A New Approach to Stateless Model Checking of LTL Properties

Elaheh Ghassabani
School of Computer Engineering
Iran University of Science and Technology
Tehran, Iran
ghasabani@comp.iust.ac.ir

Mohammad Abdollahi Azgomi*
School of Computer Engineering
Iran University of Science and Technology
Tehran, Iran
azgomi@iust.ac.ir

Abstract

Stateless model checking is an appropriate model checking technique for software verification. Existing stateless model checkers do not support the verification of linear temporal logic (LTL) because the existing algorithms of verifying LTL formulae are state-based, while stateless model checkers do not store any program states. This paper proposes a novel approach to stateless model checking of LTL formulae, based on the Actor formalism. Instead of translating an LTL formula into a Buechi automaton, which is the standard approach in model checking, the formula is translated into a set of actors that communicate with one another as well as with the main engine that explores the state space. As state space explosion is one of the main obstacles in practical applications of model checking, having such techniques that do not rely on recording of the visited states, can be a solution to this problem. We have modeled the proposed method using Rebeca, which is an actor-based modeling language with a formal foundation. The whole Rebeca model is translated into the Promela modeling language. Then, the models are verified using model checkers RMC and Spin. The proposed method modeled in Rebeca, the verification results, and an illustrative example are also presented in this paper.

Keywords: Software verification, stateless model checking, linear temporal logic (LTL), Actor model.

1. Introduction

In recent years, it has become more prevalent to develop concurrent programs in order to utilize the computational power of parallel or multi-core processors. Even by using conventional methods of testing, such as various forms of stress and random testing, it is still difficult to detect all concurrency errors in a program [1]. *Model checking* [2, 3] is a promising method for detecting and debugging concurrency errors [1, 3, 4]. However, traditional model checkers are not appropriate to verify code written in general purpose programming languages because they make users (manually) model their target systems. Therefore, the validity of verification results relies on the constructed model. For this reason, it is essential that the input model conform to the target code. However, modeling is a demanding task that needs special expertise and knowledge [5]. A solution to this problem, useful for real-world software, is to have a tool that verifies program code instead of the

*Corresponding Author. Address: School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Tehran, Iran, Postal Code: 16846-13114, Fax: +98-21-73225322

program model specified in a formal modeling language. Such tools are called *code model checkers*.

From one point of view, code model checking can be classified into two categories: (1) stateful model checking, and (2) stateless model checking. Although stateful techniques are ideally suited to verify sequential programs, they usually run into the state space explosion problem verifying parallel programs. Owing to saving (all) the state space, the rise in the concurrency level may result in more complexity as well as the exponentially growth of the state space. In such situations, stateless model checking can be useful. Stateless model checking is especially appropriate to explore the state space of large and complicated programs because accurate capturing and controlling all the needed states of a large program could be a hard, or even impossible, task [1, 6, 7]. Global variables, heap, thread stacks, and register contexts are all part of the program state. Even if all the program states could be captured and controlled, processing such large states would be very expensive [8, 9].

The notion of stateless model checking was proposed in [10] by Godefroid simultaneously with the appearance of code model checking. A stateless model checker explores the program state space without capturing program states. The program is concretely executed, and its state space is systematically explored. Therefore, all execution paths of the program generating by nondeterministic choices are covered [1, 11].

In this paper, we refer some model checking techniques as conventional. Needless to say, conventional model checkers are tools that apply such techniques. Generally speaking, model-based model checkers (e.g. `Spin` [12]) as well as stateful (or state-based) ones, such as `GMC` [13] and `Spin` [12], are known as conventional model checkers. Obviously, a stateful *code* model checker, such as `GMC` [13] and `MOPS` [14, 15], falls into the category of the conventional tools. Using a conventional model checker, users can formally specify their system properties. Most of these tools often support *linear temporal logic* (LTL) [16] for this purpose. On the downside, such tools face state space explosion verifying LTL in large programs.

We stated that stateless model checkers do not suffer from state space explosion owing to their stateless nature. However, as far as we know, existing *stateless (code) model checkers* do not support verifying LTL formulae because existing LTL checking algorithms are state-based. This paper proposes a new Actor-based method for stateless model checking of LTL formulae. Using this method, we are able to dynamically check any desired number of LTL formulae without having any concerns about state space explosion. The method is designed as an LTL verification method for a new stateless model checker called *distributed stateless code model checker* (`DSCMC`) [17, 18]. `DSCMC` has been developed based on the Actor model [19, 20]. This tool is suited to verify concurrent (multi-threaded) programs [17]. The proposed LTL verification method is also based on

the Actor model hence, all its internal components are independent actors running in parallel.

The remainder of this paper is organized as follows. Section 2 gives the formal background required for this paper. Section 3 covers related work. Section 4 describes the proposed method for verifying LTL formulae. In this section, we model our method using Rebca modeling language, specify the properties of the model in LTL, and then describe the verification process and results. Section 5 gives an example to illustrate the proposed method. Section 6 briefly discusses the implementation issues. Finally, Section 7 mentions some concluding remarks.

2. Preliminaries

This section presents the formal background of this paper. The first subsection states required formal definitions. The next subsection is a brief introduction to the semantics of LTL. Finally, the last subsection briefly introduces the Actor model [20] as well as Rebeca modeling language [21] used in order to model actors' interactions.

2.1. Program model

Transition systems are often used in computer science as models to describe the behavior of systems. They are directed graphs where nodes represent states and edges model transitions, i.e. state changes. A state describes some information about a system at a certain moment of its behavior. A state of a sequential computer program indicates the current values of all program variables together with the current value of the program counter that indicates the next program statement to be executed [3]. We use transition systems with atomic propositions for the states. Atomic propositions (*APs*) intuitively express simple known facts about the states of the system under consideration [3].

Definition 1. Transition system. A transition system *TS* is a tuple $(S, Act, \rightarrow, I, AP, L)$ where S is a set of states, *Act* is a set of actions, $\rightarrow \subseteq S \times Act \times S$ is a transition relation, $I \subseteq S$ is a set of initial states, *AP* is a set of atomic proposition, and $L : S \rightarrow 2^{AP}$ is a labeling function [3].

Here, 2^{AP} denotes the power set of *AP*. For convenience, we write $s \xrightarrow{\alpha} s'$ instead of $(s, \alpha, s') \in \rightarrow$. The intuitive behavior of a transition system can be described as follows. The transition system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\rightarrow$. That is, if s is the current state then a transition $s \xrightarrow{\alpha} s'$ originating from s is selected nondeterministically and taken, i.e. the action α is performed and the transition system evolves from state s into the state s'. This selection procedure is repeated in state s and finishes once a state is encountered that has no outgoing transitions [3].

For a sequential program, a program graph (*PG*) over a set of typed variables is a digraph whose

edges are labeled with conditions on these variables and actions. Intuitively, a program graph is like the program control flow graph (CFG), where executing an instruction changes the program state. Needless to say, each sequential program has a program graph, which can be interpreted as a transition system [3]. After interpretation, states of the transition system are pairs of the form *(l, η)* where *l* is a program location and *η* denotes values of all the program variables in location *l* [3]. Here, there is no need to state the formal definition and transition system semantics of a program graph (i.e. the definitions of *PG* and *TS (PG)*); for more information, please see [3].

In a computer program, *APs* can be defined on the program variables. These are known facts expressed in the form of the *simple conditions*. For example, in a program that has two boolean variables *a* and *b*, *APs* for states can be defined as different combination of simple conditions on these variables; e.g. s_i might have *APs* like *"a == true", "b == false",* etc.

A concurrent system is composed of a finite set of threads or processes, whose state space is defined by using dynamic semantics in the style of transition systems. Each process executes a sequence of statements in a deterministic sequential programming language, such as C, C++ or Java. Threads are a particular type of processes that share the same heap [22]. A multithreaded program can be modeled as a concurrent system, which consists of a finite set of threads, and a set of shared objects. Threads communicate with one another only through shared objects [23].

The transition system of a multithreaded program with *n* threads running in concurrent is defined as $TS\ (PG_1 \mid\mid\mid PG_2 \mid\mid\mid \ldots \mid\mid\mid PG_n)$ where PG_i is the program graph of i^{th} thread and ||| denotes the interleaving operator. Interleaving means the nondeterministic choice between activities of the simultaneously acting threads. For the sake of simplicity, we do not mention the formal definition of interleaving of program graphs. For a precise definition, please see [3].

Definition 2. Path [3]. Let $\pi = s_0\ s_1\ s_2\ \ldots$ be an infinite path of transition system *TS*, where s_i is a state of the transition system. A path is formed from a sequence of actions. Thus, π is formed from the execution of actions α_j for $j = 0, 1, 2, \ldots$ such that $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \cdots$.

Definition 3. A path of *s* is a path started from *s*. *Paths(s)* is called the set of all the paths of *s*. *Paths(TS)* is a set of all the paths in *TS* [3].

2.1.1. Stateless model checking

A stateless model checker explores the state space of a program without capturing any program states. The program is executed under the control of a special scheduler, which systematically enumerates all execution paths of the program obtained by the nondeterministic choices. In other words, the scheduler controls the nondeterministic execution of threads [1, 7, 11]. Obviously, this

method is a kind of code model checking which is execution-based [24].

In this paper, we also follow the Godefroid's method [7]. As this method is applied to the source code level, it is very similar to software testing. In fact, it is a systematic testing method. A stateless model checker systematically explores all possible interleavings of threads in the program under specific input for that program. Intuitively, a stateless model checker explores the state space of a program by concretely and continuously re-executing the program such that the model checker generates a different thread scheduling scenario for each execution. [25].

As a stateless model checker concretely executes programs, it can be a time-intensive process to verify a program. For this reason, the existing stateless model checkers explore the program state space from a fixed initial state (i.e. from fixed program input) as if set *I* had only one member. However, It is worth mentioning that even by taking this approach, they can find many concurrency errors in large programs, which are impossible to detect using conventional model checking [1, 22, 23, 26, 27]. When a program is executed, the execution is equal to a path of its state space (i.e. a path of *TS(PG)*). It should be pointed out that the process of stateless model checking is composed of finite *iterations*. Each iteration is equivalent to the execution of program *P* under the control of the model checker scheduler.

Definition 4. Each execution of program *P* (i.e. each iteration of stateless model checking for *P*) is a path in the transition system of the program graph of *P* (i.e. a path in $TS(PG_P)$). Hereafter, we define some notations:

1. $SMC(P, s_0)$ is a function of stateless model checking (i.e. state space exploration) for program *P* from the initial state s_0 such that $SMC(P, s_0) \hookrightarrow Paths(s_0)$, where $\hookrightarrow$ is the notation of mapping in a function, and $Paths(s_0) \in Paths\ (TS(PG_p))$.
2. $TS = (S, Act, \rightarrow, I, AP, L)$ is the transition system of the program graph for program *P* (i.e. $TS = TS(PG_p)$) during $SMC(P, s_0)$, where $s_0 \in I$.
3. i_{j,s_0} denotes j^{th} iteration of the stateless model checking process (*SMC*), which denotes the process is started from $s_0 \in I$.
4. $i_{j,s_0} \hookrightarrow \pi_j,\ \pi_j \in Paths(s_0),\ Paths(s_0) \in Paths(TS(PG_p))$.

2.1.2. Fair stateless model checking

In this paper, we take the approach proposed by Musuvathi and Qadeer [1] as fair stateless model checking. In this method, it is unexpected for a program not to terminate under a fair schedule. In other words, non-termination under fair scheduling is potentially an error [1]. Our method is also applicable to programs that are expected to terminate under all fair schedules. However, these

programs may not terminate under unfair schedules. Such programs are called fair-terminating [1].

The concept of fair-terminating programs is based on the observation of the test harnesses for real-world concurrent programs. Practically speaking, concurrent programs are combined with a suitable test harness that makes them fair-terminating when it comes to testing. By doing so, every thread in the program is eventually given a chance by the fair scheduler to make progress, which guarantees the (correct) program as a whole can make progress towards the end of the test. Such a test harness can be created even for systems such as cache-coherence protocols that are designed to "run forever"; the harness limits the number of cache requests from the external environment [1].

Therefore, our method is applicable to a fair stateless model checker that has an explicit scheduler that is (strongly) fair and at the same time sufficiently nondeterministic to guarantee full coverage of safety properties. Such fair scheduler has been implemented in the CHESS model checker [1, 28, 29] as well as DSCMC [17, 18].

Definition 5. Fairness [3]. Every thread that is enabled infinitely often gets its turn infinitely often.

It should be noted that stateless model checkers expect the program under test to eventually terminate. In other words, practically, it is not possible for a stateless model checker to identify or generate an infinite execution. They have some mechanism to deal with non-terminating programs [1, 10, 25, 26, 28]. For example, they may ask the user to set a large bound on the execution depth. This bound can be orders of magnitude greater than the maximum number of steps the user expects the program to execute. The model checker stops if an execution exceeds the bound, and reports a warning to the user. The user can examine this execution to see whether it actually indicates an error. In the rare case it is not, the user simply increases the bound and runs the model checker again [1].

Above all, the stateless model checkers that do not apply the fair stateless model checking method, like Inspect [25, 30], are unable to properly verify the nonterminating programs that are fair-terminating. This is because they cannot detect existing cycles in the state space [1, 10, 25].

2.1.3. Program states

In a multi-threaded program containing a finite set of threads and a set of shared objects, threads communicate with each other only through shared objects. Operations on shared objects are called visible operations, while the rest are invisible operations. A state of a multi-threaded program contains the global state of all shared objects and the local state of each thread. In a multi-threaded program, a visible operation performed by a thread is considered as a transition that advances the program from one global state to a subsequent global state. Such a transition is followed by a finite sequence of invisible operations of the same thread, ending just before the next visible operation of

that thread [22, 23].

To avoid exploring redundant interleavings, stateless model checkers should use dynamic partial order reduction (DPOR) [22] because the number of possible interleavings grows exponentially as the program is getting large. Partial order reduction algorithms only explore a proper subset of the enabled transitions at a given state *s* such that it is guaranteed to preserve the interested properties. DPOR dynamically tracks threads interactions to identify points where alternative paths in the state space need to be explored [22, 25].

To perform DPOR, a stateless model checker explores the program state space by concretely executing the program and observing its visible operations. It considers consecutive invisible operations with only one visible operation as a single operation [11, 17]. In this paper, we use the notion of code partitioning. Stateless model checkers are expected to apply some mechanisms for detecting global transitions. Therefore, we refer such mechanisms to partitioning, whereby code is divided into several global locations. In fact, the model checker interleaves threads according to these locations.

Each partition of the code (each location) starts with a visible operation, and ends just before the next visible operation. When a thread is scheduled, if it can progress, it continues executing until the end of its current location. After reaching the end of a location, it yields the processor to the model checker. If the thread holding the processor cannot progress, the scheduler should choose another thread. It goes without saying that this event may occur at the beginning of a location because only the first command of each location can be a waiting function call (a visible operation). Therefore, when it comes to LTL checking, we use the described definition for a state.

2.2. Linear temporal logic (LTL)

This subsection is a brief introduction to (propositional) linear temporal logic [16], a logical formalism that is appropriate for specifying linear-time (LT) properties [3]. LTL is called linear because the qualitative notion of time is path-based and viewed to be linear: at each moment of time there is only one possible successor state, and thus each time moment has a unique possible future. Technically speaking, this follows from the fact that the LTL formulae are path-based (i.e. they are interpreted in terms of sequences of states) [3].

Definition 6. LT Property [3]. A linear-time property (LT property) over the set of atomic propositions *AP* is a subset of $(2^{AP})^{w}$.

Here, $(2^{AP})^{w}$ denotes the set of words that arise from the infinite concatenation of words in 2^{AP}. An *LT property* is thus a language (set) of infinite words over the alphabet 2^{AP}. LTL formulae over the set *AP* of atomic proposition are formed according to the grammar

$\phi ::= true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \phi_1 \, U \, \phi_2$ where $a \in AP$. Here, for the trace $\sigma = A_0 A_1 A_2 \ldots \in (2^{AP})^w$, $\sigma[j\ldots] = A_j A_{j+1} A_{j+2} \ldots$ is the suffix of σ starting in the $(j+1)^{st}$ symbol A_j [3, 16]. The satisfaction relation $(\vDash)$ is defined as follows [3, 16]:

- $\sigma \vDash true$.
- $\sigma \vDash a$ *iff* $a \in A_0$ (i.e. $A_0 \vDash a$).
- $\sigma \vDash \phi_1 \wedge \phi_2$ *iff* $\sigma \vDash \phi_1$ *and* $\sigma \vDash \phi_2$.
- $\sigma \vDash \neg \phi$ *iff* $\sigma \nvDash \phi$.
- $\sigma \vDash \phi_1 \, U \, \phi_2$ *iff* $\exists j \geq 0.\ \sigma[j\ldots] \vDash \phi_2$ *and* $\sigma[i\ldots] \vDash \phi_1$, *for all* $0 \leq i < j$.

For the derived operators *F* (i.e. "eventually", sometimes in the future) and *G* (i.e. "always", from now on forever) the expected result is [3]: $F\,\phi \stackrel{def}{=} true \, U \, \phi$, and $G\,\phi \stackrel{def}{=} \neg F \neg\phi$.

Definition 7. Semantics of LTL over paths and states [3]. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system, and ϕ be an LTL-formula over *AP*. *TS* satisfies ϕ, denoted $TS \vDash \phi$, iff $\pi \vDash \phi$ for all $\pi \in Paths(TS)$.

For LTL checking, it is usually assumed that all paths and traces of a transition system are infinite. This assumption is made for the sake of simplicity; it is also possible to apply the semantics of LTL to finite paths. In other words, for LTL semantics it is irrelevant whether or not *TS* is finite [3]. Therefore, while stateless model checking, the fact that programs are expected to be fair-terminating makes no difference to LTL semantics.

2.3. Actor model

Actor is a model for concurrent computing to develop parallel and distributed systems. Each actor is an autonomous entity that acts asynchronously and concurrently with other actors. It can send/receive messages to/from other actors, create new actors, and update its own local state. An actor system is composed of a collection of actors, some of whom may send messages to (or receive messages from) actors outside the system [31]. An actor using a command like *send(a, v)* creates a new message with receiver *a* and contents *v*, and then puts it to the message delivery system. This system guarantees the received message will be finally delivered to actor *a*. It can create another actor with a command like *newadr()*. Suchlike commands create a new actor and return its address. Each actor may have its own behaviors to process received messages. In other words, an actor's behavior embodies the code that should be executed by the actor after receiving a message [19].

As we stated, this paper uses the Actor model to propose its new verification method, which is also implemented by using an actor language. An actor language is an extension of a functional language. Erlang [32, 33] is arguably the best known implementation of the Actor model [31]. We

are implementing the method proposed in this paper by using Erlang. In such languages, *functions* are used to define actors' behaviors. That is, each actor has a *behavioral functional* that embodies the actor behaviors after receiving particular messages.

In this paper, we model our method using Rebeca (Reactive Object Language) [34, 35], which is an actor-based modeling language with a formal foundation [34]. Then, we use the model checking technique to verify our models. For this purpose, model checker RMC [36] is used, which is a tool for direct model checking of Rebeca models, without using back-end model checkers. Using RMC, properties should be specified based on *state variables* of *rebecs*.

Rebeca is a Java like language, which is mainly a modeling language with formal verification support and a background theory [21]. A Rebeca model consists of concurrently executing reactive objects called rebecs. In fact, rebecs are actors that communicate with each another by asynchronous message passing. Each message is put in the unbounded queue of the receiver rebec, specifying a unique method to be invoked when the message is serviced [35].

Fig. 1 illustrates the definition of a simple Rebeca class. Although in a pure actor model the queue length is unbounded, the modeler has to declare the maximum queue size in the class definition owing to model checking. This size is indicated in parenthesis next to the *reactiveclass* name. A class definition uses two central declarations *knownrebecs* and *statevars*. The *knownrebecs* entry shows the actors this rebec can communicate with. The rebecs included in the *knownrebecs* part of a reactive class definition are those rebecs whose message servers may be called by instances of this reactive class. The *statevars* defines variables used for holding the rebec state [37].


```
reactiveclass Rebec1(5) {
    knownrebecs { Rebec2 actor2 ; }
    statevars { }
    msgsrv initial ( ) {
            self.msg1 ( );
    }
    msgsrv serv_msg1( ){
            /* Send a message to actor2, which should be processed by
               method process_msg in reactiveclass Rebec2. This message
               contains an integer value like "7" */
            actor2.process_msg (7);
    }
    msgsrv serv_msg2 ( ){
            /* Handling message 2 */
    }
}
```

Fig. 1 A typical class definition in Rebeca

After these declarations, the methods that handle messages are defined like Java code. These methods are called the message servers of the *reactiveclass* because their task is to serve incoming messages. Each reactive class definition has a message server named *initial*. In the initial state,

each rebec has an initial message in its message queue, thus the first method executed by each rebec is the *initial* message server. A message server contains one or more Rebeca statements. The logical and arithmetic expressions in Rebeca are similar to Java. However, not all of the Java expressions are valid in Rebeca, and only a set of essential set of operators are included [37]. For more information about Rebeca, please see [34, 37].

The execution of rebecs in a Rebeca program takes place in a coarse grained interleaving scheme. In this manner, each rebec takes a message from the top of its queue and executes its corresponding message server. During execution, other rebecs are not allowed to be executed; i.e. the execution of a message server is atomic [37, 38].

3. Related work

As far as we know, prior to DSCMC [17, 18], there have been three stateless model checkers, namely Inspect [11, 30], CHESS [28, 29], and VeriSoft [7, 39]. Among these tools, CHESS and Inspect are concerned with verifying multi-threaded programs. The strengths of Inspect are that it is distributed, and supports POSIX threads [26, 40]. On the downside, it cannot deal with programs with cyclic state space. In comparison with Inspect, CHESS is able to detect and prune unfair cycles in the program state space so it can be used for detecting problems related to fair cycles (e.g. livelocks) [1].

Unfortunately, none of the foregoing tools support verifying LTL formulae because existing algorithms of checking LTL formulae are based on graph algorithms and need to save the transition system of a program as the graph of its state space. In other words, these algorithms are state-based so they are applicable to stateful model checking techniques. A stateless model checker does not save any program states. Therefore, it is impracticable to apply existing LTL checking algorithms to stateless model checking.

LTL checking algorithms usually follow an automata-based approach taken from [41]. In this approach, the negation of the LTL formula is translated into a *Buchi automaton* [3, 42], synchronized with the transition system of the program state space, and then the verification problem is reduced to a simple graph problem [42]. Handling of large state spaces is so difficult (or even impractical) that the state space explosion has always been a pressing and serious problem in the stateful model checking field.

In order to verify LTL formulae, stateful model checkers have to capture the state space of the program, and the number of states grows exponentially in the number of variables in the program graph: for N variables with a domain of k possible values, the number of states grows up to k^N. Even if a program only contains a few variables, the state space that must be analyzed may be very large. This exponential growth in the number of parallel components and the number of variables

leads to the enormous size of the state space of practically relevant systems. The reality is that verification problem in stateful model checking is particularly space-critical [3]. Nevertheless, many researches have been undertaken into this field leading the way to great achievements including some recent work in [43-46].

Of all the researches in this area, the work by Ganai *et al.* [46] is more relevant to stateless model checking. Coping with state space explosion, they combined state-based and path-based (like stateless method) model checking, and then used a divide and conquer technique to explore state space. The main focus of their work is on proposing a new state exploration technique by combining state-based and path-based methods together. In other words, they also used the conventional techniques for verifying LTL formulae and did not propose a new LTL verification method (the focus of this paper).

Another work in this area was carried out by Evangelista and Kristensen [43]. They proposed an algorithm that is a combination of the common on-the-fly LTL model checking algorithms with sweep-line method [47]. Conventional on-the-fly LTL model checking is based on the exploration of a product Buchi automaton; i.e. the negation of the LTL formula to be checked is represented as a Buchi automaton, and then the product of this property automaton and the state space, viewed as a Buchi automaton, are explored using a nested depth-first traversal [42] in search for a cycle containing an acceptance state (an acceptance cycle). This work also has nothing to do with stateless model checking and is appropriate to the stateful techniques.

De Wulf *et al.* [44] proposed algorithms for LTL satisfiability and model-checking. In their algorithms nondeterministic automata were not constructed from LTL formulae. They directly alternated automata using efficient exploration techniques based on anti-chains. Similar to the previous work, their method is also not suitable for stateless model checking.

In this literature, the concept of runtime verification really stands out, which checks whether a system execution satisfies or violates a given correctness property [48]. A procedure that on-the-fly verifies conformance of the system's behavior to the specified property is called a monitor. Nowadays, there are a variety of formalisms to specify properties on observed behavior of computer systems including variants of temporal logic such as LTL_3 and TLTL [49]. In addition, currently, a lot of methods have been proposed to construct monitors [48-50].

The main idea of runtime verification is to monitor and analyze software and hardware system executions. Although this idea is fairly analogous to the idea of stateless model checking, methods used for runtime verification are completely different. In runtime verification, monitoring is carried out as follows. Two "black boxes", the system and its reference model, are executed in parallel and stimulated with the same input sequences; the monitor dynamically captures their output traces and tries to match them. The main problem is that a model is usually more abstract than the real system,

both in terms of functionality and timing. For this reason, trace-to-trace matching is difficult, which causes the system to generate events in different order or even miss some of them [48].

To sum up, as far as we know, any LTL verification method in the stateless model checking field has not been proposed yet; this paper presents a new LTL checking method for this field.

4. Method

This section describes a novel method for stateless model checking of LTL formulae. In this method, LTL formulae are dynamically checked during program execution without storing any program states. For this reason, it is possible to verify any number of LTL properties away with affecting on the size of the program state space and state space explosion. The method to verify LTL formulae, proposed in this section, is quite different from conventional LTL checking algorithms.

In our method, we suppose that there is a stateless model checker that runs the program under test and systematically explores its state space. This model checker should accept all possible interleavings under strong fairness [1, 3, 17]. To generate different possible interleavings, the program must be repeatedly run under the stateless model checker until all possible thread scheduling options are generated. This paper concentrates on how LTL properties can be verified using such model checkers. Our method is proposed as a *unit of LTL checking*, which should cooperate with the stateless model checker. This unit is an actor system, a collection of actors with a hierarchical structure. To verify (rather than systematically test) a program, we need a new definition of stateless model checking, called *complete stateless model checking (CSMC)*.

Definition 8. ***CSMC***. Let program P have n possible initial states, $n \geq 1$ (i.e. the set I has n members). *CSMC* for P, denoted $CSMC_{P,n}$, is defined as a set of stateless model checking functions: $CSMC_{P,n} = \cup SMC(P, s_i)$ *for* $i = 1, 2, \ldots, n$.

Corollary 1. If $I = \{s_0\}$ then $CSMC_{P,1} = SMC(P, s_0)$.

In fact, the definition of *CSMC* includes all the possible members of the I set, whereas *SMC* considers only one selected member of this set. Hereafter, for the sake of simplicity, we suppose that the initial states set, I, has only one member then according to Corollary 1, $CSMC_{P,1} = SMC(P, s_0)$. Therefore, Theorem 1 is stated based on this premise, but it can easily be extended to a program with any number of initial states. LTL formulae are considered based on paths of a transition system. In our method the whole process of checking LTL is performed through different program executions.

Theorem 1. Program *P* satisfies LTL property ϕ iff ϕ is held by all iterations of stateless model checking:

$$TS\left(PG_p\right) \vDash \phi \text{ iff } i_{j,s_0} \vDash \phi \ , \forall i_{j,s_0} \in SMC\left(P,\ s_0\right)$$

PROOF.

1. During *SMC(P, s_0)*, according to Definitions 3 and 4: $i_{j,s_0} \hookrightarrow \pi_j$, $\pi_j \in Paths(s_0)$, $Paths(s_0) \in Paths(TS(PG_p))$. As a result, π_j is obtained from i_{j,s_0}.
2. From Definition 8: $TS(PG_p) \vDash \phi$, *iff* $\pi \vDash \phi$ *for all* $\pi \in Paths(TS)$.
3. Based on Corollary 1, $CSMC_{P,1} = SMC\ (P, s_0)$; consequently $Paths(TS) = Paths(s_0)$.
4. According to 2 and 3: $TS(PG_p) \vDash \phi$, *iff* $\pi \vDash \phi$ *for all* $\pi \in Paths(s_0)$.

In consequence of 1 and 4,

$$TS\left(PG_p\right) \vDash \phi \text{ iff } i_{j,s_0} \vDash \phi \ , \forall i_{j,s_0} \in SMC\left(P,\ s_0\right) \qquad \square$$

Intuitively, if an LTL property is violated in one program execution, it means that the property has been violated in one path of $TS(PG_P)$; consequently, the program does not satisfy this property. In the same way, if an LTL property is held by all iterations of stateless model checking then the property is satisfied by all paths of $TS(PG_P)$; consequently, the program satisfies the property.

We use this theorem as a basis for LTL checking in stateless model checking. It shows the feasibility of applying LTL checking to this field. But, the major need in this regard is to have an LTL checking method that can work with the stateless nature of the model checker. The remainder of this section proposes such method to solve this problem.

4.1. An actor system for LTL checking

In light of the grammar of LTL formulae [3], terminals in this grammar are atomic propositions (i.e. $a \in AP$) [3]. As we mentioned in Section 2, an atomic proposition is a simple condition defined on program variables (e.g. *a > 0*, *b = 0*, *c != d*, etc.). Therefore, every LTL formula ends in simple conditions. The result of a simple condition is always either *true* or *false*. We use this fact for designing the unit of LTL checking.

Now, let us introduce the idea of the method with a simple example. Suppose you specify an LTL property as *"(¬ ((a > 0) ∧ (b = 1))) U (c = 0))"* where *a*, *b*, and *c* are integer variables in the program. The parse tree for this property is shown in Fig. 2. All LTL properties, like this property, are evaluated from leaves towards the root of the parse tree; i.e. in this example, first, operator *and* (∧) should be evaluated, next, the *not* operator (¬), and then operator *until* (*U*) can be evaluated. We exploit this fact in our method; as it can be seen, leaves of a parse tree are simple conditions (or

APs) while both of its root and intermediate nodes are LTL operators.

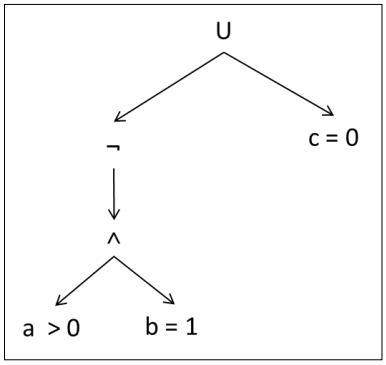

Fig. 2 The parse tree of *(¬ ((a > 0) ∧ (b = 1))) U (c = 0))*

To check an LTL formula by our method, first a property is parsed, next an actor whose behavior corresponds to the root operator is created, and then existing sub-trees of the root are sent to the behavioral function of this actor as its arguments; e.g. in the above example, an actor who behaves corresponding to operator *U* is created and two sub-trees are sent to its behavioral function as its input arguments. Thereafter, this actor also makes the parse tree for each input argument (i.e. each sub-tree). In the same way, an actor for the root operator of each sub-tree is created and related sub-trees are sent to them. This process is continued by new actors until reaching the leaves; e.g., in this example, the actor with *until* behavior creates another actor with *not* behavior, and then the created actor creates a new actor with *and* behavior. When this new actor reaches a leaf (i.e. a simple condition) after parsing one of its arguments, it should create a new actor that checks a simple condition (we call such actors *condition checkers*). The intermediate nodes of the primary parse tree are called *workers* that are actors that behave corresponding to LTL operators. This hierarchical structure described here is shown in Fig. 3 (a). There are two other kinds of actor in this hierarchy, *property checker* and *master*, which are described below.

In this paper, we suppose the existence of a mechanism in the model checker so that *condition checkers* are be able to monitor the state of the intended *APs*. At the implementation level, the model checker can think of different mechanisms. For example, based on the property the user has defined, it can instrument the program code such that at every point in the code that the variables in the *APs* of the property are defined[a], a piece of code is added to the original code, by which the simple conditions in the property (i.e. *APs*) can be monitored during stateless model checking. By doing so, *condition checkers* are informed about the status of their desired *APs* at the end of each state.[b] A similar mechanism has been implemented in `DSCMC` [17].

[a] The variable definition means that a new value is assigned to the variable (*e.g.* using of the assignment operator *"="*).

[b] The definition of a state in the context of stateless model checking is given in Section 2.1.3.

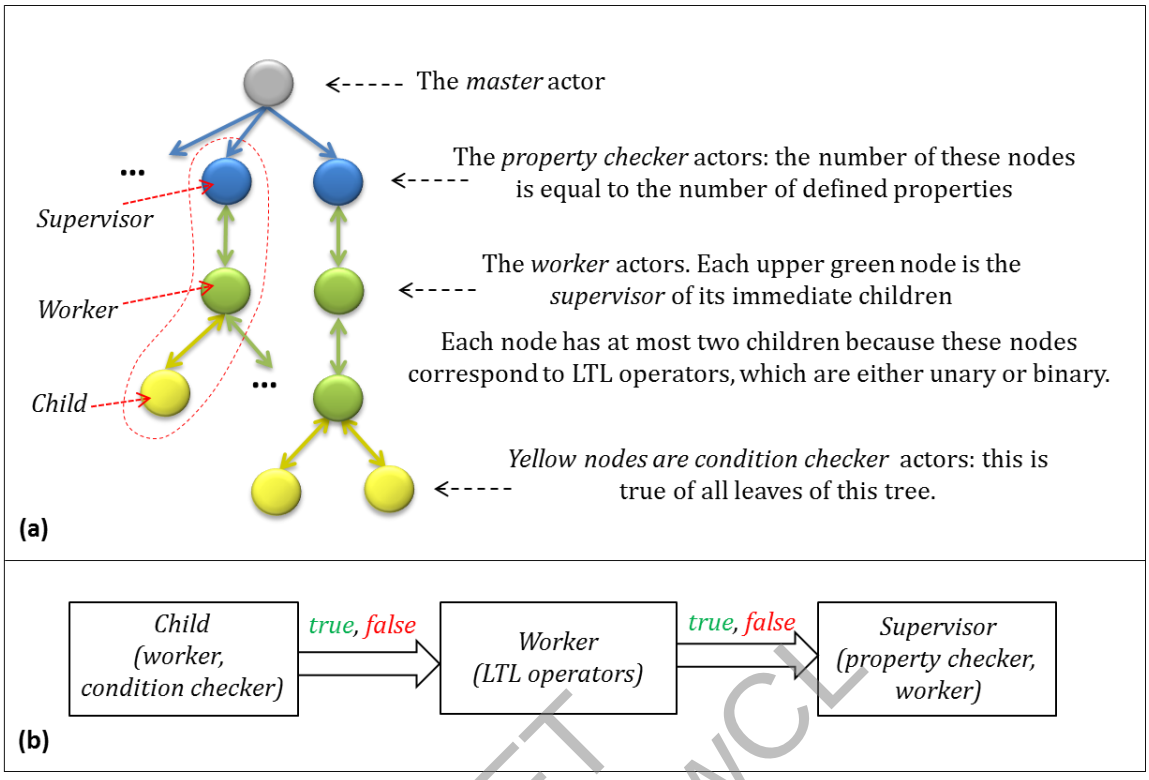

Fig. 3 The hierarchical structure of the *Unit of LTL checking*. (a) Existing actors and their roles. (b) Usage of the hierarchical structure of the actor system for modeling.

The *unit of LTL checking* (Fig. 3 (a)) has a major actor as the *master* actor, whose task is to load the user-defined LTL properties at the beginning of stateless model checking, and then create a *property checker* actor for each property. The *property checker* actors use a function for parsing a given property. This function creates the parse tree of its input argument, and then returns the root of this tree and sub-trees of the root. Thereafter, the *property checker* creates a *worker* that will be in charge of the sub-tree. The return sub-tree is also sent as an input argument to the behavioral functions of this *worker*.

As it was previously pointed out, the created *worker* by *property checker* also parses its input arguments (i.e. sub-tree(s) sent by *property checker*). Then, with respect to the parse tree of its arguments, it also creates other *worker* actor(s). Needless to say, *workers* are different in their behavior. Permissible behaviors for *workers* exactly correspond to LTL operators. For example, *and worker*, *not worker*, and *until worker* have the same semantics of operators $\wedge$, $\neg$, and U, respectively. You can see the procedure for creating the described hierarchy in Fig. 4.

1. master (Properties) {
Load Properties;
For each Property in the Properties list
Create a *property checker* actor;
Assign each *property checker* a Property;
}

2. property checker (Property) {
Parse Property;
For the root operator
Create a *worker* actor with behavior of the root;
Assign the subtree of the root to the *worker*;
}

3. worker (Args) {
Parse each available subtree in Args;
For each available root operator in subtrees of Args
Create a *worker* actor with behavior of the root;
Assign the subtree of the root to the *worker*;
If there is no LTL operator in the root
Create a *condition checker*;
Assign the related AP to the *condition checker*;
}

Fig. 4 Procedure for creating the verification hierarchy

Our method for verifying LTL formulae is similar to the *divide-and-conquer* method; each *property checker* actor parses an LTL property, and then creates a *worker* actor to evaluate the operator in the root of the parse tree. Consequently, the created *worker* also repeats this process until a *worker* actor reaches the simple condition(s). In other words, tasks are downwardly dispatched, then, results are upwardly collected from *workers* to their supervisors, and finally the results of evaluating get to *property checkers*, which are at the top level of the verification hierarchy (of course, after the *master*).

4.2. Modeling in Rebeca

This section models the actor system described in the previous subsection. In this regard, we use Rebeca modeling language [34, 35, 38].

We need an abstract model that correctly embodies the possible interactions between actors. For this purpose, we exploit the hierarchical structure shown in Fig. 3. In this structure, the position of an intermediate node (*worker*) is similar to Fig. 3 (b). As described earlier, *workers* correspond to LTL operators. The behavior of each *worker* actor is modeled in Rebeca using the structure shown in Fig. 3. That is, each *worker* has a supervisor and at least one child. That is to say, each *worker* is an LTL operator that can have at most two children. Each child may also be an LTL operator. Besides, *condition checkers* are also children of their immediate parent (*worker*). Each *worker* has one supervisor (its immediate parent), which may be an LTL operator or a *property checker*. As it can be seen, a child only sends its evaluation result (*true* or *false*) to its *supervisor*, regardless of whether it is a *condition checker* or an LTL operator. A *supervisor* also can receive either *true* or *false* from the *worker* regardless of the fact that the *worker* is which LTL operator.

On account of the above structure, it does not matter to a *worker* who its children and its supervisor are. Every *worker* only receives *true* or *false* from its children, and only sends *true* or *false* to its supervisor. Therefore, we can model the LTL checking unit, and verify the behavior of each actor (i.e. behavioral functions) independently. In this model, the behavior of each *worker* is

characterized in Rebeca, and then other actors the *worker* can communicate with are modeled as black boxes that correspond to the same *worker's* supervisor and children. That is, black boxes used in the model, namely *Child*, and *Parent*, are actors that behave like a *child* and a *supervisor*, respectively (see Fig. 3). A child is expected to send only either *true* or *false* to the *worker*, and a supervisor also expects to receive the result of the verification from the *worker*.

Before moving on to modeling, we should point out the role of actors *master*, *property checker*, and *condition checker*. These actors are important when it comes to the implementation of the model. In terms of modeling, it makes no difference to the result of verification who creates *property checkers* and *workers*, or how the model checker informs *condition checkers* about the *APs* status. The main focus of the model should be maintained on how a *worker* behaves as a particular LTL operator, and how it evaluates its operands. For this reason, we model a *worker* regardless of who its parent and child (children) are. For instance, as for the *Until* operator, the model should demonstrate the way by which this actor evaluates results of its operand; e.g. how to act when it receives a *true* message from its left operand, how to act when it never receives a *true* from its right operand, and so on.

We model the behavior of the *workers* that correspond to *and (∧)*, *not (¬)*, and *Until (U)* operators. Other LTL operators can be derived from these operators. Each of the forgoing operators is independently modeled; i.e. the children (or supervisor) of each operator are viewed as a black boxes that only send (or receive) *true* and *false*.

Fig. 5 shows program graphs of *workers*. In these state charts, the parts of the model where actors are created and killed are omitted in order to simplify models.

Fig. 6 models three rebecs that correspond to (a) stateless model checker, (b) *child*, and (c) *supervisor*. The rebec who models stateless model checker, *SMC*, initiates the execution of the model from the method *sendAP* on line 10 of Fig. 6 (a). This rebec models the fact that at the end of each state, the stateless model checker sends the status of the desired *AP*s to the *condition checkers*. After that, *condition checkers* send the results to their supervisors, and next their supervisors, according to their own functionality, evaluate the results and send them to their own supervisors. In practice, this process should continue until the most upper *worker* evaluates the results and sends the result of its evaluation to its own *property checker*. In the models, we suppose that *Child* (Fig. 6 (b)) is an intermediate *worker* that its immediate *supervisor* is one of the LTL operators *and*, *not*, or *until*. Practically, such an actor receives the results of verification from its children, but here, rebec *Child* itself randomly generates this results at line 14, Fig. 6 (c).

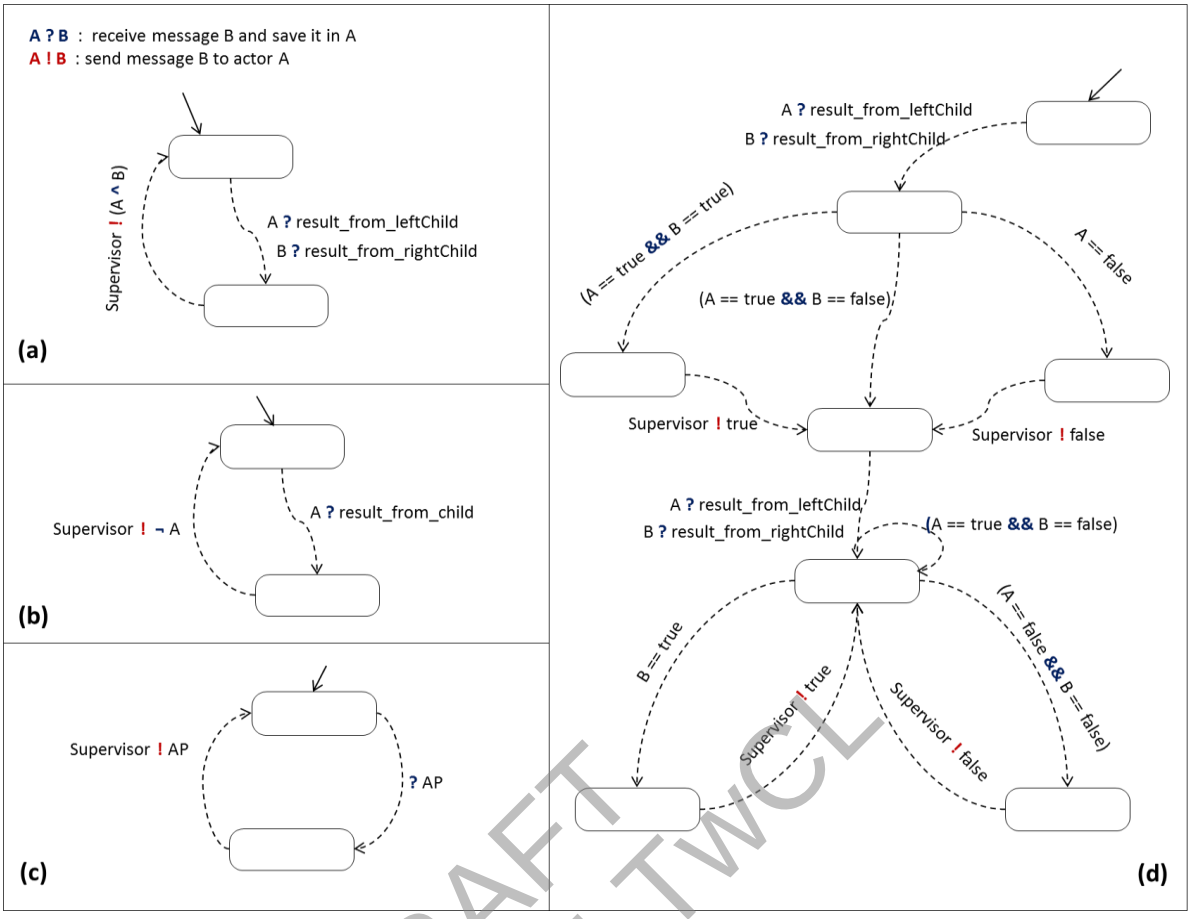

Fig. 5 Program graphs for the behaviors of *workers*—(a) Behavior of the *And worker*. (b) Behavior of the *Not worker*. (c) Behavior of the *condition checker*. (d) Behavior of the *Until worker*.

There are two rebecs of *Child* in our model: *leftWorker* and *rightWorker*. For a binary operator, the worker corresponding to that operator receives two results: one is sent by *rightWorker* and the other is sent by *leftWorker*. As for the unary operator *not*, only messages from *leftWorker* are processed.

In the models, rebec *Supervisor* (Fig. 6 (b)) models the immediate *supervisor* of the *worker* whose behavior is supposed to be modeled (i.e. one of the *workers* that acts as one of the LTL operators *not*, *and*, *until*). *Supervisor* using message server *result_fromOp* receives the result of verification from such a *worker* (line 13, Fig. 6 (b)).

In Rebeca, verification is performed based on state variables of rebecs so for the rebecs in Fig. 6., two variables used to specify properties of our models are state variables *result* and *resultReceived* in rebec *Supervisor* (lines 4-5 , Fig. 6 (b)). The received result from the LTL operator (*worker*) is saved in variable *result*. We need variable *resultReceived* while modeling because Rebeca initializes state variables at the beginning of execution so the variable *result* has a value even before receiving the real result from the *worker*. Therefore, when *resultReceived* turns into *true*, it denotes that the *Supervisor* has just received the *result* from the *worker* (at line 14 Fig. 6 (b)).

```
1. reactiveclass SMC (2) {
2.      knownrebecs {
3.              Child leftWorker;
4.              Child rightWorker;
5.      }
6.      statevars {      }
7.      msgsrv initial ( ) {
8.              self.sendAP( );
9.      }
10.     msgsrv sendAP ( ) {
11.             leftWorker.send_result ( );
12.             rightWorker.send_result ( );
13.     }
14. }
```

(a)

```
1.  reactiveclass Supervisor (2) {
2.      knownrebecs {     }
3.      statevars {
4.              boolean result;
5.              boolean resultReceived;
6.              boolean endOfVerification;
7.      }
8.      msgsrv initial ( ) {
9.              result = false;
10.             resultReceived = false;
11.             endOfVerification = false;
12.     }
13.     msgsrv result_fromOp (boolean res) {
14.             resultReceived = true;
15.             result = res;
16.             endOfVerification = true;
17.             self.endV( );
18.     }
19.     msgsrv endV ( ) {    self.endV ( );    }
20. }
```

(b)

```
1.  reactiveclass Child (2) {
2.      knownrebecs { And operator; }
3.      statevars {
4.              boolean result;
5.              boolean isLeft;
6.              boolean isRight;
7.      }
8.      msgsrv initial(boolean position) {
9.              result = false;
10.             if (position){  isRight = true; isLeft = false;  }
11.             else {   isRight = false;  isLeft = true;   }
12.       }
13.       msgsrv send_result( ) {
14.             result = ?(true, false);
15.             if  (isLeft)    operator.from_leftChild (result);
16.             else  operator.from_rightChild (result);
17.       }
18. }
```

(c)

Fig. 6 Rebeca model for the stateless model checker, children and supervisors. (a) The rebec for *Stateless model checker*. (b) The rebec for *Supervisor*. (c) The rebec for *Child*.

4.2.1. Modeling LTL operator *Until*

Fig. 7 shows the rebec for the actor that behaves corresponding to LTL operator *until*. This rebec behaves similar to the state chart shown in Fig. 5 (d). As our method uses the Actor model, it is nondeterministic that which actor first processes its incoming messages. Therefore, in the model, you may see some code or state variables for required synchronization. For example, when *leftWorker* and *rightWorker* send their own results to rebec *Until*, it is unpredictable that which actor first sends its message. However, we know that both of them send messages about the same state. Therefore, rebec *Until* first requires to receive both of these messages, and then evaluates them. This situation is modeled using state variables *rFlag* and *lFlag* as well as message servers *from_leftChild* and *from_rightChild*.

```
1. reactiveclass Until(3) {
2.     knownrebecs { Supervisor parent; SMC mc;}
3.     statevars {
4.         boolean leftOp;    boolean rightOp;
5.         boolean rFlag;     boolean lFlag;
6.         boolean first; boolean updatedLeftOp;
7.         boolean updatedRightOp;
8.     }
9.     msgsrv initial( ) {
10.        first = true;
11.        leftOp = false;          rightOp = false;
12.        rFlag = false;           lFlag = false;
13.        updatedLeftOp = true;
14.        updatedRightOp = false;
15.    }
16.    msgsrv until_bhv( ) {
17.        if  (first) {
18.            first = false;
19.            if  (leftOp && rightOp) {
20.                parent.result_fromOp (true);
21.            }
22.            else if (!leftOp) {
23.                parent.result_fromOp (false);
24.            }
25.            else if (leftOp && !rightOp)  {
26.                mc.sendAP(  );
27.            }
28.        }
29.        else {
30.            if (rightOp) {
31.                parent.result_fromOp(true);
32.            }
33.            else if  (!leftOp && !rightOp) {
34.                parent.result_fromOp (false);
35.            }
36.            else if (leftOp)  {    mc.sendAP( );    }
37.        }
38.    }
39.    msgsrv from_leftChild (boolean res){
40.        leftOp = res;
41.        lFlag = true;
42.        if (lFlag && rFlag) {
43.            lFlag = false;
44.            rFlag = false;
45.            if (first)
46.                updatedRightOp = (rightOp & leftOp);
47.            else
48.                updatedRightOp = rightOp;
49.            updatedLeftOp = leftOp;
50.            self.until_bhv( );
51.        }
52.    }
53.    msgsrv from_rightChild (boolean res){
54.        rightOp = res;
55.        rFlag = true;
56.        if (lFlag && rFlag) {
57.            lFlag = false;
58.            rFlag = false;
59.            if (first)
60.                updatedRightOp = (rightOp & leftOp);
61.            else
62.                updatedRightOp = rightOp;
63.            updatedLeftOp = leftOp;
64.            self.until_bhv( );
65.        }
66.    }
67. }
```

Fig. 7 Rebeca model for *Until worker*

When both of variables *lFlag* and *rFlag* become *true*, it means that rebec *Until* has received the result from both of its children then it comes to processing. Therefore, method *until_bhv* at line 16 of Fig. 7 is executed. This method exactly represents the same concept shown in Fig.4 (d). In this method, the rebec uses variables *leftOp* and *rightOp*. Variable *leftOp* contains the value of the last message sent by *leftWorker*. In the same way, *rightOp* contains the value of the last message sent by *rightWorker*. In order to verify the model, we use two state variables *updatedLeftOp* and *updatedRightOp*, which contain the results respectively sent by *leftWorker* and *rightWorker* after synchronization. This is because the initial state that should be considered to verify the model is not the same initial state in the model. In light of the fact that Rebeca itself initializes state variables, *leftOp* and *rightOp* contains initial values before receiving any messages from *leftWorker* and *rightWorker*. This situation brings about some problem while specifying properties of the model because the *until* operator is sensitive to the initial state of its operands. To resolve this issue, we use auxiliary variables *updatedLeftOp* and *updatedRightOp* for specifying properties.

4.2.2. Modeling LTL operator *And*

The rebec corresponding to LTL operator ∧ is shown in Fig. 8. This rebec also behaves according to the state chart shown in Fig. 5 (a). This rebec also first receives the results from both of its children using message servers *from_leftChild* and *from_rightChild*, and then it behaves as LTL operator ∧ using method *and_bhv* at line 11 of Fig. 8.

Rebec *And* uses two state variables *leftOp* and *rightOp* for saving the results sent by *leftWorker* and *rightWorker*, respectively. In addition, these variables are used for specifying the properties of the model as well as its verification.

```
1. reactiveclass And (3) {
2.     knownrebecs {  Supervisor parent;  }
3.     statevars {
4.             boolean leftOp;    boolean rightOp;
5.             boolean rFlag;     boolean lFlag;
6.     }
7.     msgsrv initial (  ) {
8.             leftOp = false;      rightOp = false;
9.             rFlag = false;       lFlag = false;
10.    }
11.    msgsrv and_bhv (  ) {
12.            parent.result_fromOp (leftOp & rightOp);
13.    }
14.    msgsrv from_leftChild (boolean res){
15.            leftOp = res;
16.            lFlag = true;
17.            if (lFlag && rFlag) {
18.                    lFlag = false;
19.                    rFlag = false;
20.                    self.and_bhv( );
21.            }
22.    }
23.    msgsrv from_rightChild (boolean res){
24.            rightOp = res;
25.            rFlag = true;
26.            if  (lFlag && rFlag) {
27.                    lFlag = false;
28.                    rFlag = false;
29.                    self.and_bhv( );
30.            }
31.    }
32. }
```

Fig. 8 Rebeca model for *And worker*

4.2.3. Modeling LTL operator *Not*

The rebec shown in Fig. 9 models the behavior of the *worker* that acts as LTL operator ¬ . Also, this behavior can be seen in Fig. 5 (b). For the sake of brevity, we use the same structure of *Child* and *SMC* shown in Fig. 6 for the *Not* rebec. Therefore, this rebec also has a message server named *from_righChild*, while this message server has no effect on the behavior of this rebec because it only considers the result sent by *leftWorker* saving it in state variable *opr* at line 11. This state variable is used for verifying the model as well.

After receiving the message from its child, rebec *Not* processes it using method *not_bhv* at line 7. That is, it negates *opr* and sends it for its *Supervisor* (i.e. variable *parent*) on line 8.

```
1. reactiveclass Not (3) {
2.     knownrebecs {    Supervisor parent;   }
3.     statevars {      boolean opr;     }
4.     msgsrv initial ( ) {
5.         opr = false;
6.     }
7.     msgsrv not_bhv (boolean opr) {
8.         parent.result_fromOp (!opr);
9.     }
10.    msgsrv from_leftChild (boolean res){
11.        opr = res;
12.        self.not_bhv (opr);
13.    }
14.    msgsrv from_rightChild (boolean res){
15.        // There is no right child!
16.        // This method will never affect rebec Not
17.    }
18. }
```

Fig. 9 Rebeca model for *Not worker*

4.3. Verification results

To verify our model, we use model checking; in this section, properties that should be satisfied by models are specified in LTL. We have used the latest versions of model checkers RMC [36] and Spin [12] for verifying our models.[c]

In terms of LTL operator *U*, the safety property, which was verified and proved to be true in the model, is that when left operand remains *true* until the right operand becomes *true*, *Supervisor* should receive a *true* message from rebec *Until*, otherwise it should receive a *false*. In our method, as described above, the left operand is the same result sent by *leftWorker* saved in *updatedLeftOp*, and the right operand is the same result sent by *rightWorker* saved in *updatedRightOp*. This property is specified in LTL as follows:

$$G\ (\ (until.updatedLeftOp\ U\ until.updatedRightOp) \wedge\ parent.resultReceived) \rightarrow\ parent.result)$$
$$G\ (\neg\ (until.updatedLeftOp\ U\ until.updatedRightOp) \wedge\ parent.resultReceived) \rightarrow \neg\ parent.result)$$

The safety property that rebec *And* is expected to hold is that *Supervisor* receives a *true* from rebec *And* when both of its left operand and right operand are *true*, otherwise it should receive a *false*. This property was also verified and proved true. In our model, the left operand is the same result sent by *leftWorker*, and the right operand is one sent by *rightWorker*, which are saved in variables *leftOp* and *rightOp*, respectively. Therefore, the LTL specification of this property is as follows:

$$G\ (\ (and.leftOp \wedge and.rightOp) \wedge parent.resultReceived) \rightarrow\ parent.result)$$
$$G\ (\ \neg\ (and.leftOp \wedge and.rightOp) \wedge parent.resultReceived) \rightarrow \neg\ parent.result)$$

For LTL operator ¬, it is expected that *Supervisor* receives a true from rebec *Not* if the operand of operator ¬ is *false*. Obviously, in our method, the operand of a *not* operator is an actor. In the models, the value of this operand is the same result sent by *leftWorker* to rebec *Not*. The rebec

[c] Both of Rebeca and Promela models as well as the output of RMC and Spin have been attached to this paper.

saves this value into its state variable *opr*. Therefore, the LTL property should be held by rebec *Not* is as follows:

$$G ((\neg\ not.op \wedge\ parent.resultReceived) \rightarrow parent.result)$$
$$G ((not.op \wedge\ parent.resultReceived) \rightarrow \neg\ parent.result)$$

Table 1 shows the results of the verification of the forgoing properties by model checkers `Spin` [12] and `RMC` [36].

Table 1. Verification results

Property	**Spin Version 6.2.5**					
	Status	**Depth reached**	**Transitions**	**States**	**Time (sec)**	**Memory (MB)**
Until	Satisfied	272	3161	2045	0.2	64.636
And	Satisfied	109	2703	1797	0.2	64.636
Not	Satisfied	108	3041	1991	0.2	64.636
Property	**RMC Version 2.2.0**					
	Status	**Depth reached**	**Transitions**	**States**	**Time (sec)**	**Memory (MB)**
Until	Satisfied	21	504	224	0	8.17
And	Satisfied	15	384	164	0	5.92
Not	Satisfied	15	458	196	0	6.9

5. An illustrative example

This section describes a simple example of stateless model checking of an LTL property to illustrate the proposed method. This example is a version of the mutual exclusion problem with two threads. The pseudo code of the problem is shown in Fig. 10 (a). The safety property that program should satisfy is that two threads do not enter the critical section at the same time, which is specified in LTL as "G (¬ (*crit1* ∧ *crit2*))"; consequently, the *APs* used by the user in the LTL property are *crit1* and *crit2*. In this section, we describe the process of verification of this property step by step.

As we described in Section 2.1.3, the stateless model checker is expected to partition the program code according to visible operations. You can see the partitioned code of Fig. 10 (a) in Fig. 10 (b). Based on the rule of partitioning, each of *T1* and *T2* is divided into six locations. During stateless model checking, the model checker schedules threads based on these locations.

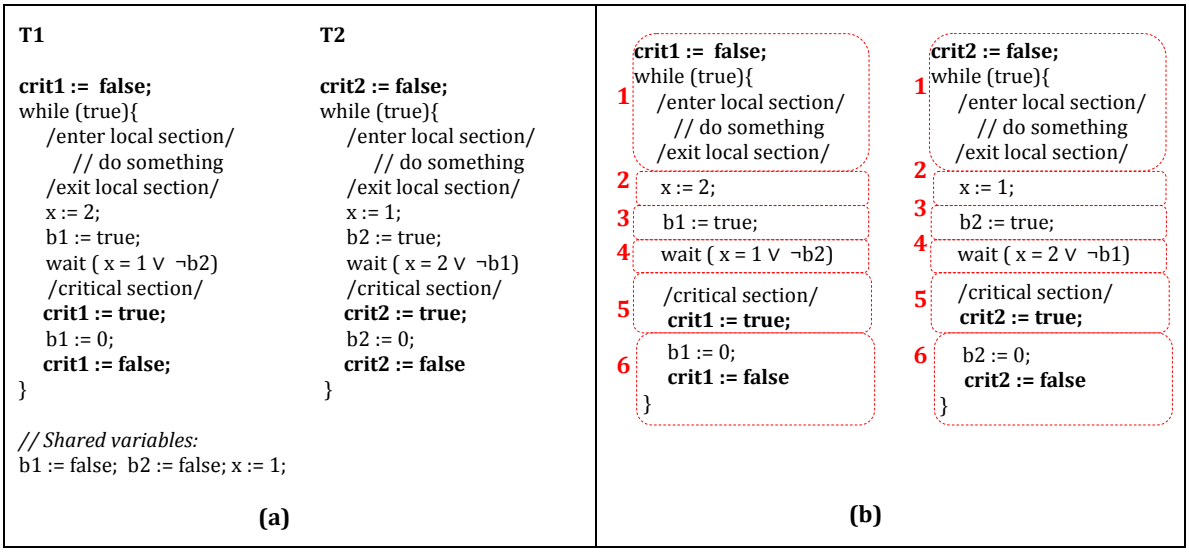

Fig. 10 An example of the mutual exclusion problem

It should be pointed out that at the end of each location (i.e. after a thread yields the CPU due to reaching the end of its current location), the model checker informs *condition checkers* about the current status of their *AP*. Hereafter, we suppose that stateless model checker has partitioned the program (i.e. Fig. 10 (b)) and is ready to start model checking.

Now, let us start explaining the verification process for this example. First of all, the *master* actor loads the defined property. Here, the user has defined only one property so *master* only creates one *property checker*, which is responsible for verifying the specified property. Now, *property checker* should create the hierarchy of *workers*. Frist, it standardizes the specified property as follows:

$$G\ (\neg\ (crit1 \wedge crit2)) = \neg F\ \neg\ (\neg\ (crit1 \wedge crit2)) = \neg\ (true\ U\ (crit1 \wedge crit2))$$

Next, *property checker* should parse this property and initiates creation of the hierarchy of *workers*. The parse tree for this property is shown in Fig. 11. After parsing, *property checker* creates a *worker* that behaves as a *not* operator, and sends the sub-tree of the *not* operator to this *worker*. This *worker* also parses the received sub-tree, creates an *until worker*, and sends the sub-tree under operator *until* to the created *worker*. The *until worker* also parses its sub-trees and creates a *condition checker* as its right child, which only generates *true*. For its left child, the *until worker* creates an *and worker* sending the reminder of the tree to this *worker*. After parsing the received sub-tree, the *and worker* creates two *condition checkers* that check *APs crit1* and *crit2*.

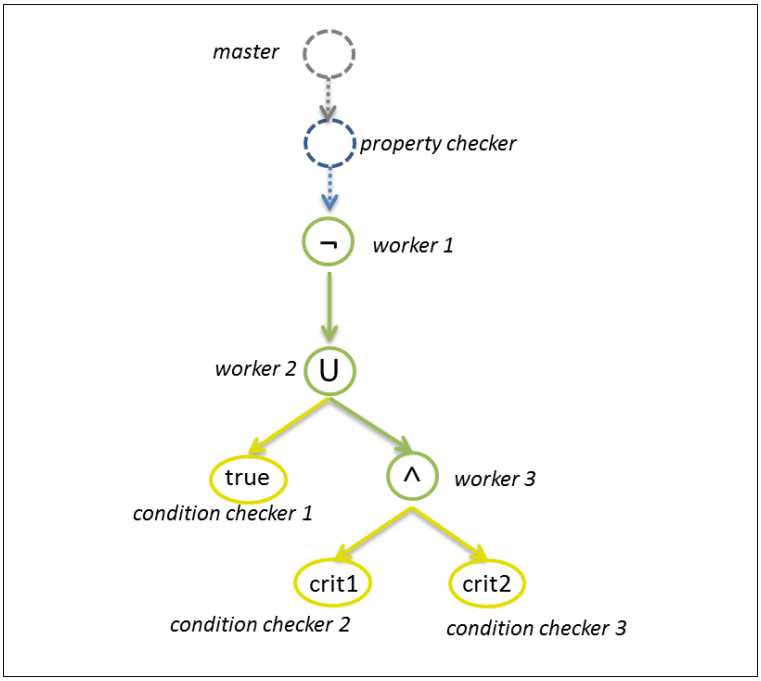

Fig. 11 The parse tree of property ¬ (*true* U (*crit1* ∧ *crit2*))

After creating the hierarchy, the model checker begins exploring the state space and verification. Suppose the model checker first schedules *T2*; therefore, *T2* performs its computation from the beginning of location 1 to the end of this location. At the end of this location, *T2* is preempted, and the stateless model checker sends the status of *APs* to the *condition checkers*. At this time, both of *crit1* and *crit2* are *false*, hence "*condition checkers 2*" and "*condition checkers 3*" send false to the "*worker 3*" (see Fig. 11). As "*worker 3*" is an *and* operator, it generates *false* because both of its operands are *false*. Therefore, "*worker 2*" receives a *false* message from its right *worker* and receives a *true* from its left *worker*. According to the behavior of an *until worker* (Fig. 7 and Fig. 5 (d)), the "*worker 2*" still waits for hearing from its children in the next status. For the sake of brevity, we summarize this process in Table 2. As you can see in Table 2, the model checker schedules threads as follows: *"s_1: T2, s_2: T2, s_3: T1, s_4: T1, s_5: T1, s_6: T1,* ***s_7: T1****, s_8: T2, s_9: T2, s_{10}: T2"*. The described situation recurs until s_7.

We go on explaining with s_7, where *T1* enters the critical section and *crit1* becomes *true*. As a result, at the end of this state, "*condition checker 2*" receives a *true* message and "*condition checker 3*" receives a *false*. Consequently, "*worker 3*" generates a *false* message so the reminder of the process is similar to what was described above. This situation recurs until the end of s_9, where *T2* also enters the critical section causing *crit2* to become *true*.

At the end of s_{10}, the model checker sends *true* to both "*condition checker 2*" and "*condition checker 3*", whereby "*worker 3*" also concludes a *true* result, and sends it to "*worker 2*". Therefore, "*worker 2*" receives *true* from both left *worker* and right *worker* so it also sends a *true* message to

"*worker 1*". As "*worker 1*" is a *not* operator, it negates the received result. Consequently, "*worker 1*" evaluates the result of verification as *false*. This result is sent to *property checker*. When *property checker* receives a *false*, it concludes that a violation has occurred.

Table 2. The verification process for the example shown in Fig. 10
Symbol "-"under some states denotes its corresponding actor sends no message in that state.

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}
The scheduled thread	-	T2	T2	T1	T1	T1	T1	T1	T2	T2	T2
Location number *T1 points to*	1	1	1	2	3	4	5	6	6	6	6
Location number *T2 points to*	1	2	3	3	3	3	3	3	4	5	6
x1	1	1	1	2	2	2	2	2	2	2	2
b1	False	False	False	False	True	True	True	True	True	True	True
b2	False	False	False	False	False	False	False	False	True	True	True
crit1	False	False	False	False	False	False	False	True	True	True	True
crit2	False	False	False	False	False	False	False	False	False	False	True
Message received by *"condition checker 2"*	-	False	False	False	False	False	False	True	True	True	True
Message received by *"condition checker 3"*	-	False	False	False	False	False	False	False	False	False	True
Message sent by *"worker 3"*	-	False	False	False	False	False	False	False	False	False	True
Message sent by *"worker 2"*	-	-	-	-	-	-	-	-	-	-	True
Message sent by *"worker 1"*	-	-	-	-	-	-	-	-	-	-	**False**

To briefly explain this example without complexity, we considered a non-terminating program. But, suppose that loop "*while (true)*" does not exist. Therefore, the program eventually comes to an end. In this case, the stateless model checker generates possible finite executions of the program. Suppose it generates different executions as follows:

i_{0,s_0} : *T1, T1, T2, T1, T1, T2, T2, T2, T1, T1, T2, T2, T2.*

i_{1,s_0} : *T1, T2, T2, T1, T2, T1, T1, T2, T2, T2, T1, T1, T1.*

…

i_{k,s_0} : *T2, T2, T1, T1, T1, T1, T1, T2, T2, T2.*

As a result, the property is independently checked in each iteration. According to Theorem 1, if the property is satisfied in all iterations, then the program satisfies the property. Obviously, if the property is violated in (at least) one iteration, for example in i_{k,s_0}, it means that the program does not satisfy the property.

6. Implementation issues

All actors of the verification hierarchy act in parallel with the stateless model checker. This hierarchy is very quickly formed at the beginning of stateless model checking. We are going to implement our method by Erlang programming language [33], in which processes (i.e. actors) are very lightweight and cheap to create (about 100 times lighter than threads) [51]. Message passing in Erlang is also very fast (about one micro second) [32, 51]. Therefore, there is no concern about the process creation and message-passing overhead. Erlang provides the best implementation of the Actor model [31], whereby we can precisely implement the proposed method.

7. Conclusions

This paper proposes a new verification method for stateless model checking of LTL properties. As far as we know, none of the existing stateless model checkers support checking LTL formulae because the existing LTL checking algorithms are state-based. For this reason, they are only applicable to stateful model checking techniques.

Our method is different from common LTL checking methods. The conventional algorithms are graph-based and need the program state space, while our method verifies formulae dynamically without storing any program states. The proposed method is designed based on the Actor model. Thanks to this model, we can create cheap and lightweight actors that check LTL properties simultaneously with stateless state space exploration.

The method proposed in this paper is designed as the unit of LTL checking for DSCMC [17, 18], which is a parallel stateless code model checker. We are implementing this method in DSCMC. This tool needs to analyze and instrument program code before performing stateless model checking. Currently, code is manually instrumented in DSCMC. Therefore, in the future, code instrumentation must be automated for using DSCMC in large programs. Once this has been done, we will be able to utilize the proposed method for real-world programs.

As for stateless model checking, it may be impractical to precisely handle non-deterministic user input. Then in practice, using the method to verify large programs may be transformed to *systematically* testing, but it is still powerful enough to explore the state space of large programs whose state space exploration is impractical using state-based methods [1, 7, 22, 23, 26, 28]. However, to cover more execution paths, the method can be improved by employing test generation techniques, such as white-box fuzz testing [52, 53] and symbolic execution [54].

References

[1] M. Musuvathi and S. Qadeer, Fair stateless model checking, *ACM SIGPLAN Notices - Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 43 (6) (2008) 362-371.

[2] E. M. Clarke and E. A. Emerson, Using branching time temporal logic to synthesize synchronization skeletons, *Science of Computer Programming*, 2 (3) (1982) 241-266.

[3] C. Baier and J. P. Katoen, *Principles of model checking*, MIT Press, 2008.

[4] D. Bošnački and S. Edelkamp, Model checking software: On some new waves and some evergreens, *International Journal on Software Tools for Technology Transfer (STTT)*, 12 (2) (2010) 89-95.

[5] T. Cristian and D. Nobelt, "FixD: Fault detection, bug reporting and recoverability for distributed applications," in *Parallel and Distributed Processing Symposium*, 2007, pp. 1-8.

[6] P. Godefroid, *Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem*, Springer-Verlag, 1996.

[7] P. Godefroid, Software model checking: the VeriSoft approach, *Formal Methods in System Design*, 26 (2) (2005) 77-101.

[8] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A pragmatic approach to model checking real code," in *5th Symposium on Operating Systems Design and Implementation*, 2002, pp. 75–88.

[9] G. Holzmann, The model checker SPIN, *IEEE Transactions on Software Engineering*, 23 (5) (1997) 279–295.

[10] P. Godefroid, "Model checking for programming languages using VeriSoft," in *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 174-186.

[11] Y. Yang, X. Chen, and G. Gopalakrishnan, "Inspect: A runtime model checker for multithreaded C programs," School of Computing, University of Utah, Technical Report UUCS-08-004, 2008.

[12] *Spin- Formal Verification*. Available: http://spinroot.com/

[13] *GMC- GIMPLE Model Checker*. Available: http://d3s.mff.cuni.cz/~sery/gmc/

[14] *MOPS*. Available: http://www.cs.berkeley.edu/~daw/mops/

[15] H. Chen and D. Wagner, "MOPS: An Infrastructure for Examining Security Properties of Software," in *9th ACM Conference on Computer and Communications Security*, 2002, pp. 235-244.

[16] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (SFCS '77)*, 1977, pp. 46-57.

[17] E. Ghassabani and M. Abdollahi Azgomi, "DSCMC: A distributed stateless code model checker," School of Computer Engineering , Iran University of Science and Technology, Technical Report IUSTCE-PDE-TR-12-09-DSMC, 2012.

[18] *DSCMC Web Page*. Available: http://pdel.iust.ac.ir/projects/dscmc.htm

[19] G. Agha, I. A. Mason, S. F. Smith, and C. L. Tolcott, A foundation for Actor computation, *Journal of Functional Programming*, 7 (1) (1997) 1-72.

[20] G. Agha, *Actors: A model of concurrent computation in distributed systems*, MIT Press, 1986.

[21] M. Sirjani, "Formal specification and verification of concurrent and reactive systems," PhD Thesis, Department of Computer Engineering, Sharif University of Technology, Theran, Iran, 2004.

[22] C. Flanagan and P. Godefroid, Dynamic partial-order reduction for model checking software, *ACM SIGPLAN Notices - Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 40 (1) (2005) 110-121.

[23] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang, "Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis," in *16th International SPIN Workshop on Model Checking Software*, 2009, pp. 279-295.

[24] R. Jhala and R. Majumdar, Software model checking, *ACM Computing Surveys*, 41 (4) (2009) 1-54.

[25] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Runtime model checking of multithreaded C/C++ programs," School of Computing, University of Utah, Technical Report UUCS-07-008, 2007.

[26] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, Distributed dynamic partial order reduction, *International Journal on Software Tools for Technology Transfer (STTT)*, 12 (2) (2010) 113-122.

[27] M. Musuvathi and S. Qadeer, Iterative context bounding for systematic testing of multithreaded programs, *ACM SIGPLAN Notices - Proceedings of the 2007 PLDI conference*, 42 (6) (2007) 446-455.

[28] M. Musuvathi, S. Qadeer, and T. Ball, "CHESS: A systematic testing tool for concurrent software," Microsoft Research, Technical Report MSR-TR-2007-149, 2007.

[29] *CHESS- Microsoft Research*. Available: http://research.microsoft.com/en-us/projects/chess/

[30] *Inspect*. Available: http://www.cs.utah.edu/~yuyang/inspect/

[31] R. K. Karmani and G. Agha, "Actors," in *Encyclopedia of Parallel Computing*, Springer US, 2011, pp. 1-11.

[32] J. Armstrong, *Programming Erlang: Software for a concurrent world*, Pragmatic Bookshel, 2007.

[33] *Erlang Reference Manual*. Available: http://www.erlang.org/doc/man/erlang.html

[34] *Rebeca Formal Modeling Language*. Available: http://www.rebeca-lang.org

[35] M. Sirjani, A. Movaghar, A. Shali, and F. S. Boer, Modeling and verification of reactive systems using Rebeca, *Fundamenta Informaticae*, 63 (4) (2004) 385-410.

[36] *Rebeca Model Checker (RMC)*. Available: http://www.rebeca-lang.org/wiki/pmwiki.php/Tools/RMC

[37] H. Hojjat, "Rebeca2," Rebeca Research Group, Reference Manual version 1.1.2, 2012.

[38] H. Sabouri and M. Sirjani, Actor-based slicing techniques for efficient reduction of Rebeca models, *Science of Computer Programming*, 75 (10) (2010) 811-827.

[39] *VeriSoft Home-Page*. Available: http://cm.bell-labs.com/who/god/verisoft/

[40] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Distributed dynamic partial order reduction based verification of threaded software," in *14th International SPIN Conference on Model Checking Software*, 2007, pp. 58-75.

[41] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *1st Symposium on Logic in Computer Science Cambridge (LICS '86)*, 1986, pp. 332-344.

[42] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," in *2nd International Workshop on Computer Aided Verification (CAV)*, 1990, pp. 233-242.

[43] S. Evangelista and L. M. Kristensen, "Hybrid on-the-fly LTL model checking with the sweep-line method," in *33rd International Conference on Application and Theory of Petri Nets (PETRI NETS'12)*, 2012, pp. 248-267.

[44] M. de Wulf, L. Doyen, N. Maquet, and J. F. Raskin, "Antichains: Alternative algorithms for LTL satisfiability and model-checking," in *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 63-77.

[45] F. Emmanuel, J. Nayiong, and J. F. Raskin, "Compositional algorithms for LTL synthesis," in *8th International Conference on Automated Technology for Verification and Analysis*, 2010, pp. 112-127.

[46] M. K. Ganai, C. Wang, and W. Li, "Efficient state space exploration: interleaving stateless and state-based model checking," in *International Conference on Computer-Aided Design (ICCAD '10)*, 2010, pp. 786-793.

[47] S. Christensen, L. M. Kristensen, and T. Mailund, "A sweep-line method for state space exploration," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2001, pp. 450-464.

[48] M. M. Chupilko and A. S. Kamkin, "Runtime verification based on executable models: On-the-fly matching of timed traces," in *8th Workshop on Model-Based Testing*, 2013, pp. 67-81.

[49] A. Bauer, M. Leucker, and C. Schallhart, Runtime verification for LTL and TLTL, *ACM Transactions on Software and Methodology (TOSEM)*, 20 (4) (2011) 1-64.

[50] A. Morgenstern, M. Gesell, and K. Schneider, "An asymptotically correct finite path semantics for LTL," in *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'12)*, 2012, pp. 304-319.

[51] S. Vinoski, Concurrency with Erlang, *IEEE Internet Computing Journal*, 11 (5) (2007) 90-93.

[52] P. Godefroid, "Random testing for security: Blackbox vs. whitebox fuzzing," in *2nd International Workshop on Random Testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 1-1.

[53] P. Godefroid, M. Y. Levin, and D. Molnar, SAGE: Whitebox fuzzing for security testing, *Magazine: ACM Queue - Networks*, 10 (1) (2012) 20-27.

[54] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, *et al.*, "Combining unitlevel level symbolic execution and system-level concrete execution for testing nasa software," in *International Symposium on Software Testing and Analysis (ISSTA '08)*, 2008, pp. 15-26.