Model-based Runtime Verification Framework for Self-optimizing Systems

Y. Zhao  S. Oberthur  M. Kardos  F.J. Rammig

Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany

Abstract
This paper describes a novel on-line model checking approach offered as service of a real-time operating system (RTOS). The verification system is intended especially for self-optimizing component-based real-time systems where self-optimization is performed by dynamically exchanging components. The verification is performed at the level of (RT-UML) models. The properties to be checked are expressed by RT-OCL terms where the underlying temporal logic is restricted to either time-annotated ACTL or LTL formulae. The on-line model checking runs interleaved with the execution of the component to be checked in a pipelined manner. The technique applied is based on on-the-fly model checking. More specifically for ACTL formulae this means on-the-fly solution of the NHORNSAT problem while in the case of LTL the emptiness checking method is applied.

Keywords: Model-based Runtime Verification, On-the-fly ACTL/LTL Model Checking, Self-optimizing System.

1 Motivation

Mechatronic systems represent a special class of complex cross-domain embedded systems. The design of such systems involves a combination of design techniques and technologies used in the mechanical and electrical engineering as well as in computer science. The increasing complexity, even emphasized by the system heterogeneity, is one of the major problems in today’s mechatronic industry (e.g., automotive industry). To deal with this complexity an approach

1 This work is developed in the course of the Collaborative Research Center 614 - Self-Optimizing Concepts and Structures in Mechanical Engineering - Paderborn University, and is published on its behalf and funded by the Deutsche Forschungsgemeinschaft (DFG).
is to build mechatronic systems in a self-reflecting, self-adapting and self-optimizing way. In the Collaborative Research Center 614 “Self-optimizing concepts and structures in mechanical engineering” we are researching such an approach. The main focus is put on self-optimizing applications with highly dynamic software components which are optimized and even replaced at runtime. Moreover, the considered applications run under real-time constraints (see Fig. 1). As failures of these technical systems usually have severe consequences, safety and predictability is of paramount importance. This puts new demands on verification of such complex and highly dependable systems.

For real-time systems with a dynamic task set, acceptance tests with respect to schedulability are the state of the art. In reconfigurable and dependable systems the safety and consistency after component replacement has to be checked as well. This extends the classical area of on-line acceptance testing. Traditionally in real-time systems one tries to execute as many checking activities as possible off-line. In systems of dynamic structure this would mean that all components that may be used in a substitution have to be checked (e.g., using conventional model checking) to be correct in an arbitrary context, i.e., in the most general context. Of course this very general correctness requirement would result in highly over-dimensioned and thus inefficient components. This would be a contradiction to the overall objective of self-optimization. Therefore we decided to develop a novel technique for on-line model checking context-specific parts of components at runtime. Consequently we offer this verification as a service of the underlying RTOS. The real-time restrictions make it necessary to perform the on-line model checking at the level of (UML) models, to assume that the models are implemented correctly, and to assume that any non context specific internals of components have been verified off-line.

Fig. 1. Case study
To deal with this problem, we present a model-based runtime verification mechanism, by which the safety and the consistency of the on-line reconfiguration of such a system can be ensured to a greater extent. Let’s take a typical example to show how the model-based runtime verification mechanism is applied to self-optimizing systems (Fig. 1). Suppose a real-time application that contains four components A, B, C and D running in parallel. Now, due to some reason, say, the change of environment or what else, a substitution request is detected by the RTOS at time point $t_r$ that the component C will be replaced by component E at the $t_d$th time step after $t_r$. Before the replacement is really done at time point $t_r + t_d$, the RTOS will request the model-based runtime verification service to check if the system still keeps safety and consistency after the replacement. According to the response from the verification service, Yes, No or Unknown, the RTOS will accept or reject the requirement to substitute the component E for the component C.

Obviously, the substitution of the component E for the component C will cause the environment of each component in the system to be changed at runtime directly (i.e., B, D and E) or indirectly (i.e., A). Recall the compositional reasoning [15] for such component-based systems that each component is checked correct under the given assumption to the environment on which the component depends. As in our case study, the environment of each component in the system might be changed dynamically due to the runtime reconfiguration. The following question arises: “Does the changed environment still satisfy the required assumption?” To answer this question, the traditional model checking is unfortunately not suitable any more: on the one hand, it is difficult to predict how and when the reconfiguration will happen; on the other hand, it is difficult to check the safety and consistency of the reconfiguration within the given timing constraint. In practice, it is unrealistic to check off-line all the possible cases of the reconfigurations due to the huge time and space complexity. To our knowledge, the state of the art runtime verification (Section 4) is also not suitable for our needs: on the one hand, only linear temporal logic formulae as well as assertions and invariants can be checked by tracing the program execution; on the other hand, potential errors can be detected only when they have really happened. Due to the above reasons, we present our model-based runtime verification mechanism as system service of an RTOS, by which we are able to on-line check the safety and consistency for self-optimizing systems at model level so that the potential errors can be predicted and thus avoided in time before they really happen.

The paper is organized as follows: Section 2 introduces the basic concepts; Section 3 details our model-based runtime verification framework; Section 4 discusses the related work; finally, Section 5 ends with the conclusion.
2 Preliminaries

2.1 Real-time UML Statechart

According to the design technique [14] presented within the Collaborative Research Center 614, the self-optimizing systems are designed with the CASE tool Fujaba\(^2\) based on the modeling concepts of UML 2.0. That is, the architecture of a system is specified by a component diagram together with the definitions for ports and connectors; the overall behavior of the system is specified by UML state machines with real-time extension, called real-time UML statecharts, associated to each component. In fact, the whole behavior of a component \(C\) is the parallel composition of the real-time UML statecharts \(M^r_i\) \((1 \leq i \leq m)\), which are the refinements of the corresponding protocol state machines associated to the ports \(P_i\) \((1 \leq i \leq m)\) of \(C\), and the internal synchronization statechart \(M^s\) of \(C\), i.e., \(M_C = M^r_1 \parallel M^r_2 \parallel \cdots \parallel M^r_m \parallel M^s\). It is easy to reason that the overall behavior of the system model is the parallel composition of a set of real-time UML statecharts.

As far as real-time UML statecharts are concerned, there are many different variants to extend the usual UML statechart with timing constraints in the literature. Here we introduce the real-time UML statechart presented in [13]. Simply speaking, a real-time UML statechart is obtained by adding real-time annotations to the usual UML statechart (with some simplifications). That is, except for the timing constraints, initial state, history state, simple state, composite state (sequential state and concurrent state) and final state as well as transition in the real-time UML statechart have similar meanings as the corresponding counterparts in the usual UML statechart. In addition, like in timed automata, transitions in a real-time UML statechart may be urgent (non-urgent) and have a priority (integer). Note that a non-urgent transition is identical to a transition with priority 0. In a sense, real-time UML statechart can be seen as a variant of the hierarchical timed automaton.

Without loss of generality, Fig. 2 illustrates a typical part of a real-time UML statechart. The state \(S_1\) has the time invariant \(t_0 \leq 5\) (time units) and \(S_2\) has the time invariant \(t_0 \leq 20\) and \(t_1 \leq 13\), where \(t_0\) and \(t_1\) are global clocks. The entry action entry\(S_1()\) of \(S_1\) has the worst case execution time (wcet) \(w = 1\) (time unit) and the clock \(t_0\) is reset while entering \(S_1\), the do activity do\(S_1()\) of \(S_1\) has \(w = 2\) together with period \(p \in [2, 3]\) and the exit action exit\(S_1()\) of \(S_1\) has \(w = 1\). Similarly, the clocks \(t_0\) and \(t_1\) are reset while exiting \(S_2\). The transition from \(S_1\) to \(S_2\) is triggered whenever the event \(e\) is available and the guard \(x \leq 2\) and the time guard \(1 \leq t_0\) are held. In the

\(^2\) [URL: http://wwwcs.upb.de/cs/fujaba/](http://wwwcs.upb.de/cs/fujaba/)
mean time, the clock $t_2$ is reset and the action with $w = 2$ is executed. The firing of the transition has to be finished within the time interval $[1, 10]$ and whenever the clock $t_1 \in [3, 6]$. By default, the transition is urgent and has the priority 1.

Fig. 2. Part of real-time UML statechart

2.2 Real-time OCL

Real-time OCL (RT-OCL) [12] is a state-oriented temporal extension to the usual Object Constraint Language by introducing additional bounded temporal logic operators over the sequence of active state configurations of real-time UML statecharts. E.g., the following invariant requires that for each instance of the class C, at each time point of the next 20 time units, on all possible execution paths, the states $S_1$ and $S_2$ must be subsequently entered:

$$
\text{context } C
\text{ inv:}
\text{ self} @ post[1, 20] \rightarrow \forall \forall (p : OclPath \mid p \rightarrow \text{includes}(\text{Sequence}\{S_1, S_2\}))
$$

The introduced notations are compliant with the syntax of the OCL 2.0 Proposal and are mapped to real-time CTL (RTCTL) [11], a discrete time variant of the Computation Tree Logic, for further application to model checking. In the future, we’ll further extend the real-time OCL defined in [12] to cover timed linear temporal logic.

2.3 Abstract State Machine Language

The Abstract State Machine Language (AsmL) [17,18] is an executable specification language built upon the theory of Abstract State Machines (ASMs) [4,16], a formal method for high-level modeling and specification that has proven its strong modeling and specification abilities in various application domains\(^3\). The main strength of AsmL resides in its rich and expressive syntax, formally underpinned by the ASM theory, which gives user the ability

\(^3\) http://www.eecs.umich.edu/gasm/
to create precise and comprehensible specifications at any desired level of abstraction. Among other things, AsmL provides a powerful type system that facilitates a wide scale of designs ranging from pure mathematical specifications of algorithms to the complex object-oriented software specifications. The type system incorporates basic primitive types, object-oriented structures such as classes, structures, interfaces and mathematical structures like tuples, sets, bags, sequences and maps. Furthermore, it also supports type generalization and so-called constraint types. Besides these language features, the AsmL comes with a tool support that allows usual validation via specification execution as well as enhanced model-based testing. Moreover, the AsmL tool suite provides a functionality to drive the exploration of the model state space. This feature can be used for constructing a corresponding Kripke structure from the given specification that can further serve as basis for the application of model checking algorithms [21].

3 Model-based Runtime Verification Framework

3.1 Overview

Note that our runtime verification service (Fig. 3) is working at model level. That is, both the models and the properties to be checked must be known ahead of time. To do this, first of all, the real-time UML statecharts of the modeled system and the related real-time OCL constraints are exported from the Fujaba Tool Suite in form of XML documents and translated into the corresponding AsmL models and real-time ACTL/LTL formulae respectively at the Translation phase. Then, the Kripke structure of each AsmL model is derived by applying the exploration functionality to the AsmL model and the ACTL/LTL formulae to be verified are transformed into Büchi automata. Finally, the resulting Kripke structures and Büchi automata are stored into a repository in advance. Whenever a verification request from an RTOS is received (Fig. 1), the verification service will fetch the related Kripke structures and Büchi automata from the repository and then quickly go to trigger on-the-fly model checking.

The self-optimizing operation may cause the system to be reconfigured at runtime in many ways. This paper mainly concerns such a case that one component is replaced with another one. Obviously, the replacement may change the environment of every active component in the system directly or indirectly. On the other hand, the only constraint on the components replaceable with each other is that they must follow the compatible protocols, which means the protocol of the new one must be the same as or the refinement of the old one. Therefore, it is quite necessary to invoke the runtime verification service
to make sure that such a reconfiguration is really safe and consistent.

For component-based systems, compositional reasoning [15] is an efficient solution to the state space explosion problem inherent to model checking [6]. That is, each component is verified correct under the given assumptions to the environment of the component. According to our design technique [14], the properties required to each component are specified in real-time OCL at the design phase by developers. We translate them into assertions, invariants, ACTL or LTL formulae with real-time annotations at the translation phase of our verification framework. Note that the assertions (invariants) here denote such trivial properties that the propositions should be held at some time and place (at any time and place). The time-annotated ACTL formulae are just RTCTL (Real-time Computation Tree Logic) [11] with only universal quantifiers allowed. The time-annotated LTL is defined in a similar way. That is, a time interval of the form \([a, b]\), where \(a\) and \(b\) are \(\text{Integers}\) and \(a \leq b\), is attached to the usual temporal operators, thus, named bounded temporal operators. E.g., the formula \(AG(p \rightarrow AF_{[0, t]}q)\) specifies that \(p\) always leads to \(q\) within \(t\) time steps. However, to avoid the fairness conditions caused by the \(\text{eventuality}\) operators, we require that the \(\text{eventuality}\) operators must be bounded ones if any. In this way, the bounds on the \(\text{eventuality}\) operators prevents indefinite postponement. The real-time ACTL (LTL) formulae are just an intermediate representation for the properties to be checked. These formulae are finally transformed into \(\text{Büchi}\) automata which are denoted as
unit delay state transition graphs. In fact, the *Kripke* structures derived from AsmL models are also denoted as unit delay state transition graphs.

Of course, the implementation of each component in the system must conform to the corresponding model of the component. This is automatically achieved by using Fujaba to generate code directly from the design model. Therefore, the implementation of a component is the refinement of the model of the component or, put it another way, the model is the abstraction of the corresponding implementation. This means that an ACTL (LTL) formula being *true* at the model level implies that it is also *true* at the implementation level, while it being *false* at the model level does not imply that it is also *false* at the implementation level. That is, our runtime verification is conservative due to being applied to model level. However, the benefit of predicting and avoiding errors is gained just due to its being applied to model level.

### 3.2 Pipelined Working Principle

It is easy to see that the timing constraint is the main barrier for our model-based runtime verification. To leap over this barrier, we adopt a pipelining technique to gain more execution time for verification. The sequence diagram Fig. 4 illustrates the cooperation between the verification service and the real-time application. More precisely, the pipelined working mode is done between the RTOS and the verification service and thus transparent to the application.

Whenever the RTOS receives a component substitution request from the application, it will invoke the verification service to check if the substitution is legal or not. The answer must be given within the required timing constraint, say $t_d$, in our example. If lucky, the verification may finish the checking task before the timing constraint is over. Unfortunately, it might be not the case for more complex systems. Therefore, it is quite possible that, within $t_d$ time units, only the next $t_1$ time steps starting from the initial states are checked *Yes*, which means the substitution is safe up to the coming $t_1$ time steps. In this case, the RTOS does allow the application to make the substitution and execute forward $t_1$ time steps. During this period, the verification continues to check, say the next $t_2 - t_1$ time steps. Accordingly, the application can then go ahead the next $t_2 - t_1$ time steps. Note that at each time point $t_d + t_i$ ($i \geq 1$) (with respect to $t_r$), the application can report its current state, say $s_i$, to the verification. Based on this runtime information, the verification can locate in the system model the corresponding state with respect to $s_i$ and thus avoid checking the whole state space of the system model by only checking a sufficient sub-space reachable from this specific state mapped from $s_i$. In this way, the computation load of the verification can be reduced to a greater extent.
The above process is repeated. If at some time point an error is detected, then the verification can be terminated with the answer \textit{No} to the RTOS. Otherwise we still continue this process. At some time point, say $t_d + t_j$ (relative to $t_r$), although the checking result is positive, however, if the time interval $t_{j+1} - t_j$ is not more than the pre-defined time constant $t_c$, which denotes the minimum time steps that the verification must keep ahead of the application, then we have to give up the verification process and report \textit{Unknown} to the RTOS. Note that these two cases only mean that the errors might happen in the future, because we do not know if the errors are spurious or not. To avoid that the errors really happen, we have to conservatively choose to reject the substitution request. That is, an exception will be raised by the RTOS together with a counter example if necessary. As to how the application will respond to the exception, it is beyond the scope of the paper. Finally, if a sufficient sub-space that covers this actual run of the real-time application is successfully checked, then we can report definitely \textit{Yes} to the RTOS and terminate the verification process. From now on, the application can guarantee to execute safely and consistently after the substitution.

Of course, in order to make applicable our model-based runtime verification technique, some preconditions are required to hold:

1) During the design phase the components under the given assumptions on the environments they depend on should be checked \textit{correct};

2) The implementation of each component conforms to the corresponding
design model;
3) The properties as well as assertions and invariants to be checked are limited to time-annotated ACTL and LTL formulae;
4) The processing speed for verification is faster than that for application.

Note that we implicitly assume that the components under consideration own finite state machines. In fact, Fig. 4 just illustrates an ideal pipelined cooperation between the application and the verification via the RTOS as intermediary without considering any implementation detail.

3.3 Model Checking Methodology

It is easy to see that the pipelined working principle between the verification service and the real-time application requires that model checking must be done on-the-fly in a top-down way. Fig. 5 intuitively demonstrates how the on-the-fly ACTL/LTL model checking works in our model-based runtime verification framework, where \( \Diamond \) stands for \( \preceq \) (simulation relation) for ACTL model checking and \( |=| \) (satisfaction relation) for LTL model checking. As mentioned in Section 3.2, from initial states, only the next \( t_1 \) time steps may be checked \textit{Yes} within the given \( t_d \) time units. Similarly, within the next \( t_1 \) time units, the next \( t_2 - t_1 \) time steps may be checked \textit{Yes}. This procedure is repeated until a definite answer \textit{Yes}, \textit{No}, or \textit{Unknown} is concluded. Note that when the on-the-fly model checking runs to the \( (t_d + t_i) \)’th time step (relative to \( t_r \)), it will be informed that the current state of the application is \( s_i \). Therefore, model checking can locate the corresponding state with respect to \( s_i \) in the system model. For simplicity, we also use \( s_i \) to denote its counterpart in the system model. From now on, model checking can continue from this \( s_i \) in the system model. In this way, only a part of the state space of the system model needs to be traversed.

Without loss of generality, suppose a real-time system model \( M \) contains \( n \) components \( C_1, C_2, \ldots, C_n (n \geq 2) \) working in parallel and is requested at time point \( t_r \) to replace the component \( C_k \) (\( 1 \leq k \leq n \)) with another component \( C'_k \) at time point \( t_d \) relative to \( t_r \), denoted as \( M' = M(C'_k/C_k)@t_r \). Due to the non-determinism inherent in concurrent and reactive systems, instead of getting a definite state, we can only deduce all possible states that could be reached by a component \( C_i \) (\( i \neq k \)) at time point \( t_r + t_d \). That is, the starting states of \( C_i \) at time point \( t_r + t_d \) is not unique, i.e., \( M' \) may have more than one initial state, when the runtime verification is triggered. For convenience, we introduce a dummy state \( p_0 \) as the unique initial state of \( M' \). Thus, let \( M' = (AP_{M'}, S_{M'}, R_{M'}, p_0, L_{M'}) \), where \( AP \) denotes a set of atomic propositions, \( S \) a set of states, \( R \subseteq S \times S \) the transition relation and \( L : S \rightarrow 2^{AP} \) the labeling
function over states. In practice, the new system model is constructed on-the-fly, i.e., the parallel composition of $C_1, C_2, \ldots, C_k, \ldots, C_n$ at time point $t_r + t_d$ is guided by the property automaton to be checked. In this way, only a small portion of the overall state space of $M'$ might be constructed before a counterexample is found (if any).

Since runtime checking assertions and invariants, which consist only of propositions, is trivial, in what follows we focus mainly on runtime checking time-annotated ACTL and LTL formulae with *eventuality* operators being bounded ones if any.

### 3.3.1 On-the-fly ACTL Model Checking

Let $B_i$ be the *Büchi* automaton derived from the ACTL formula associated to the component $C_i$ ($1 \leq i \leq n$) and $B'_k$ the *Büchi* automaton of the component $C'_k$. Obviously, the original system $M$ holds $B = B_1 \land B_2 \land \cdots \land B_n$. After the substitution, the new system $M'$ should hold $B' = B_1 \land B_2 \land \cdots \land B'_k \land \cdots \land B_n$. Therefore, the goal of our on-the-fly ACTL model checking is to check if the reconfigured system model $M(C'_k/C_k)@((t_r \triangleright t_d) \preceq B_1 \land B_2 \land \cdots \land B'_k \land \cdots \land B_n)$. Note that the property automata $B_1, B_2, \ldots, B'_k, \ldots, B_n$ are all constructed and stored in a repository in advance. Hence, we can build $B'$ by composing $B_1, B_2, \ldots, B'_k, \ldots, B_n$ on-the-fly. Similar to $M'$, $B'$ may also have more than one initial state. For convenience, we introduce a dummy state $q_0$ as the unique initial state of $B'$ and thus let $B' = (AP_{B'}, S_{B'}, R_{B'}, q_0, L_{B'})$.

To make available on-the-fly ACTL model checking, we go to check the

---

**Fig. 5.** On-the-fly ACTL/LTL model checking methodology
simulation preorder between $M'$ and $B'$ incrementally [23]. In doing so, the decision problem of checking simulation preorder is converted into the satis-
fiability problem for weakly negative Horn formulae [22], called NHORNSAT problem. The basic idea is to encode the properties of the simulation relation between $M'$ and $B'$ into a type of CNF (Conjunctive Normal Form) formula $\Gamma$, i.e., weakly negative Horn formula, and then prove on-the-fly that the CNF formula $\Gamma$ is satisfiable in polynomial time.

Let $X_{p,q}$ be a variable in $\Gamma$, where $p$ and $q$ are states in $M'$ and $B'$ respectively. Then, the clauses in the formula $\Gamma$ are of the following three types:

1) Positive literal $X_{p,q}$, when $(p, q)$ to be in the simulation relation;
2) Negative literal $\overline{X_{p,q}}$, when $(p, q)$ cannot be in any simulation relation;
3) Implication clause of the form $X_{p,q} \rightarrow \bigvee_{p'q'} X_{p'q'}$, when for $(p, q)$ to be in the simulation relation, one of the $(p', q')$'s must be also in the simulation relation. Here $(p', q')$ belongs to the successors of $(p, q)$.

It is easy to reason that, starting from the initial states of $M'$ and $B'$, we can construct the CNF formula $\Gamma$ by adding to $\Gamma$ the proper clauses derived from the reachable pairs of states in $M' \times B'$ layer by layer in BFS (Breadth First Search) order.

Fortunately, an efficient on-the-fly algorithm is presented in [2], which receives one Horn clause at a time and allows fast queries about the satisfiability of the whole formula so far received. Let $l$ be the size of the inserted clause and $n$ the size of the whole formula so far received. Then, the algorithm inserts a clause of size $l$ in $O(l)$ amortized time, propagates the effect of this insertion operation on the previous result in $O(n)$ and decides the satisfiability of the formula heretofore constructed in $O(1)$. This outperforms by an order of magnitude the best known algorithms for the same problem in [8] and [20]. Similarly, a dualization of the algorithm in [2] also gives an efficient linear time on-the-fly solution to the NHORNSAT problem [23]. However, to make this on-the-fly algorithm cooperate seamlessly with a real-time application via RTOS as intermediary in a pipelined way, we need to make some extensions.

Given states $p$, $p'$ and $q$, where $p, p' \in S_{M'}$, $(p, p') \in R_{M'}$ and $q \in S_{B'}$. According to the definition of simulation relation, if $(p, q)$ is in the simulation relation, then, for any transition $(p, p') \in R_{M'}$, there must exist a transition $(q, q') \in R_{B'}$ such that $(p', q')$ is also in the simulation relation; otherwise, $(p, q)$ can not be in any simulation relation. The function $getClauses(p, p', q)$ below returns a set of clauses (literals) derived from $p$, $p'$ and $q$.

\[
getClauses(p, p', q) = \begin{cases} \\
\text{if exists } q' \in \text{nextStates}(q) \text{ where isConsistent}(p', q') \text{ then} \\
\text{return } \{\overline{X_{p,q}} \bigvee_{q' \in \text{nextStates}(q)} X_{p'q'} | \text{isConsistent}(p', q')\}\end{cases}
\]
else
    return \{X_{p,q}\}
endif

In order to keep track of the positive literals in a clause newly created by 
\texttt{getClauses}(p, p', q), the following function \texttt{getLiterals}(p, p', q) returns a set of such literals. Here we introduce \texttt{LiteralTable} to store the literals derived from p, p' and q in form of the tuple \((p, p', q), \text{literals}\) to avoid the repeated computations over p, p' and q if any.

\[\text{getLiterals}(p, p', q) = def\]
\[
    \text{if exists } q' \in \text{nextStates}(q) \text{ where isConsistent}(p', q') \text{ then}
    \]
\[
    \text{let } \text{literals } = \{X_{p,q'} \mid q' \in \text{nextStates}(q) \land \text{isConsistent}(p', q')\}
    \]
\[
    \text{add } ((p, p', q), \text{literals}) \text{ to } \text{LiteralTable}
    \]
\[
    \text{return } \text{literals}
    \]
\[
    \text{else}
    \]
\[
    \text{add } ((p, p', q), \emptyset) \text{ to } \text{LiteralTable}
    \]
\[
    \text{return } \emptyset
    \]
\[
\text{endif}
\]

Algorithm 1 demonstrates how the runtime ACTL model checking works. The \textit{input} parameters of the algorithm are the model \(M'\) of the reconfigured system, the property automaton \(B'\), the timing constraint \(t_d\) and the current state \(s_i\) of the reconfigured system. The \textit{output} of the algorithm is an \texttt{Integer} representing how many time steps the real-time system could go ahead safely. In particular, we define \texttt{Yes}, \texttt{No} and \texttt{Unknown} as \texttt{ constants}, say, \texttt{Yes} = 1, \texttt{No} = 0 and \texttt{Unknown} = \(-1\). The main data structures in this algorithm is listed and explained as follows:

- \texttt{timer} measures the time elapsed so far;
- \texttt{oldTimes} keeps the time steps obtained by the last checking period;
- \texttt{newTimes} keeps the time steps obtained by the present checking period;
- \texttt{currentLiterals} keeps a set of literals to be processed;
- \texttt{oldLiterals} keeps a set of literals already processed;
- \texttt{newLiterals} keeps a set of literals newly created;
- \texttt{oldClauses} keeps a set of clauses already checked;
- \texttt{newClauses} keeps a set of clauses newly created;
- \texttt{ComputedTable} keeps a list of the tuple \((s_i, \text{timeSteps}, \text{currentLiterals})\);
- \texttt{LiteralTable} keeps a list of the tuple \((p, p', q), \text{literals}\).

Notice that \texttt{ComputedTable} and \texttt{LiteralTable} are exploited to improve
the efficiency of the algorithm. The tuple \((s_i, \text{timeSteps}, \text{currentLiterals})\) in \(\text{ComputedTable}\) means that starting from \(s_i\), the next \text{timeSteps} has already be checked \text{safe} and, thus, whenever \(s_i\) becomes the current state again, the program can start checking directly from the \text{currentLiterals}, which is \text{timeSteps} far away from \(s_i\). Initially, \(\text{ComputedTable}\) contains only the tuple \(((p_0, q_0), 0, \{X_{p_0,q_0}\})\). The tuple \(((p, p', q), \text{literals})\) in \(\text{LiteralTable}\) means that the literals derived from \(p, p'\) and \(q\) have already been generated and, thus, whenever \(p, p'\) and \(q\) are traversed again, the program can refer directly to the previously calculated result. Initially, \(\text{LiteralTable}\) is empty.

Once receiving a verification request from the RTOS, the algorithm begins to work from the initial state \((p_0, q_0)\), i.e., the first current state \(s_0 = (p_0, q_0)\). Let all the state \((p', q')\) reachable from \((p, q)\) at the \(t_i\)’th time step be of the \(t_i\)’th layer state with respect to \((p, q)\). In this sense, \(\text{currentLiterals}\) keeps all the literals derived from the \text{timeSteps}’th (i.e., current) layer states relative to \(s_i\) (line 6-11). Consequently, \(\text{newLiterals}\) keeps all the literals derived from the next layer states and \(\text{newClauses}\) all the clauses newly created accordingly (line 15-24). If \(\text{oldClauses}\) together with \(\text{newClauses}\) is still satisfiable, we increment \(\text{newTimes}\) and then update \(\text{oldLiterals}\) and \(\text{oldClauses}\); otherwise, return No to the RTOS (line 30-40). This process is repeated within the \text{while}-loop until the timing constraint is reached (line 12) or all the states reachable from \(s_i\) are checked (line 25). In the former case, \(t_d\) is assigned to a new timing constraint available in the next round of checking (line 42). However, if this new \(t_d\) is not more than the predefined time constant \(t_c\), the algorithm has to give up the checking task and report \text{Unknown} to the RTOS (line 43-44); otherwise, the algorithm updates the \(\text{ComputedTable}\), informs the RTOS that the application can go ahead safely within the next \(t_d\) time steps and then continues the next round of checking (line 46-49). In the later case, the algorithm just tells the RTOS \text{Yes}, because all the states reachable from the current state (i.e., a sufficient subgraph starting from the current state) have already been checked (line 25-26). That is, from now on, the real-time application can run safely forever.

\textbf{Algorithm 1} \textit{On-the-fly ACTL Model Checking Algorithm}

\begin{verbatim}
input: \(M', B', t_d, s_i\)
output: Integer
begin
1  initialization()
2  NextRound :
3    timer := 0
4    oldTimes := newTimes
5    newTimes := 0
\end{verbatim}
if inComputedTable(s_i) then
    currentLiterals := getCurrentLiterals(ComputedTable(s_i))
else
    currentLiterals := \{X_{p,q} | X_{p,q} ∈ oldLiterals \land p = s_i\}
    add (s_i, 0, currentLiterals) to ComputedTable
endif
while timer ≤ t_d do
    newLiterals := ∅
    newClauses := ∅
    for all X_{p,q} ∈ currentLiterals do
        for all p' ∈ nextState(p) do
            if inLiteralTable(p, p', q) then
                newLiterals := newLiterals ∪ literals(LiteralTable(p, p', q))
            else
                newLiterals := newLiterals ∪ getLiterals(p, p', q)
                newClauses := newClauses ∪ getClauses(p, p', q)
            endif
        endfor
    endfor
    if all X_{p,q} ∈ newLiterals holds that X_{p,q} is marked with s_i then
        return Yes
    else
        mark all X_{p,q} ∈ newLiterals with s_i
    endif
    if newClauses = ∅ then
        newTimes ++
        currentLiterals := newLiterals
    elseif isSatisfiable(oldClauses, newClauses) then
        newTimes ++
        currentLiterals := newLiterals
        oldLiterals := oldLiterals ∪ newLiterals
        oldClauses := oldClauses ∪ newClauses
    else
        return No
    endif
endwhile

\[ t_d := newTimes + getTimeSteps(ComputedTable(s_i)) - oldTimes \]
if t_d ≤ t_c then
    return Unknown
else
let timeSteps = newTimes + getTimeSteps(ComputedTable(s_i))
add (s_i, timeSteps, currentLiterals) to ComputedTable
output t_d
goto NextRound
endif
end

Obviously, this algorithm guarantees to terminate if the checking result is negative (i.e., No or Unknown). According to the theorem and corollary below, we can also conclude that this algorithm guarantees to terminate if the checking result is positive. Hence, this algorithm will terminate in all cases.

**Theorem 3.1** Given a system model $M = (AP_M, S_M, R_M, s_0, L_M)$ and an infinite run $\pi = (s_0, s_1, s_2, \cdots, s_i, \cdots)$ of the system model. Then, the following proposition holds:

$$M(s_0) \supseteq M(s_1) \supseteq M(s_2) \supseteq \cdots \supseteq M(s_i) \supseteq \cdots$$

where $M(s_i)$ denotes the subgraph of $M$ starting from $s_i$.

The proof is simple and thus omitted here. From this theorem, we can derive easily a corollary as follows:

**Corollary 3.2** If there is a loop from $s_i$ to $s_j$ in $M$, then $M(s_i) = M(s_j)$.

Recall that the $ComputedTable$ stores a sequence of current states and the corresponding checking results with respect to these states. Given a current state $s_i$, if a literal derived from a state reachable from $s_i$ is checked, then this literal will be marked with $s_i$ (line 28). Obviously, if all the literals derived from the states of $M(s_i)$ are marked with $s_i$, it means that the whole subgraph $M(s_i)$ is checked. Because the subgraphs of $M$ starting from the subsequent current states are all covered by $M(s_i)$, therefore, the algorithm can be finished with the checking result $Yes$ (line 25-26).

In the algorithm, a clause is added to oldClauses only once (line 17-22). Hence, the function $isSatisfiable(oldClauses, newClauses)$ is invoked only when there is non-empty newClauses that will be added to oldClauses (line 33). From the time that all the clauses are checked, the algorithm will no longer need to check the satisfiability problem and just traverse over the graph generated from $M'$ and $B'$ layer by layer until some termination condition is held. Every time a newClauses is added to oldClauses, the time complexity of $isSatisfiable(oldClauses, newClauses)$, as mentioned before, depends on the time of inserting newClauses into oldClauses and the time of propagating the effect of this insertion operation on the previous results. Note that whenever a literal $X_{p,q}$ is evaluated to false, which means that $(p, q)$ can not be in any simulation relation, the literals derived from the states reachable from $(p, q)$
will no longer be processed by isSatisfiable(oldClauses, newClauses). Even though, in the worst case, the number of the literals created is \(O(|S_{M'}| \cdot |S_{B'}|)\) and the number of the clauses generated is \(O(|R_{M'}| \cdot |R_{B'}|)\), by introducing the current state \(s_i\), in every checking period, the algorithm only checks a subgraph starting from \(s_i\). Therefore, in a local view, the time complexity of every round is acceptable with respect to the timing constraints. Thus, this runtime ACTL model checking is feasible in practice.

### 3.3.2 On-the-fly LTL Model Checking

As for on-the-fly LTL model checking, we follow the emptiness checking technique [7]. In doing so, for each component \(C_i (C'_k)\), the property automaton \(B_i (B'_k)\) is derived from the negation of the LTL formula to be checked. Therefore, the goal of our on-the-fly LTL model checking is to check if the reconfigured system model \(M(C'_k/C_k)@(t_r \triangleright t_d) \models B_i (B'_k)\), we can conclude that the component replacement is not safe. By introducing a dummy initial state \(q_0\) and then linking \(q_0\) to the initial states of \(B_i (i \neq k)\) and \(B'_k\), it is easy to combine \(B_1, B_2, \ldots, B'_k, \ldots, B_n\) together into one unified automaton \(B'\) denoted as \(B' = (AP_{B'}, S_{B'}, R_{B'}, q_0, L_{B'})\).

Thus, the emptiness of the intersection of \(M'\) and \(B'\) can be checked on-the-fly. That is, the states of the intersection of \(M'\) and \(B'\) are computed layer by layer in BFS order starting from the initial state \((p_0, q_0)\) on demand. Of course, we still need to do something to make this emptiness checking fit into the pipelined working manner. Algorithm 2 shows how the runtime LTL model checking works. The input and output of the algorithm are the same as those of algorithm 1. Most data structures of the algorithm are also the same as those of algorithm 1, except that we introduce currentStates, oldStates and newStates with functions similar to those of currentLiterals, oldLiterals and newLiterals in algorithm 1.

Once receiving a verification request from the RTOS, the algorithm begins to work from the current state \(s_0 = (p_0, q_0)\). Here ComputedTable initially contains the only tuple \(((p_0, q_0), 0, \{(p_0, q_0)\})\). currentStates keeps the current layer of states to be processed. If a next layer state derived from currentStates was created before, which means a loop is found, then, the algorithm can finish the checking task and return No to the RTOS (line 17-18); otherwise, the next layer state is added to newStates (line 20). In case that no new next layer state can be derived from currentStates, i.e., newStates = \(\emptyset\), which means that the intersection of \(M'\) and \(B'\) is empty, then, the algorithm can finish the checking task and inform Yes to the RTOS (line 24-25). Otherwise, the algorithm increments newTimes and updates currentStates and oldStates.
(line 27-29). This process is repeated within the while-loop until the timing constraint is reached (line 12). The rest of the algorithm is the same as that of algorithm 1.

Algorithm 2 On-the-fly LTL Model Checking Algorithm

```
input: M', B', t_d, s_i
output: Integer
begin
1 initialization()
2 NextRound:
3 timer := 0
4 oldTimes := newTimes
5 newTimes := 0
6 if inComputedTable(s_i) then
7   currentStates := getCurrentStates(ComputedTable(s_i))
8 else
9   currentStates := {(p, q) | (p, q) ∈ oldStates ∧ p = s_i}
10  add (s_i, 0, currentStates) to ComputedTable
11 endif
12 while timer ≤ t_d do
13  newStates := Ø
14  for all (p, q) ∈ currentStates do
15    for all (p', q') holds that p' ∈ nextStates(p) ∧ q' ∈ nextStates(q) ∧
16      isConsistent(p', q') do
17      if (p', q') ∈ oldStates then
18        return No
19      else
20        newStates := newStates ∪ {(p', q')}
21      endif
22  endfor
23 endfor
24 if newStates = Ø then
25  return Yes
26 else
27  newTimes + +
28  currentStates := newStates
29  oldStates := oldStates ∪ newStates
30 endif
31 endwhile
32 t_d := newTimes + getTimeSteps(ComputedTable(s_i)) − oldTimes
33 if t_d ≤ t_c then
```
return Unknown

else
  let timeSteps = newTimes + getTimeSteps(ComputedTable(s_i))
  add (s_i, timeSteps, currentStates) to ComputedTable
  output t_d
  goto NextRound
endif

Obviously, this algorithm guarantees to terminate. In particular, the time complexity of the algorithm, i.e., \(O(|S_M'| \cdot |S_B'|)\), does not increase compared to the static emptiness checking. On the contrary, by introducing the current states from the application, only a subgraph of \(M'\) is processed in every checking period. Therefore, in a local view, the time complexity of every round is acceptable with respect to the timing constraints. Consequently, this runtime LTL model checking is feasible in practice as well.

4 Related Work

The main characteristic of the state of the art runtime verification is to monitor the actual runs of the program under test and then check if the traces derived from these specific runs conform to the required specification. Typically, [3] presents runtime checking for the behavioral equivalence between a component implementation and its interface specification by writing the interface specification in the executable AsmL so that one can synchronously run the interface specification and the component implementation while monitor if they are equivalent on the observed behaviors; [1] presents runtime certified computation whereby an algorithm not only produces a result for a given input, but also proves that the result is correct with respect to the given input by deductive reasoning; [24] presents runtime checking for the conformance between a concurrent implementation of a data structure and a high-level executable specification with atomic operations by first instrumenting the implementation code to extract the execution information into a log and then executing a verification thread concurrently with the implementation while using the logged information to check if the execution conforms to the high-level specification; [5] presents monitoring-oriented programming (Mop) as a light-weight formal method to check conformance of implementation to specification at runtime by first inserting specifications as annotations at various user selected places in programs and then translating the annotations into the efficient monitoring codes in the same target language as the implementation during a pre-compilation stage. Similar to Mop, Temporal Rover [9]
is a commercial code generator allowing programmers to insert specifications in programs via comments and then generating executable verification code, which is compiled and linked as part of the application under test, from the specifications. In addition, Java PathExplorer (JPaX) [19] is a runtime verification environment for monitoring the execution traces of a Java program by first extracting events from the executing program and then analyzing the events via a remote observer process.

5 Conclusion

This paper presents our ongoing research on model-based runtime verification technique, which will be applied to the self-optimizing systems. In fact, our model-level runtime verification can be seen as an extension to the state of the art runtime verification. That is, by introducing the pipelined cooperation between the verification and the application via the RTOS as intermediary, we are able to check if a component substitution still maintains safe and consistent at runtime, provided that a constraint on the checking time is required. Indeed we can check time-annotated ACTL and LTL properties, but not really limited to them. Up to now, we have not measured yet what the performance of our runtime verification will look like. Nevertheless, experience demonstrates that the properties to be checked in practice are usually not very complex [10]. Therefore, the size of the Büchi automata derived from these properties tends to be reasonable, what makes our model-based runtime verification mechanism applicable to the self-optimizing systems.

References


