TEMPORAL LOGIC RUNTIME VERIFICATION OF DYNAMIC SYSTEMS

Motlatsi Seotsanyana

Mobile Intelligent Autonomous Systems, Modelling and Digital Science,
Council for Scientific and Industrial Research, P.O. Box 395,
Pretoria 0001, SOUTH AFRICA.
Email: mseotsanyana@csir.co.za

ABSTRACT

Robotic computer systems are increasingly ubiquitous in everyday life and this has led to a need to develop safe and reliable systems. To ensure safety and reliability of these systems, the following three main verification techniques are usually considered: (1) theorem proving, (2) model checking, and (3) testing. However, the behaviour of robotic systems depends heavily on the environment and changes over time, which makes it hard to predict and analyse prior to their execution. Therefore, this paper provides a novel framework that automatically and verifiably monitors these systems at runtime. The main aim of the framework is to assist the operator through witnesses and counterexamples that are generated during the execution of the system. The framework is suitable for manufacturing and mine operations. In addition, the framework can explicitly capture sensor specifications of the environment and react to the changes accordingly to ensure safe and reliable operation during runtime of the system. The framework first constructs a region automaton of the environment and then represents operation rules in a formal specification language called universal computation tree logic (ACTL). This formal specification language allows the expression of complex desired behaviours, such as collision avoidance in the case of haul truck operations.

Keywords: runtime, verification, theorem proving, model checking, testing, temporal logic, automaton, observer-pattern.

1 INTRODUCTION

Robotic computer systems are increasingly ubiquitous in everyday life and this has led to a need to develop safe and reliable systems. To ensure safety and reliability of these systems, the following three main verification techniques are usually considered: (1) theorem proving, (2) model checking, and (3) testing. Theorem proving is the process of using deductive methods to develop computer programs that show that some statement (i.e., conjecture) is a logical consequence of a set of axioms and hypotheses. Unfortunately, theorem proving process is generally harder and requires considerable technical expertise and a deep understanding of the specification. It is also generally slower, more error-prone and labour intensive. Model checking, on the other hand, is an automatic verification technique for finite state concurrent systems such as safety critical systems, communication protocols, and sequential circuit design. Model checking is an attractive alternative to simulation and testing to validate and verify systems. But, model checking techniques are hindered by the state-space explosion problem, where the size of the representation of the behavior of a system grows exponentially with the size of the systems. Often, software systems have infinite state spaces, due to unbounded real and integer input variables and timing constraints, and thus model checking software systems without any abstractions is almost always impossible. The current practice is that the correctness of computer systems is achieved by human inspection: peer reviews and testing with little or no automation. Peer reviews refer to the inspection of software by a team of engineers that were preferably not involved in the design of the system, while testing refers to a process whereby software is executed with some inputs, called test cases, along different execution paths known as runs. However, testing is never complete: it is difficult to say when to stop as it is infeasible to check all the runs of a complex system and it is easy to omit those runs which may reveal subtle errors. It also has a drawback of showing the presence of errors, but not their absence.

All the aforementioned verification techniques often do not scale up well to large systems such as robotic systems, which depends heavily on the environment and changes over time. This behaviour makes it hard to predict and verify these dynamic systems prior to their execution. Therefore, runtime verification is an appropriate technique to complement these techniques. Leucker and Schallhart [1] define runtime verification as the "*discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property*". Although not new, runtime verification is receiving increasing interest due to the advent of new verification techniques such as model checking techniques.

In this paper, we provide a novel framework that automatically and verifiably monitors robotic systems at runtime. The runtime verification monitor is implemented as a lightweight model checking algorithm. The main aim of the framework is to assist the operator through witness and counterexample sets that are generated during the execution of the system. Witness is a set of states of a system that satisfy a formal property specification, while counterexample is a set of states of a system that violates a property. The framework is suitable for manufacturing, and mine operations and it is based on observer design pattern. In addition, the framework can explicitly capture sensor specifications of the environment and react to the changes accordingly to ensure safe and reliable system operation at runtime. The framework requires the construction of a region automaton for the environment and then representation of operation rules in a formal specification language called universal computation tree logic (ACTL). This formal specification language allows the expression of complex desired behaviours, such as collision avoidance in the case of haul truck operations.

The rest of the paper is organized as follows. Section 2 outlines part of the related work. Section 3 describes the general problem of the dynamic environment that is augmented with sensors. Section 4 presents an overview of temporal logic and describes the syntax and semantics of computation tree logic -and another class of the logic called universal computation tree logic. Section 5 describes the runtime monitor; the focus is on its inputs (the environment and formal specification properties) and outputs (the witness and counterexample sets) at runtime. Section 6 illustrates the application of the framework with a case study, while Section 7 concludes the paper. Section 8 highlights the possible extension of the framework.

2 RELATED WORK

There is currently an increasing amount of work being done on runtime verification. In this paper we only present a part of this work. We discuss only work done in [6, 7, 8, 9, 10] since all these papers use formal methods techniques.

Monitoring and Checking (MaC) [6] provides a general framework that makes sure that the target program runs correctly with respect to a formal requirement specification. The framework consists of two specification languages: Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL). The former is used to define methods and objects to be monitored; a filter keeps a list of monitored local and global variables, as well as, addresses of monitored objects. The later is used to write high specification requirement. The main reason for using two specification languages is to separate the implementation details from high-level requirement checking and thus makes the framework portable to different programming languages and specification formalisms.

Monitoring –Oriented Programming (MoP) [7] is a framework and methodology for building program monitors. It allows formal property specification to be added to the target program and does not place any restriction on a formalism to be used, as long as the corresponding translator of the specification language exists. The translated code must contain the following components: declaration, initialization, monitoring body, success condition, and failure condition. The user puts annotations in the target program at which the monitoring code must be inserted. Currently, the MoP supports three specification languages: past time and future time linear temporal logic as well as extended regular expressions.

Java with assertion (Jass) [8] is a general monitoring methodology implemented for sequential, concurrent and reactive systems written in java. The tool Jass is a pre-compiler that translates annotations into a pure java code in which a compliance with specification is tested dynamically at runtime. Assertions extends the Design by Contract [11], that allows specification of assertions in the form of pre- and post-conditions, class invariants, loop invariants, and additional check to be inserted in any part of the program code. Jass also offers *refinement check* and *trace assertion*. Refinement check is used to facilitate specification of classes on different levels of abstraction, while trace assertion are used to monitor the correct behaviour of method invocations, ordering and timing of methods invocation.

Java PathExplorer (JPaX) [9] is a general purpose monitoring approach for sequential and concurrent programs developed in Java. The tool offers two main facilities: logic-based monitoring and error pattern analysis. Formal specifications are written in linear temporal logic (both past and future) or in Maude [12, 13]. JPaX instruments Java byte code to send stream of events to the observer that performs two functions: it checks events against a high-level specification (logic-based monitoring) and also checks low-level programming error (error pattern analysis).

Temporal Rover [10] is a specification based commercial tool for programs written in C, C++, Java, Verilog and VHDL. In Temporal Rover, the user annotates the sections of the target program where a property needs to be checked at runtime. Temporal Rover supports the following formalisms: Linear–Time Temporal Logic (LTL) and Metric Temporal Logic (MTL) and the properties which can be specified with these logics include: future time temporal properties as well as lower and upper bound properties, and relative-time and real-time properties. The tool takes the target program as its input and its parser generates an identical program to the properties inserted in the target program and during execution the generated code validates the executing program against specified properties.

3 PROBLEM FORMULATION

The goal of this paper is to present a framework that enables the verification of dynamic robotic systems at runtime. We have developed a reactive monitor that takes an automaton of the environment and formal specifications of operation rules as inputs and synthesis warnings, alerts or errors that might lead to system failures. The synthesis is based on the witness and counterexample sets. The problem that we are out to solve is depicted by Figure 1. In this section the *environment* and *operation rules* as well as the *design pattern* employed are described in detail.

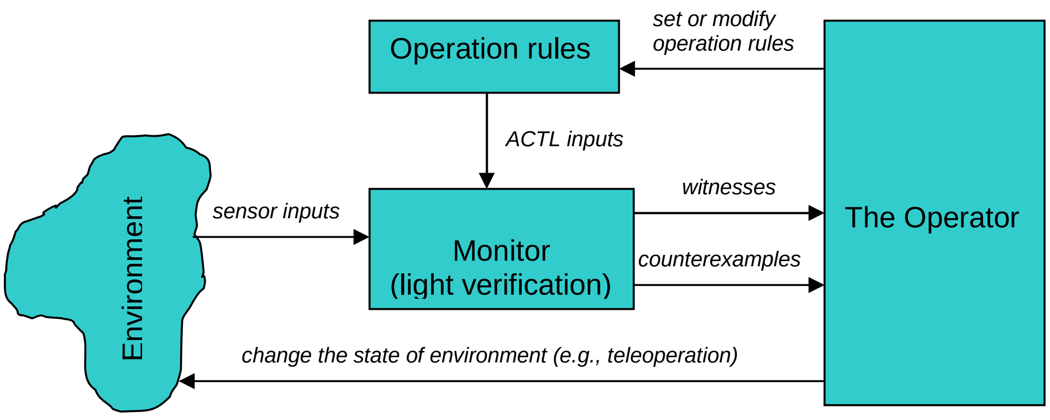
Operation rules
set or modify operation rules
ACTL inputs
Environment
sensor inputs
Monitor (light verification)
witnesses
counterexamples
The Operator
change the state of environment (e.g., teleoperation)

Figure 1: An overview for framework

Environment: To achieve our goal, we assume that the environment is partitioned into a finite number of regions of interest $\pi_1,\ldots,\pi_n$, where $\Pi = \cup_i^n \pi_i$ and $\pi_i \cap \pi_j = \phi$ if $i \neq j$. There are a number of objects $o_i^j,\ldots,o_n^m$ moving in the environment, where n denotes the number of regions and m the number of objects in the environment. These objects maybe machines (e.g., haul trucks), people (e.g., workers), etc., and they form part of the environment to one another (i.e., one object becomes part of environment of another object). The partition creates Boolean propositions $\gamma = \{o_1^1, o_2^1, \ldots o_n^m\}$ which are true if the objects are located in π_i, for example o_1^1 and o_1^2 are true if and only if $\pi_1 = \{o_1^1, o_1^2\}$. The objects interact with their environment through sensors, which are assumed to be binary. The m binary sensors $\xi = \{x_1^1,\ldots,x_n^m\}$ in n regions have their own dynamics which are not explicitly modeled in this paper. The possible behaviour of these variables will be captured with a suitable temporal logic formulas presented in Section 3. Finally, the environment is formally defined as an automaton $A = (\xi, \gamma, S, S_0, \delta, L)$ where:

- ξ is a set of binary sensor inputs
- γ is a set of Boolean propositions of objects interacting in the environment

- Π is a set of partitions of interests that form the environment
- $S \subset \mathbb{N}$ is a set of states
- $S_0 \subset S$ is a set of initial states
- $\delta : S \times 2^{\xi} \to 2^{S}$ is the transition relation, i.e. $\delta(s, X) = S' \subseteq S$ where $s \in S$ is a state and $X \subseteq \xi$ is the subset of sensor propositions that are true
- $L : S \to 2^{\gamma}$ is a state labelling function where $L(s) = y$ and $y \in 2^{\gamma}$ is the set of propositions that are true in state $s \in S$.

Operation rules: These are rules that describe the desired behaviour of objects interacting within the environment. In our framework these rules will be formally expressed in a suitable universal computation tree logic (ACTL) [2, 3] presented in Section 3. Informally speaking, ACTL will be used to specify a variety of operation rules grouped in the following categories: avoidance, coverage, sequencing, and conditions. Avoidance refers to the use of sensors to avoid colliding with other objects, coverage refers to those rules that specify regions of interest to traverse, sequencing refers to the traversal of specific regions in a certain order, and condition refers to those logical conditions that express a function from truth values to truth values.

Design pattern: The monitor module detailed in Section 4 is reactive to changes caused by both the environment and operation rules. That is, the monitor looks after the dynamic environment and operation rules which maybe modified either predictably or unpredictably, and also oversees the distributed interactions of objects in the environment. An appropriate design pattern in this kind of setup is the observer design pattern, also known as Publish-Subscribe or Dependents. The observer design pattern defines a one-to-many dependency between interacting objects so that when one object (the subject) changes state, all its dependents (the observers) are notified and updated automatically. Although not new, the pattern is receiving increasing interest because of its usefulness in event-driven systems. It encompasses a well-established communications paradigm that allows any number of subjects (publishers) to communicate with any number of observers (subscribers) asynchronously and anonymously via event channels. In our case, the monitor is implemented as an observer while the environment and operation rules are implemented as subjects.

4 TEMPORAL LOGICS

Temporal logics are special types of modal logic that investigate the notion of time and order of execution paths in computer systems. These logics have been used to precisely describe the properties of concurrent systems (such as *safety* and *liveness* properties) and were first introduced by Pnueli around 1977 for the specification and verification of computer systems. In the early eighties Clarke and Emerson introduced another type of temporal logic called Computation Tree Logic (CTL) [3, 4]. This temporal logic (i.e., CTL) together with linear temporal logic (LTL) [4] are the mostly widely-used temporal logics in the formal methods community. In this paper, we use a fragment of CTL called universal computation tree logic (ACTL) [2] and the following subsections present the syntax and semantic of CTL as well as formally describing the ACTL.

4.1 Computation tree logic

The CTL is based on the concept that for each state there are many possible successors, unlike in LTL which is based on a model where each state s has only one successor state s'. Because of this branching notion of time, CTL is classified as a *branching temporal logic*. The interpretation of CTL is therefore based on a *tree* rather than a *sequence* as in LTL.

CTL Syntax: The formulas of CTL consist of atomic propositions, standard Boolean connectives of propositional logic, and temporal operators. Each temporal operator is composed of two parts, a path quantifier (universal $\forall$ or existential $\exists$) followed by a temporal modality (G, F, X, U). Note that some authors use □ and ◊ for G and F, respectively. The syntax is given by the BNF:

$$\alpha ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \exists X\alpha \mid \forall X\alpha \mid \exists[\alpha U\beta] \mid \forall[\alpha U\beta]$$

The operators $\wedge, \Rightarrow, \Leftrightarrow, true, false, F, G$ which are not mentioned in this syntax can be thought of merely as abbreviations by using the following rules:

$$\begin{array}{ll} \alpha \wedge \beta \equiv \neg(\neg\alpha \vee \neg\beta) & \alpha \Rightarrow \beta \equiv (\neg\alpha \vee \beta) \\ \alpha \Leftrightarrow \beta \equiv (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha) & true \equiv (\neg\alpha \vee \alpha) \\ false \equiv \neg true & F\alpha \equiv true\ U\ \alpha \\ G\alpha \equiv \neg F \neg\alpha & \end{array}$$

CTL Semantics: CTL semantics slightly differs from that one of LTL, that is, the notion of a sequence is replaced by a notion of a tree. The interpretation of CTL is defined by a satisfaction relation $\models$ between a model M, one of its states s and some formula. Let $AP = \{p, q, r\}$ be a set of atomic propositions, $M = (S, R, Label)$ be CTL-Model, $s \in S$ α and β be CTL-formulas. In order to define the satisfaction relation ($\models$), the following definitions are first given:

- A *path* is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$
- Let $\rho \in S^w$ denotes a path. For $i \geq 0, \rho[i]$ denotes the $(i+1)^{th}$ element of ρ, i.e., if $\rho = s_0, s_1, s_2, \ldots$ then $\rho[i] = s_i$
- $P_M(s) = \{\rho \in S^w \mid \rho[0] = s\}$ is a set of paths starting at s.

The satisfaction relation ($\models$) is then mathematically defined as follows:

$$\begin{array}{lll} s \models p & \text{iff} & p \in Label(s) \\ s \models \neg\alpha & \text{iff} & \neg(s \models \alpha) \\ s \models \alpha \vee \beta & \text{iff} & (s \models \alpha) \vee (s \models \beta) \\ s \models \exists X\alpha & \text{iff} & \exists\rho \in P(s) : \rho[1] \models \alpha \\ s \models \forall X\alpha & \text{iff} & \forall\rho \in P(s) : \rho[1] \models \alpha \\ s \models \exists[\alpha U\beta] & \text{iff} & \exists\rho \in P(s) : \exists j \geq 0 : (\rho[j] \models \\ & & \beta \wedge \forall 0 \leq k < j : \rho[k] \models \alpha) \\ s \models \forall[\alpha U\beta] & \text{iff} & \forall\rho \in P(s) : \exists j \geq 0 : (\rho[j] \models \\ & & \beta \wedge \forall 0 \leq k < j : \rho[k] \models \alpha) \end{array}$$

4.2 Universal computation tree logic

In this paper, we consider a fragment of CTL called universal computation tree logic (ACTL). The ACTL allows only universal assertions of CTL [2]. The syntax of an ACTL formula α is given by:

$$\alpha ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha \wedge \beta \mid \forall X\alpha \mid \forall F\alpha \mid \forall G\alpha \mid \forall[\alpha U\beta]$$

For each ACTL formula, there exists an equivalent existential normal form (ENF). Since our algorithm outlined in Section 4 uses ENF, The syntactic abbreviations for ACTL formulas are provided in Table 1:

$\forall X\alpha \quad \equiv \neg\exists X\alpha$	this means that α holds at all successor states of the current state.
$\forall F\alpha \quad \equiv \neg\exists G\neg\alpha$	this mean that for every path there exists a state on the *path* at which α holds.
$\forall G\alpha \quad \equiv \neg\exists F\neg\alpha$	this means that for every *path* , α holds in each state on the *path.*
$\forall[\alpha U\beta] \equiv \neg\exists[\neg\beta U(\neg\alpha \wedge \neg\beta)] \wedge \exists G\neg\beta$	this means that for every path, there exists an initial prefix of the path such that β holds at the last state of the prefix and α holds at all other states of the prefix.

Table 1: The syntactic abbreviation for ACTL formulas

5 THE MONITOR

In [1], the monitor is defined as a device that reads a finite trace of a run and outputs a verdict. In [5] a verdict is a true value from some domain that ranges over *true*, *false*, and *inconclusive*. The true value *true* means that the specification is satisfied, *false* means that the specification is not satisfied while *inconclusive* means that a conclusion cannot be reached with the current trace. However, our approach is not the same as followed by other authors [6, 7, 8, 9, 10], that is our approach do not verify finite prefixes of a run instead the monitor reacts every time the environment or operation rules change and outputs warning messages to the operator based on the set of witnesses and counterexamples emitted (as explained in Section 4.1). The monitor is formally defined as follows:

$$monitor : A \times f_A \rightarrow \{[[w]]_A, [[c]]_A\}, \text{ where:}$$

- A is an automaton that represents the environment,
- f_A is a set of ACTL formulas that defines operation rules,
- $[[w]]_A = \{s \mid s \models \alpha\}$ denotes the set of states for which the ACTL formula α is *true*, and
- $[[c]]_A = \{s \mid s \models \neg\alpha\}$ denotes the set of states for which the ACTL formula α is *false*.

The computation of these sets (i.e., the witness and counterexample sets) has its roots in a fixed point theory [4], that is, the set can be computed using fixed point iterations that are guaranteed to terminate. The following rules are applied to determine these states. Let φ and ψ be ACTL formulas and p be atomic proposition (i.e., $p \in AP$), then:

$$
\begin{aligned}
[[p]]_A &= \{s \in S \mid p \in Label(s)\} \\
[[\neg\varphi]]_A &= S \setminus [[\varphi]]_A \\
[[\varphi \vee \psi]]_A &= [[\varphi]]_A \cup [[\psi]]_A \\
[[\varphi \wedge \psi]]_A &= [[\varphi]]_A \cap [[\psi]]_A \\
[[AG\,\varphi]]_A &= [[\varphi]]_A \cap \{s \in S \mid \forall s' \in R(s) \cap F(Z)\} \\
[[AF\,\varphi]]_A &= [[\varphi]]_A \cup \{s \in S \mid \forall s' \in R(s) \cap F(Z)\}
\end{aligned}
$$

Where $R(s)$ is the successor state of $s \in S$ and Z is the power set of S (i.e., 2^S) and $F(Z)$ is a recursive function that terminates when either a greatest fix-point or a least fix-point is reached.

5.1 Counterexamples and Witnesses

In model checking, a *counterexample* is a set of states that violates an ACTL formula α (i.e., $[[c]]_A = \{s \mid s \models \neg\alpha\}$), whereas a *witness* is a set of states in which the ACTL formula α holds (i.e., $[[w]]_A = \{s \mid s \models \alpha\}$). We provide an example to illustrate the difference between these two sets. Let $p, q, r \in AP$, $S = \{0,1,2,3,4,5\}$, $S_0 = \{0\}$, and $\forall G\,p$ be the ACTL specification. Therefore: $[[w]]_A = \{1,2,3\}$ and $[[c]]_A = \{0,4,5\}$.

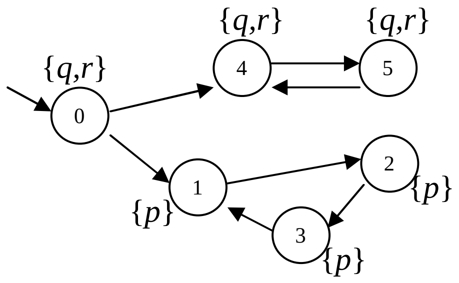

Figure 2: A Kripke structure of the environment

The computation of these sets automatically is a major advantage of model checking and with proper annotations of the automaton that models the environment, these sets provide an operator with valuable information that can lead to the avoidance of catastrophic system failures. For example, the operator can see that the environment modeled by Figure 2 does not satisfy the ACTL formula because the initial state S_0 is not in the witness set, which is the requirement in model checking that the initial state must be the element of a witness set in order for the specification to hold. If the requirement must hold, then the operator can modify the environment so that the specification holds. In this case, the operator can label the initial state S_0 with atomic proposition p (if this change does not violate other operation rules).

5.2 Light model checking

The model checking problem is defined as follows: Given a Kripke structure (or an automaton) A that represents a finite state system[1] and a temporal ACTL formula α expressing some desired specification, compute the set of all states in S that satisfy α. This is mathematically expressed as: $\{s \in S \mid A, s \models \alpha\}$. The system satisfies the specification provided all the initial states are in the computed set of states. Algorithm 1 outlines a modified version of a model checking algorithm. For the original model checking algorithm, the reader is referred to [4].

Algorithm 1 (A modified ACTL model checking)

```
1. Input: automaton $A$ and ACTL formula $\alpha$
2. Output: $[[w]]_A \cup [[c]]_A$
3. for $i \leq |\alpha|$ do
4.   for all $\beta \in Subformula(\alpha)$ with $|\beta| = i$ do
5.             compute $[[w]]_A$ from $S / [[w]]_A$
6.   od
7. od
8.   return $[[w]]_A \cup S / [[w]]_A$
```

The algorithm operates in a number of steps. The first step processes all sub-formula of length 1, the second step processes all sub-formulas of length 2, and so on. At the end of i^{th} step all the states will be labeled with sub-formulas of length equal to or less than *i* that are true in that state. Finally, the algorithm returns two sets of states that contain states where the formula of interest holds (i.e., witness set) and also where it does not hold (i.e., counterexample set).

6 CASE STUDY

Mining areas are vulnerable to a number of hazards including: flooding, roof fall, and environmental factors that prevent safe human access. These hazards have necessitated the use of robots to enter, explore, and map these areas. For a better understanding of our framework, we present a bi-dimensional and easy to visualize mining environment depicted in Figure 2(a). The specification that we want to verify at runtime is the reachability of a UAV to region 3 followed by surveillance task of regions 4 and 5. That is, we assume that the UAV will eventually reach region 3 and continuously remote video regions 4 and 5. The corresponding specification is formally defined as follows:

$$\forall F(x_3^1 \wedge \forall G \forall F(x_2^1 \vee x_3^1))$$

where x_2^1 and x_3^1 are sensors augmented in regions π_2 and π_3, respectively.

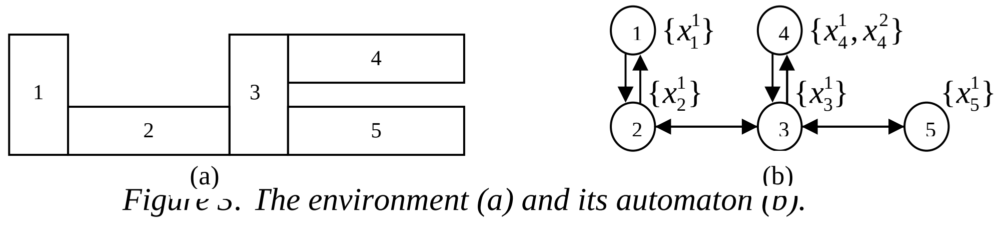

Figure 3. The environment (a) and its automaton (b).

[1] In our case, this system represents the environment.

The specification is runtime verified with the automaton representing the environment depicted in Figure 3(b) and it took approximately 0.41 seconds for our framework to output the two sets: $[[w]]_A = \{3,4,5\}$ and $[[c]]_A = \{1,2\}$. This means that if we compute a set of paths (i.e., runs) using only states in $[[w]]_A$, the specification will always be satisfied (i.e., starting from any of these states the specification will eventually always hold). But if we compute the set of paths with the states including state 1 or 2 (e.g., {2, 3, 4, 5}), the states 4 and 5 might not infinitely often be visited as there is a cycle of execution between states 2 and 3. It is clear that the execution of the system might stay in that cycle forever and this situation violates our specification. These two sets play an important role in assisting the operator to notice errors that might lead to catastrophic system failures.

7 CONCLUSION

In this paper, we have presented a framework that is appropriate for robotic systems that are reactive to changes made in both the operation rules and the dynamic environment. Motivated by the difficulty of theorem proving, model checking and testing these systems prior to their execution due to their heavily dependence on the dynamic environment and changes over time, our framework is based on the observer design pattern. The observer design pattern is useful in event-driven systems since it encompasses a well-established communications paradigm that allows any number of subjects (publishers) to communicate with any number of observers (subscribers) asynchronously and anonymously via event channels. At runtime the framework takes operation rules written in a temporal logic called universal computation tree logic, and an automaton representing the environment and outputs two set of states: *witness* and *counterexample*. Witness is a set of states that satisfy the operation rules while counterexample is a set that contains states that do not satisfy the operation rules. The case study and the results have shown that the framework is a promising platform for runtime verification of robotic systems to ensure safe and reliable functioning of these systems.

8 RECOMMENDATIONS

The presence of environmental uncertainty, while at the same time trying to meet high-level expectations of autonomous operation, is the main challenge in robotics. This challenge necessitates the use of stochastic modeling, discrete and continuous dynamics, quantitative and qualitative measures, and goal-oriented approaches of which the current software verification tools are unable to address. Therefore, we intend to expand our framework to include the runtime verification of discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs). We believe that the runtime verification of these processes will ensure the safety and reliability of systems that rely on the dynamic environment.

9 ACKNOWLEDGEMENTS

The research conducted and reported on in this paper was funded by the Council for Scientific and Industrial Research (CSIR), South Africa. The authors would like to thank Professor Jitendra Raol and Fred Senekal for their advice and comments.

10 REFERENCES

[1] Leucker, M., Schallhart, C., A brief account of runtime veri_cation. *Journal of Logic and Algebraic Programming*, 78(5):293 - 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract- Oriented Software (FLACOS'07).

[2] Clarke, E.M., Grumberg, O., Peled, D.A., Model checking, MIT Press, Cambridge, MA, 1999.

[3] Clarke E.M, Emerson, E.A, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*. London, UK: Springer-Verlag, 1982, vol 131 pp. 52–71.

[4] Christel Baier, Joost-Pieter Katoen, Principles of Model Checking, MIT Press, 55 Hayward Street, Cambridge, MA, 2008.

[5] Bauer, A., Leucker, M., Schallhart, C., "The good, the bad, and the ugly, but how ugly is ugly?" In *Workshop on Runtime Verification (RV'07)*, pages 126-138, 2007

[6] Kim, M., Kannan, S., Lee, I., Sokolsky, O., and M. Viswanathan, "Java- MaC: Run-time Assurance Tool for Java Programs "*Proc. Of the Fourth IEEE Int'l High Assurance Systems Eng. Symposium*", pp. 115-132, 1999.

[7] Chen, F., Roşu, G., "Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation" Electronic Notes in Theoretical Computer Science, vol. 89, no. 2, Elsevier, 2003.

[8] Bartetzko, D., "Jass - Java with Assertions," *Proc. of the First Workshop Runtime Verification (RV'01)*, Paris, France, Jul. 2001, K. Havelund and G. Roşu, eds., Electronic Notes in Theoretical Computer Science, Elsevier Science vol. 55, no. 2, 2001.

[9] Havelund, K., Roşu, G., "Java PathExplorer – A Runtime Verification Tool," Symposium on Artificial Intelligence, Robotics and Automation in Space, Montreal, Canada, June 2001

[10] Drusinsky, D., "The Temporal Rover and the ATG Rover," *SPIN 2000*, Springer-Verlag 2000.

[11] Meyer, B., "Applying Design by Contract," *IEEE Computer*, vol. 25, no. 10, pp. 40-51, 1992

[12] Clavel, M., Eker, S., Lincoln, P., Meseguer, J., *Principles of Maude*, Electronic Notes in Theoretical Computer Science, vol. 4, 1996.

[13] Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Quesada J., *Using Maude,* Proc. of the Third Int'l Conf. on Fundamental Approaches to SE, Lecture Notes in CS 1783, pp. 371-374, 2000.