on the specification and the adaptations is available at design time.

Păsăreanu et al. [6, 9] propose an approach to automatically generating assumptions for the environments of a component, and apply the technique for compositional verification. The output of the approach describes the environments in which a component will satisfy the expected properties. Our approach is different in the point that there exists a couple of unspecified components that make the specification incomplete and the verification unfeasible. What we do is to enforce those components with some constraints such that the global properties hold.

To verify the properties of dynamic component-based systems, there has been a trend of research based on black-box testing and monitoring at run time [18, 17, 7]. Xie and Zhe [18, 17] propose a test-based approach for the verification of component-based systems, in which the behavior of some components is not specified. The system consists of a host system and a collection of unspecified components, which are represented as finite transition systems that synchronously communicate via a set of input/output symbols. An algorithm is used to derive a set of strings that unspecified components are supposed to generate through black-box testing. Although testing approaches do not lead to state explosion, applying them at run time is still challenging.

Run-time verification [11, 14] an interesting area that addresses a problem similar to what we deal with in this paper. Runtime verification approaches assume that the implementation may be different from the specification, or the environment may change in such a way that the expected system properties violate. The aim of run-time verification is to ensure that the traces generated by the system satisfy the properties. To this end, the key idea is to generate specific elements, called *monitor*, to check the compliance at run time. Differently from model checking, run-time verification does not lead to state exploration, but it does not guarantee that the properties certainly hold.

6 Conclusion and future work

This paper presents a two-phase framework to efficiently verify adaptive systems, in which some components may dynamically change at run time. To support the framework, we developed formalisms, verification algorithms, and a prototype tool. We applied our approach to a running example, and evaluated the scalability by larger models.

This paper states the initial steps that we have taken to address the run-time model checking of dynamic systems. There are many directions to extend this work. At the moment, we are working to optimize the implementation and to explore a new symbolic approach. In the current paper, we have focused on qualitative CTL, but the future work is to support the full CTL by adding Next operator. Further steps are applying the approach to other case studies in different areas and extending the framework to support other temporal logics such as LTL.

Acknowledgments

We would like to thank Carlo Ghezzi for the fruitful discussions on this work. This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

1. Smart Grids European Technology Platform. `http://www.smartgrids.eu/`.
2. Rasmus Adler, Ina Schaefer, Tobias Schuele, and Eric Vecchié. From model-based design to formal verification of adaptive embedded systems. ICFEM'07, pages 76–95, Berlin, Heidelberg, 2007. Springer-Verlag.

3. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
4. Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*. Springer, 2009.
5. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
6. Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 331–346, 2003.
7. Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems. In *Proceedings of the 9th international conference on Software engineering and formal methods*, SEFM'11, pages 204–220, 2011.
8. Carlo Ghezzi. Engineering evolving and self-adaptive systems: An overview. In *Software and Systems Safety - Specification and Verification*, pages 88–102. 2011.
9. Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, 2002.
10. N. Gold, A. Mohan, C. Knight, and M. Munro. Understanding service-oriented software. *Software, IEEE*, 21(2):71 – 77, march-april 2004.
11. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.
12. Philip Levis, David Gay, and David Culler. Active Sensor Networks. In *Proc. of the 2nd Symposium on Networked Systems Design & Implementation - Volume 2*, pages 343–356. USENIX Association, 2005.
13. Corina S. Pasareanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software: A comparative case study. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 168–183, 1999.
14. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, April 2005.
15. Ina Schaefer and Arnd Poetzsch-Heffter. Model-based verification of adaptive embedded systems under environment constraints. *SIGBED*, 6(3):9:1–9:4, 2009.
16. Klaus Schneider, Tobias Schuele, and Mario Trapp. Verifying the adaptation behavior of embedded systems. SEAMS '06, pages 16–22, 2006.
17. Gaoyan Xie, , and Zhe Dang. Ctl model-checking for systems with unspecified finite state components. SAVCBS, 2004.
18. Gaoyan Xie and Zhe Dang. An automata-theoretic approach for model-checking systems with unspecified components. FATES, 2004.
19. Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.
20. Ji Zhang and Betty H.C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361 – 1369, 2006.
21. Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. AOSD '09, pages 161–172, New York, NY, USA, 2009. ACM.