2014 21st Asia-Pacific Software Engineering Conference

Runtime Verification by Convergent Formula Progression

Yan Shen*, Jianwen Li*, Zheng Wang†, Ting Su*, Bin Fang*, Geguang Pu*, Wanwei Liu‡ and Mingsong Chen*
*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
†Beijing Institute of Control Engineering, Beijing, China
‡National University of Defense Technology, Changsha, China

***Abstract*—Runtime verification is a dynamic verification technique widely used in practice. In this paper we revisit the runtime verification technique with formula progression, which verifies the execution trace step by step by progressing the desired property written in temporal logic. The previous work did not discuss explicitly the bound for the sizes of expanded formulas, while the successive invoking of formula progression is likely to cause divergence. In this paper[1], we present the convergent formula progression by introducing a novel fix-point reduction technique, and prove it guarantees the sizes of expanded formulas be always convergent. To the best of our knowledge, this is the first work discussing the convergence of formula progression. Furthermore, we implement the new runtime verification framework, and experiments show the efficiency of our proposed strategy.**

I. Introduction

Recently *program verification* has become a popular research area. Informally, this topic answers the question that given a system model M and a specification ϕ then how to prove that M satisfies ϕ. As safety-critical systems are strongly required nowadays, the achievements of program verification have been widely used in industry, cf. [1]. Generally speaking, the verification can be either static or dynamic. Static verification methods include model checking [2], theorem proving [3], and dynamic verification ones include runtime verification [4] and etc.

Due to the state-explosion problem of model checking [2] and the possible manual intervention in theorem proving [3], the light-weight formal technique of program verification, *runtime verification* becomes attractive [5]. Runtime verification is a complementary technique to exhaustive verification, where a *monitor* is generated from the specification ϕ, and inspects the execution of the program at run time to *check* whether the program meets the specification ϕ. There are two main strategies for this checking: One is the on-line manner in which the monitoring occurs during program execution, while the other one is off-line such that the monitoring occurs after program execution, with the traces stored in a log file. Here we can see that the monitor plays a key role in both approaches.

The monitor in runtime verification is often considered as a deterministic finite automata (DFA), for the reason that the verification aims to detect the violation in the (finite) prefix of an execution trace [6]. Therefore, runtime verification framework is normally available for safety specifications (properties) only, in which case the DFA can be generated [7]. However, lots of researches have shown that the DFA construction from specifications, such as LTL properties, can involve a doubly exponential grow-up, which in practice is not efficient [7], [8]. Thus some efforts are paid in constructing the minimal DFA such that the checking cost can be reduced as much as possible [8]. This optimization speeds up the performance indeed, while the whole automaton still needs to be generated.

Another improvement is to construct the DFA dynamically, i.e., DFA are constructed from properties under the guidance of the input execution traces. Such techniques are called *formula-rewriting* [9] or *formula progression* [10].(Although proposed independently, they actually follow the same formula expansion technique). In this paper, we fix the notation by *formula progression* and consider its application in linear temporal logic (LTL).

The formula progression technique takes an LTL formula ϕ and a current assignment P as inputs, and returns the next formula after P acting on ϕ. The technique was originally available for co-safety formulas, whose satisfaction can be determined in finite steps [10].Using this technique, the monitor generation is guided by the execution trace and thus saves time and space. There are a few research work that integrate formula progression into the runtime verification field. For instance, [5] utilizes the formula-progression-based technique to verify the Android operating system. Also, a formula-rewriting-based technique is established for the logic $FLTL_4$.

Theoretically, invoking the formula progression one at a time has only a linear complexity with respect to the input formula ϕ. However, the algorithm is normally invoked successively in runtime verification by feeding the status of the execution trace step-by-step, in which case the cost may grow up unlimitedly if the expanded formulas do not arrive in the fix-point states. Actually, the pure formula progression technique in general is divergent, which means the sizes of the generated formulas grow up unlimitedly. Although earlier work [5] introduced formula progression for runtime verification framework, they did not discuss how to ensure the convergence of the generated formulas by formula progression technique. Evidently, the divergence

[1]Geguang Pu is the corresponding author.

1530-1362/14 $31.00 © 2014 IEEE
DOI 10.1109/APSEC.2014.47

of formula progression may jeopardize the effective of this technique for runtime verification.

In this paper, we revisit the formula progression for LTL in runtime verification, and propose the *convergent formula progression* by integrating the novel *fix-point reduction* technique. We prove that by convergent formula progression the sizes of generated formulas are restricted under the bound of $2^{|\phi|}$, where $|\phi|$ means the size of input formula ϕ.

The main contributions in this paper are listed as below: 1). We propose the *convergent formula progression* by integrating the novel fix-point reduction technique introduced, which ensures the sizes of generated formulas are bounded; 2). We implement a new runtime verification framework with convergent formula progression; 3). A real case study from industry is carried on and the experiments show the efficiency and effectiveness of our method.

This paper is organized as follows. Section II introduces LTL and the formula progression algorithm in runtime verification; Section III presents the convergent formula progression with the fix-point reduction technique; Section IV introduces a detailed runtime verification framework with convergent formula progression; Section V shows the experimental results of a case study; And Section VI concludes the paper. All missing proofs in this paper can be found in the online technique report: http://www.lab205.org/home/pages/lijianwen/data/rv.pdf.

II. Preliminaries

A. Linear Temporal Logic

The Linear Temporal Logic (LTL) has been widely used in the program verification area since it was first introduced into computer science in 1977. Given an atomic set $\mathcal{P}$, we can inductively define the LTL formulas over $\mathcal{P}$ as follows:

$$\phi ::= \top \mid \bot \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X\phi \mid \phi_1 U \phi_2 \mid \phi_1 R \phi_2$$

where $a \in \mathcal{P}$, and $\top$, $\bot$ stand for *True*, *False*, respectively.

LTL is considered to be an extension of propositional logic, which inherits all unary and binary operators from propositional logic. Beyond that, LTL also introduces three temporal operators: Next (X), Until (U), and Release (R). U and R are duals of each other, i.e., it holds that $\neg(\phi_1 U \phi_2) \equiv \neg\phi_1 R \neg\phi_2$ and $\neg(\phi_1 R \phi_2) \equiv \neg\phi_1 U \neg\phi_2$. Especially, we also use the abbreviation $F\phi$ to represent $\top U \phi$, and $G\phi$ to $\bot R \phi$.

The formula ϕ is called a literal iff ϕ is an atom a or its negation $\neg a$. We use L_ϕ to denote the set of literals in ϕ, i.e., $L_\phi = \mathcal{P} \cup \{\neg a \mid a \in \mathcal{P}\}$ (We notate it L directly in the following). In this paper, we consider LTL formulas in NNF (Negation Normal Form), which can be acquired by pushing all negations inwards so that all remaining negations appear only in front of atoms. As a result, we need only consider the literal, $\wedge$, $\vee$, X, U and R operators if the formulas are in NNF. In this paper we use ϕ, ψ to represent LTL formulas, and α, β for propositional formulas.

Since we consider LTL in NNF, formulas are interpreted on infinite literal sequences over $\Sigma := 2^L$. A *trace* $\xi = \omega_0\omega_1\omega_2\ldots$ is an infinite sequence over Σ^ω. For ξ and $k \geq 1$ we use $\xi^k = \omega_0\omega_1\ldots\omega_{k-1}$ to denote the prefix of ξ up to its k-th element, and $\xi_k = \omega_k\omega_{k+1}\ldots$ to denote the suffix of ξ from its $(k+1)$-th element. Thus, $\xi = \xi^k\xi_k$. Before giving the LTL semantics, we first introduce the notion of *consistent trace*:

Definition 1 (Consistent Trace)**.** *We say a literal set A is* consistent *iff for all $a \in A$ we have that $\bigwedge a \not\equiv \bot$. A trace $\xi = \omega_0\omega_1\ldots$ is consistent iff ω_i is consistent for all $i \geq 0$.*

Let $\omega \in \Sigma$ be a consistent set of literals, and the semantics of LTL operators with respect to a consistent trace ξ is given by:

- $\xi \models \alpha$ iff $\xi^1 \models \alpha$;
- $\xi \models \phi_1 \wedge \phi_2$ iff $\xi \models \phi_1$ and $\xi \models \phi_2$;
- $\xi \models \phi_1 \vee \phi_2$ iff $\xi \models \phi_1$ or $\xi \models \phi_2$;
- $\xi \models X\ \phi$ iff $\xi_1 \models \phi$;
- $\xi \models \phi_1\ U\ \phi_2$ iff there exists some $i \geqslant 0$ such that $\xi_i \models \phi_2$ and for all $0 \leqslant j < i, \xi_j \models \phi_1$;
- $\xi \models \phi_1\ R\ \phi_2$ iff either $\xi_i \vDash \phi_2$ for all $i \geq 0$, or there exists some $i \geq 0$ with $\xi_i \models \phi_1 \wedge \phi_2$ and $\xi_j \vDash \phi_2$ for all $0 \leq j < i$;

Informally speaking, $X\phi$ holds iff ϕ holds in the next state; $\phi_1 U \phi_2$ holds iff there exists a position in which ϕ_2 holds, and before that (not including current position) ϕ_1 continuously hold; $\phi_1 R \phi_2$ holds iff ϕ_2 holds forever, or there exists a position in which ϕ_1 holds, and before that (including current position) ϕ_2 continuously holds.

B. Formula Progression

The *formula progression* technique was first introduced in the goal planning [10] in Artificial Intelligence (AI) area. The algorithm takes an LTL formula ϕ and an assignment $P \subseteq L_\phi$ as the inputs, and then computes the next formula ϕ' after P acting on ϕ. The explicit definition of *formula progression* is as follows:

Definition 2 (Formula Progression)**.** *Given an LTL formula ϕ and an assignment P over L_ϕ, we define $fprog(\phi, P)$ to compute the next formula after P acting on ϕ. Explicitly, this operator can be defined recursively over ϕ:*

- *If $\phi = a$ is a literal, then $fprog(\phi, P) = \top$ iff $a \in P$; Otherwise $fprog(\phi, P) = \bot$;*
- *If $\phi = \phi_1 \vee \phi_2$, then $fprog(\phi, P) = fprog(\phi_1, P) \vee fprog(\phi_2, P)$;*
- *If $\phi = \phi_1 \wedge \phi_2$, then $fprog(\phi, P) = fprog(\phi_1, P) \wedge fprog(\phi_2, P)$;*
- *If $\phi = X\phi_2$, then $fprog(\phi, P) = \phi_2$;*
- *If $\phi = \phi_1 U \phi_2$, then $fprog(\phi, P) = fprog(\phi_2, P) \vee (fprog(\phi_1, P) \wedge \phi)$;*
- *If $\phi = \phi_1 R \phi_2$, then $fprog(\phi, P) = fprog(\phi_2, P) \wedge (fprog(\phi_1, P) \vee \phi)$;*

Note that in the literature [10], the input formula must be co-safety [11], i.e., Release-free. The reason is that the Release formulas may not be checked within finite steps, which however is often required but not always holds in practice. In the real situation, runtime verification checks the traces of the program under scrutiny in a step-by-step manner, so the framework works only when the results can be achieved in finite steps, but leaves the new formulas to be checked further.

For simplicity, we say that ϕ' is derived from ϕ after being acted on by P, denoted as $\phi \xrightarrow{P} \phi'$, if $fprog(\phi, P) = \phi'$ holds. And given a finite trace $\eta = \omega_0\omega_1 \ldots \omega_n (n \geq 0)$, we write $\phi \xrightarrow{\eta} \phi'$ if there exists $\phi \xrightarrow{\omega_0} \phi_1 \xrightarrow{\omega_1} \ldots \phi_n \xrightarrow{\omega_n} \phi'$. Then the following theorem establishes the theoretical foundation of runtime verification with formula progression:

Theorem 1. *Let η be a finite trace and ξ be an arbitrary infinite trace. Then for an LTL formula ϕ:*

- *$\phi \xrightarrow{\eta} \top$ holds implies $\eta \cdot \xi \models \phi$;*
- *$\phi \xrightarrow{\eta} \bot$ holds implies $\eta \cdot \xi \not\models \phi$;*

According to the above theorem, the runtime verification procedure based on formula progression can be implemented in the following way: Let ϕ be the negation of the specification and η be the finite trace extracted from the program's execution path, then

1) If $\phi \xrightarrow{\eta} \top$, then the execution path of the program under scrutiny does not satisfy the specification, so we can terminate the whole verification process and give the negative result;
2) If $\phi \xrightarrow{\eta} \bot$, then the execution path of the program under scrutiny satisfies the specification, which indicates that we can stop the current execution and start another one with a new execution path;
3) If $\phi \xrightarrow{\eta} \phi'$ and ϕ' is neither $\top$ or $\bot$, then whether the execution path of the program satisfies the specification cannot be decided within the finite steps, and needs to be checked further.

III. Convergent Formula Progression

In the previous section we introduce the formula progression technique that is used in runtime verification. However, there are two main factors affecting the performance:

1) The selected execution path. This is essentially based on the initial status (inputs) of the program under verification. A good choice of the inputs, which results in a satisfied/violated path can lead to a fast verification. But such paths are not always easily found;
2) The sizes of generated formulas from formula progression. Often, it needs to explore more if larger formulas are generated, which may be even worse with respect to the divergence of formula progression.

To overcome the second point mentioned above, we present in this section the *convergent formula progression* technique in runtime verification framework, and we show that the exact bound for the sizes of generated formulas with convergent formula progression. First, we introduce some formula simplification rules for LTL which are used in the fix-point reduction technique introduced in the following.

Theorem 2 (Simplification)**.** *Let ϕ_1, ϕ_2 and ϕ_3 be LTL formulas, the following equations hold:*

1) $(\phi_1 U \phi_2) \wedge \phi_2 \equiv \phi_2$;
2) $(\phi_1 U \phi_2) \vee \phi_2 \equiv \phi_1 U \phi_2$;
3) $(\phi_1 R \phi_2) \wedge \phi_2 \equiv \phi_1 R \phi_2$;
4) $(\phi_1 R \phi_2) \vee \phi_2 \equiv \phi_2$;
5) $(\phi_1 R \phi_2) \wedge \phi_1 \equiv \phi_1 \wedge \phi_2$;
6) $(\phi_1 U \phi_2) \vee (\phi_3 U \neg \phi_2) \equiv \top$;
7) $(\phi_1 R \phi_2) \wedge (\phi_3 R \neg \phi_2) \equiv \bot$.

The above simplifications aim to reduce the temporal operators U and R, as their expansions are likely to enlarge the sizes of expanded formulas. Note that such simplifications are not limited: we here just list some that are used frequently in practice. Actually, the equations proposed above are not enough to avoid the potential explosion, and the following example shows the pure formula progression may be divergent.

Example 1. *Consider the formula $\phi = \phi_1 U \phi_2$ and the infinite trace $\xi = P^\omega$ where $P \in \Sigma$. Then we let $\psi_1 = fprog(\phi, P)$ and $\psi_n = fprog(\psi_{n-1}, P)(n \geq 2)$. We know that $\phi \equiv \phi_2 \vee (\phi_1 \wedge X\phi)$, thus $\psi_1 = fprog(\phi_2, P) \vee (fprog(\phi_1, P) \wedge \phi)$. Moreover, we have $\psi_2 = fprog^2(\phi_2, P) \vee (fprog^2(\phi_1, P) \wedge (fprog(\phi_2, P) \vee (fprog(\phi_1, P) \wedge \phi)))$. Iteratively, we can compute $\psi_n = fprog^n(\phi_2, P) \vee (fprog^n(\phi_1, P) \wedge (fprog^{n-1}(\phi_2, P) \vee (fprog^{n-1}(\phi_1, P) \wedge (\ldots \wedge (fprog(\phi_2, P) \vee (fprog(\phi_1, P) \wedge \phi))))))$. From the equations above we know, the size of ψ_n grows divergently if the equations $fprog^n(\phi_2, P) = fprog^{n-1}(\phi_2, P)$, $fprog^n(\phi_1, P) = fprog^{n-1}(\phi_1, P)$ are not detected, in which the evaluations are not $\top$ or $\bot$.*

If we consider the case that $\phi = \phi_1 R \phi_2$ is a Release formula, a similar situation also occurs. From the view of fix-point theory, the U (Until) operator is considered as the least fix point and the R (Release) operator is the largest fix point – This causes the potentially infinite iterations unless the fix point is found. However, as far as we know, previous literatures about formula progression did not study the fix-point of the generated formulas. In the example above, if the equations $fprog^n(\phi_2, P) = fprog^{n-1}(\phi_2, P)$ and $fprog^n(\phi_1, P) = fprog^{n-1}(\phi_1, P)$ are not recognized, the sizes of generated formulas from formula progression grow up unlimitedly. In the following, we propose a reduction strategy for formula progression, which successfully avoids the unlimited growing up.

Theorem 3 (Fix-point Reduction)**.** *Let the formulas $f_1(n)$ and $f_2(n)$ be defined recursively in the following way: 1)*

$f_1(1) = \phi_3 \vee (\phi_2 \wedge \phi_1)$, $f_2(1) = \phi_3 \wedge (\phi_2 \vee \phi_1)$*; and 2)* $f_1(n) = \phi_{2n+1} \vee (\phi_{2n} \wedge f_1(n-1))$, $f_2(n) = \phi_{2n+1} \wedge (\phi_{2n} \vee f_2(n-1))$ *when* $n > 1$. *Then the the following statements are true:*

1) $\phi_{2n+1} = \phi_{2k+1} (n > k) \Rightarrow f_1(n) \equiv f_1(n)[\bot/\phi_{2k+1}]$;
2) $\phi_{2n} = \phi_{2k} (n > k) \Rightarrow f_1(n) \equiv f_1(n)[\top/\phi_{2k}]$;
3) $\phi_{2n+1} = \phi_{2k+1} (n > k) \Rightarrow f_2(n) \equiv f_2(n)[\top/\phi_{2k+1}]$;
4) $\phi_{2n} = \phi_{2k} (n > k) \Rightarrow f_2(n) \equiv f_2(n)[\bot/\phi_{2k}]$.

Theorem 3 comes from the observation that the operators U and R follow the corresponding expansion rules during *formula progression*, i.e., $\phi_1 U \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge X(\phi_1 U \phi_2))$ which has the form of $f_1(n)$, and $\phi_1 R \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee X(\phi_1 R \phi_2))$ which has the form of $f_2(n)$. The formula structures satisfying the conditions in the theorem are easily detected in application. The *convergent formula progression* is the formula progression weaponed with the laws in Theorem 3. Given an LTL formula ϕ we denote the size of ϕ as $|\phi|$. Then we show next that the sizes of formulas processed by convergent formula progression are bounded.

Theorem 4 (The Convergency Theorem)**.** *Let* ϕ, ψ *be LTL formulas and* $\eta = \omega_0 \omega_1 \omega_2 \ldots \omega_{n-1}$ *be a finite trace. Then under the convergent formula progression framework,* $\phi \xrightarrow{\eta} \psi$ *holds implies* $|\psi| \leq 2^{|\phi|}$.

Theorem 4 shows that the sizes of formulas processed by formula progression are bounded to $2^{|\phi|}$ where ϕ is the input formula. Using the reduction strategy above, we can highly optimize the formula progression, and more details are shown below. Moreover, this is the first bound ever given to the sizes of expanded formulas in formula progression, to the best of our knowledge.

Although the same upper bound for automata construction for LTL has been shown in [12], the one for formula progression is necessary. In [12], each state of automata is a set of subformulas of the input formula (property) ϕ and the subformulas in the set are distinctive, thus the size of generated automata is bounded by $2^{|\phi|}$. However in the framework of formula progression, the subformulas may repeat in a state, which implies that the size of generated state can be divergent. So the conclusion in [12] cannot be adapted into the formula progression directly, and an exact bound should be proposed clearly.

IV. Runtime Verification Framework

In this section we introduce the explicit runtime verification framework used in our experiments. Fig. 1 shows such a snapshot. As indicated in the figure, we extract four components from the framework, i.e. *Program*, *Status Pool*, *CFP-based Monitor* and *Configuration* modules. Now we explain them below, respectively.

a) Program: In the framework, *Program* module is the executable source codes in the computer. As there is the *Status Pool* module caching the program status and from

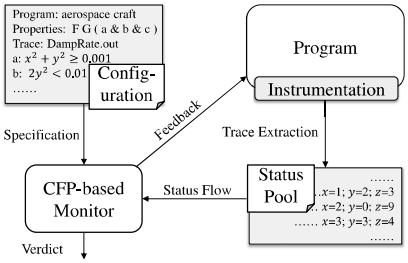

Figure 1: Runtime Verification Framework

which the *Monitor (i.e., CFP-based Monitor)* fetches status, the *Monitor* is independent on the *Program*. This indicates that the source codes can be in any programming language, such as C, Java and etc. To collect the program status, instrumentation has to be done to provide with the interfaces, which output the evaluations of concerning variables, as well as indicate the way feedbacks communicating to the original Program. The interfaces must obey the protocol of *Status Pool* module.

b) Status Pool: The *Status Pool* module stores the program status (trace information) sequentially. It could be a block in memory or a separate file. The existence of this module successfully makes the *Monitor* independent on the *Program*, which makes our framework more flexible.

In our experiment, we make it a log file, *Trace* file. Each line of the *Trace* file records a set of assignments for desired variables in the *Program* in a period. Every assignment is separated by the ";" notation. These data are further processed by the *Monitor* module such that they can match the atoms appearing in the properties for verification. During the verification process, the *Trace* file is dynamically updated whenever the *Program* produces fresh status, or after a period to cut down the consumption of opening or closing files.

c) Configuration: This module is the main input of the *Monitor*. In our experiment, we make it a file. As indicated in Fig. 1, it contains the specification and all the other related information that the *Monitor* needs when constructed, i.e., the name of program under scrutiny, the *Trace* file path, one or more LTL properties to be verified, and the mapping of the atoms on the program variables. A *block* of this file must contain all the above information, and the file recognizes one or more blocks. The atoms of LTL properties represent the evaluations of the expressions on program variables. For instance, as shown in the figure, x is a program variable and $x > 1$ is an expression whose value is to be evaluated by the *Monitor*. The corresponding mechanism is introduced below.

d) CFP-based Monitor: The *Convergent Formula Progression based Monitor* module constitutes the core of our runtime verification framework. And the *Convergent*

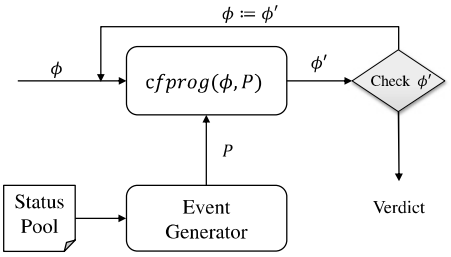

Figure 2: The verification workflow in the *CFP-based Monitor*.

Formula Progression separates our framework from others apparently. It takes the *Configuration* information and the execution trace of the *Program* as input. Mainly, this module has to finish the following five tasks in a period:

1) Fetch the current program status from the *Status Pool*;
2) Determine the values of atoms in LTL formulas by evaluating the expressions they represent with the program status information;
3) Invoke the *convergent formula progression* algorithm;
4) Check whether the properties being verified are satisfied or not, and give out the verdicts;
5) Send the feedback to the *Program* module.

The module follows the processes above recursively until a specific verification result (i.e., $\top$ or $\bot$) is achieved. Here, the *CFP-based Monitor* contains a parser to extract the execution status from the *Status Pool*. As the atoms and their expressions are all indicated in the *Configuration* module, the *Monitor* module also includes an evaluator to determine the values of atoms in LTL properties. Both the parser and evaluator are integrated into the *Event Generator* component, as shown in Fig. 2.

The figure also illustrates the verification workflow in *CFP-based Monitor* module. Once the current assignment P comes, *Monitor* invokes the convergent formula progression algorithm with the parameters P and the input formula ϕ, denoted by $cfprog(\phi, P)$ in the figure, and returns the next formula ϕ', which represents the next checking goal. If $\phi' = \top$, the *Monitor* returns the result that the LTL property under scrutiny is violated at exactly the checking point, and sends the feedback to the *Program* to stop the execution. If ϕ' is $\bot$, then the current program trace has satisfied the property, and it sends the feedback to the *Program* to ask for a new trace for more complete verification. Otherwise, *Monitor* sends the feedback to the *Program* asking for more status information on the current execution trace, replaces ϕ with ϕ' and then the verification proceeds.

Now we use a running example to illustrate the whole workflow in our runtime verification framework:

Example 2. *Assume that we have a Program M, which has three crucial variables x, y, z. Now we want to verify whether $M \models (\neg(x > 1)R\neg(y < 10))R\neg(1 < z < 5)$ holds. Then following our framework, we have:*

1) *Let a, b, c be atoms on $x > 1$, $y < 10$ and $1 < z < 5$ respectively, and write the mappings into the Configuration file together with the negation formula $\neg((\neg aR\neg b)R\neg c)$, i.e., $(aUb)Uc$;*
2) *The Monitor reads the trace information from the Trace file. Assume here the prefix of the trace is $\{x = 2; y = 12; z = 6\}\{x = 2; y = 12; z = 6\}\{x = 1; y = 9; z = 4\}$. Then the Monitor read the first status and evaluates that $a = \top, b = \bot, c = \bot$. So the assignment $P = \{a, \neg b, \neg c\}$ is created;*
3) *Let $\phi = (aUb)Uc$. The Monitor invokes the convergent formula progression algorithm, and computes that $\phi' = cfprog(\phi, P) = (aUb) \wedge (aUb)Uc$. As ϕ' is not yet $\top$ or $\bot$, so the Monitor continues the verification process;*
4) *Now $\phi = (aUb) \wedge (aUb)Uc$ and $P = \{a, \neg b, \neg c\}$ from the second status. Invoking the convergent formula progression again we get $\phi' = (aUb) \wedge ((aUb) \wedge (aUb)Uc)$. And by our Fix-point Reduction, we get $\phi' = (aUb) \wedge (aUb)Uc$;*
5) *Repeat the formula progression again with $\phi = (aUb) \wedge (aUb)Uc$ and $P = \{\neg a, b, c\}$, then we get $\phi' = \top$, which implies that the Program M violates the property $(\neg(x > 1)R\neg(y < 10))R\neg(1 < z < 5)$. So the Monitor sends the feedback to terminate the Program, and the verification process ceases.*

V. Experiments

In this section we talk about the experimental details under the runtime verification framework.

A. Experimental Strategies

a) Platform: We conduct our experiments in a computer which contains two 2.93GHz Intel Core Duo CPUs. The computer has a 4 GB RAM and the operating system is Ubuntu 12.04.

b) Model (Program): In the experiments, we select a model from China Academy of Space Technology (CAST). A detailed description about this model can be found in the literatures [13], [14]. This model is a periodical embedded system that is used to control and adjust the attitude of satellites. It consists of 8 different modes, and in every period the system (model) is in one of them. All the modes are implemented in C language, and contain about 2000-3000 lines of source codes. To check whether the *Program* satisfies an LTL property, we must decide whether all these 8 sub programs satisfies the property. Although in the system level, the complete verification is required during the transformation among different modes, it can be less complicated to verify each sub programs separately. The mechanism is trivial: all sub programs satisfy the LTL property implies

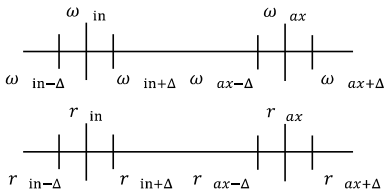

Figure 3: The bound description of the variables ω and r in the experimental model.

that their combination, i.e., the whole system, also satisfies this property. For simplicity, we use the notations $M1$-$M8$ to represent these 8 sub programs.

Note that runtime verification may not terminate if $\top$ or $\bot$ cannot be reached on an infinite execution trace, which hardly contributes to the evaluation of our experiments, but only causes unnecessary cost. To avoid this non-termination, we address two limits to our experiments: 1) The length of the trace from the program execution is bounded by 10,000; 2) The timeout for verifying on each trace is set to be 10 minutes for each property. The first limitation also makes us verify as many program executions as possible, which leads the runtime verification closer to ideal goal of complete verification. In the experiments, we randomly choose 100 execution traces for each mode (sub program).

c) LTL Properties: In our experiments we choose sets of random formulas [15] and five kinds of pattern formulas that are used frequently in model checking as our benchmarks.

Random formulas are generated by randomly choosing atoms and connectives (operators). Some such generating algorithms are presented by invoking the random functions in programming languages. There are also a few executable scripts that implement the generating algorithms, see [15]. Using these scripts, one can generate the formulas they want by fixing the atom number, formula size, and the appearing frequency of each operator. In our experiments, we generate five groups of random formulas, varying on size from 10 to 50, fixing atom number to 8 and varying on the operator frequencies ([0,1]) as well. We generate 100 different formulas for each group, and also guarantee those among groups are distinct. In the following, we use the notations $F1$-$F5$ to represent these five groups of random formulas.

Researches show that there are five kinds of pattern formulas that are used frequently in model checking area [2]. Their briefly descriptions are as below (Note that P, Q, R in the formulas below represent arbitrary LTL formulas.):

- $P1$ (Globally) : GP. It represents that P must hold in every position;
- $P2$ (Before R): $FR \rightarrow (PUR)$. It means that P must hold before R holds;
- $P3$ (After Q): $G(Q \rightarrow GP)$. This pattern indicates that if Q holds, then P must hold after at some time – And this must be always true;
- $P4$ (Between Q and R): $G((Q \wedge \neg R \wedge FR) \rightarrow (PUR))$. It means that it is always true that, Q holds currently and R will hold at some future time but now, implies P holds at every position from the one Q holds to that R holds;
- $P5$ (After Q until R): $G(Q \wedge \neg R \rightarrow (GP \vee (PUR)))$. It means that it is always true that, Q holds currently implies P holds at every position until that R holds (Note here R may never hold, so P should hold forever in this situation).

In our experimental model, there are two crucial variables that are used to achieve the goal of controlling and adjusting satellite's attitude: ω and r which represent the angle value and angular rate of the satellite, respectively. In practice, both values must be within the safe ranges, which are shown in Fig. 3. As depicted in the figure, both ω and r must be normally in the ranges of $[\omega_{min}, \omega_{max}]$ and $[r_{min}, r_{max}]$. However, they are also allowed to have the deviation of Δ due to the continuity of physical property. Thus, although the safe area of ω is from ω_{min} to ω_{max}, it is indeed allowed to be within the range of $[\omega_{min-\Delta}, \omega_{max+\Delta}]$. The similar situation also applies to variable r.

According to the statements from system developers, we summarize the following specifications on these two variables: 1) The values of ω and r must be always in the ranges of $[\omega_{min-\Delta}, \omega_{max+\Delta}]$ and $[r_{min-\Delta}, r_{max+\Delta}]$; 2) Before $\omega \geq \omega_{min}$, the angular rate $r \geq r_{max}$ must hold such that the angle can reach the regular bound $[\omega_{min}, \omega_{max}]$ as soon as possible; 3) After $\omega \geq \omega_{max-\Delta}$, the angular rate $r \leq r_{max}$ must hold so that ω can go back to be regular; 4) Once ω is in the very safe area, i.e., $[\omega_{min+\Delta}, \omega_{max-\Delta}]$, the angular rate r can be any of its allowed value, i.e. $r \in [r_{min-\Delta}, r_{max+\Delta}]$; 5) After $\omega \geq \omega_{max}$, the angular rate $r \leq r_{min}$ must hold so that ω go back to be regular as soon as possible until it reaches the safe area, i.e., $[\omega_{min}, \omega_{max}]$. Compared to the third specification, this one is more crucial. Based on the above observations, we establish a mapping from the specifications to the five pattern formulas, which is shown in Table I.

d) Evaluation: As our work focuses on the improvement of formula progression, which is a dynamic DFA construction technique, we do not compare our work with the static DFA construction techniques, but with the original one. And as mentioned above, there are seldom open runtime verification toolkits using formula progression so far [5]. We could not compare the performance of our framework as well as the verification results with those of others that are off-the-shelf, such as *JUnit*^{RV}[16]. So the evaluation in our experiments follows this way: To show the efficiency of the proposed Fix-point Reduction (Theorem 3) in the paper, we compare the verification costs between the original

[2]http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml

Table I: The mapping from expressions in model to atoms in the pattern formulas.

	P	Q	R
$P1$	$\omega \geq \omega_{min-\Delta} \wedge$ $\omega \leq \omega_{max+\Delta} \wedge$ $r \geq r_{min-\Delta} \wedge$ $r \leq r_{max+\Delta}$	-	-
$P2$	$r \geq r_{max}$	-	$\omega \geq \omega_{min}$
$P3$	$r \leq r_{max}$	$\omega \geq \omega_{max-\Delta}$	-
$P4$	$r \geq r_{min-\Delta} \wedge$ $r \leq r_{max+\Delta}$	$\omega \geq \omega_{min+\Delta}$	$\omega \leq \omega_{max-\Delta}$
$P5$	$r \leq r_{min}$	$\omega \geq \omega_{max}$	$\omega \geq \omega_{min} \wedge$ $\omega \leq \omega_{max}$

runtime verification framework and the convergent one, which integrates the Fix-point Reduction; And moreover, to show the scalability of the framework, we run sets of random LTL properties as well as specialized pattern formulas that are used frequently in model checking. By fixing the *Program*, We investigate the verification results and performance, which somewhat affirm the usefulness of our proposed framework.

Summarily, the notations used in the experimental part are listed as below. 1) M1-M8: The eight different models extracted from our project. Each model contains 100 execution traces, the length of which is bounded by 10,000. The timeout addressed on each trace is 10 minutes for each property; 2) F1-F5: The five groups of random LTL formulas. Each group contains 100 different formulas and each formula is distinct from that in other groups. 3) P1-P5: The five frequently used patterns.

B. Experimental Results

In this section we exhibit the experimental results. We first verify the five meaningful properties corresponding to those frequently used patterns, i.e., $P1$ to $P5$, with our convergent formula progression framework. The models under scrutiny are chosen from $M1$ to $M8$. Table II details the verification results. The value of each meta is a tuple, e.g., "(22.3, Unv)" on program $M1$ and the property in pattern $P1$, indicating the cost is 22.3 seconds and this property is unverified on this run. Here, "Sat", "Vio" represent "Satisfied" and "Violated", respectively.

We highlight the three cases in which the properties under scrutiny are violated, and send the report to the program developers. With the professional and careful judgment, the selection of the deviations (Δ) for ω and r matters the most to the result whether the property is satisfied or not. And we get the feedback that they would take it more rigorous to avoid the potential "bug" in further application. This is an evidence showing that the runtime verification framework proposed in this paper is useful and promising.

Then we show how the Fix-point Reduction plays the key role in the performance of formula progression. We test the cases on $M1$-$M8$, with properties chosen from

Table II: Verification Results on the Five Pattern Properties

	$P1$		$P2$		$P3$		$P4$		$P5$	
$M1$	22.3	Unv	5.6	Sat	27.3	Unv	27.2	Unv	6.7	Unv
$M2$	22.8	Unv	5.8	Sat	27.2	Unv	27.5	Unv	21.2	**Vio**
$M3$	22.9	Unv	6.5	Sat	27.0	Unv	19.7	Unv	22.4	Unv
$M4$	7.8	**Vio**	7.6	Sat	28.9	Unv	19.2	Unv	21.8	Unv
$M5$	23.0	Unv	5.4	Sat	28.9	Unv	18.7	Unv	23.1	Unv
$M6$	21.7	Unv	6.9	Sat	24.7	Unv	6.8	**Vio**	19.9	Unv
$M7$	22.6	Unv	6.4	Sat	26.8	Unv	17.6	Unv	24.5	Unv
$M8$	25.4	Unv	9.8	Sat	26.8	Unv	19.6	Unv	23.1	Unv

Table III: Verification Results on Random Properties

	$F1$		$F2$		$F3$		$F4$		$F5$	
$M1$	387	40/60	389	44/56	399	41/59	399	44/56	373	53/47
$M2$	392	40/60	389	34/66	399	41/59	399	53/47	372	53/47
$M3$	383	37/63	389	54/46	390	53/47	399	46/54	372	46/54
$M4$	373	29/71	374	45/55	379	60/40	398	54/46	372	43/57
$M5$	393	32/68	389	45/55	380	47/53	400	49/51	372	54/46
$M6$	391	40/60	390	53/47	399	61/39	400	47/53	373	55/45
$M7$	395	44/56	395	41/59	394	41/59	398	54/46	373	45/55
$M8$	392	45/55	392	54/46	389	40/60	399	47/53	373	52/48

random benchmarks $F1$-$F5$. The results of our convergent framework are shown in Table III, and those of the original one are omitted for the sake of brevity. Also, each meta contains a tuple information, e.g., "(387, 40/60)" on program $M1$ and property group $F1$, in which "387" is the running cost of the verification on $M1$, and "40/60" indicates that among all the 100 properties, 40 ones are verified to get an explicit verdict, either satisfied or violated, and the results of other 60 ones are not known yet, i.e., when the timeout or the maximal trace length is reached.

This experiment shows that, each verification process successfully finishes in 10 minutes under our convergent runtime verification framework. Every status of the bounded trace is processed before the timeout meets. However, there are more than 80% of verification cases cannot terminate within timeout, i.e., more than 8000 cases out of the total amount of 10,000 (100 traces multiple 100 properties for each model), under the original runtime verification framework. In these cases, the rest of traces are yet to be processed. This is mainly ascribed to the blow-up of generated formulas size, which causes the iterative invoking of the technique becomes increasingly slower. By weaponing the Fix-point Reduction, the running cost is dramatically cut down.

In the worst case, the size of generated formula increases unlimitedly from the original formula progression. For example, there is a trace $\{a\}^*$ in $M1$ and a formula $(aUb)Uc$ in group $F1$, where a, b, c are atoms mapping to expressions from the model. The experiment data shows that, after the progression is invoked more than 100 times, i.e., the monitor has processed the 100-length prefix of the trace, the input formula of formula progression becomes extremely large, which causes the hardly obtaining of the next formula. Thus our experiments show accurately the importance and efficiency of the Fix-point Reduction.

From Table III we see that there are tiny differences

among the verification costs on each program. The costs all range from 370 seconds to 400 seconds. More interesting, there are quite a few random properties that are violated by the programs. For example, consider the formula $\phi = (h \rightarrow Fg)Uc$, in which h represents $r \geq r_{max}$, g represents $r \leq r_{min}$ and c means $\omega \leq \omega_{max}$. So the formula informally means the value of the angular rate r is in $[r_{min}, r_{max}]$ until $\omega \leq \omega_{max}$ holds. To our best knowledge, this property should be satisfied in all programs, but the result shows that it is violated in $M2$. We submit this "bug" to the developers, and encouragingly it is accepted. However it is not always true that the violated properties detected are interesting: They may be meaningless in practice. Approximately there are about 1% that are meaningful, and only 4 of them are accepted.

VI. Conclusion

In this paper we propose the convergent formula progression with the novel fix-point reduction technique. We prove the exact bound of generated formulas with the convergent formula progress. Meanwhile, we implement this new technique in the runtime framework and the experiments not only show the efficiency of the method we proposed, but also apply the new framework to a real case from industry which succeeds to help the developers on bug detection and future development.

Acknowledgement

We thank anonymous reviewers for the useful comments. Geguang Pu is partially supported by Shanghai Knowledge Service Platform NO. ZF1213. Jianwen Li is partially supported by STCSM project NO. 14511100400. Ting Su is partially supported by NSFC project NO. 61021004. Yan Shen is also supported by NSFC project NO. 61361136002 and 91118007. Bin Fang is partially supported by China HGJ project NO. 2014ZX01038-101-001. Zheng Wang is supported by the Open Project of Shanghai Key Laboratory of Trustworthy Computing No. 07dz22304201304 and NSFC No. 91118007. Wanwei Liu is partially supported by NSFC grant No. 61103012. Mingsong Chen is partially supported by NSFC grant No. 61202103.

References

[1] L. Fix, "Fifteen years of formal property verification at Intel," in *Proc. 2006 Workshop on 25 Years of Model Checking*, ser. Lecture Notes in Conmputer Science. Springer, 2007.

[2] E. Clarke, "The birth of model checking," *This Volume*, 2007.

[3] D. W. Loveland, *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science)*. sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.

[4] H. Barringer, K. Havelund, D. Rydeheard, and A. Groce, "Rule systems for runtime verification: A short tutorial," in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Bensalem and D. Peled, Eds. Springer Berlin Heidelberg, 2009, vol. 5779, pp. 1–24.

[5] A. Bauer, J.-C. Küster, and G. Vegliach, "Runtime verification meets android security," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, A. Goodloe and S. Person, Eds. Springer Berlin Heidelberg, 2012, vol. 7226, pp. 174–180.

[6] Y. Falcone, K. Havelung, and G. Reger, "A Tutorial on Runtime Verification," in *Engineering Dependable Software Systems*, ser. NATO Science for Peace and Security Series - D: Information and Communication Security, G. K. Manfred Broy, Doron Peled, Ed. IOS Press, 2013, vol. 34, pp. 141–175, summer School Marktoberdorf 2012.

[7] O. Kupferman and M. Vardi, "Model checking of safety properties," *Form. Methods Syst. Des.*, vol. 19, no. 3, pp. 291–314, Oct. 2001.

[8] K. Rozier and M. Vardi., "Deterministic compilation of temporal safety properties in explicit state model checking," in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7857, pp. 243–259.

[9] L. Zhao, T. Tang, J. Wu, and T. Xu, "Runtime verification with multi-valued formula rewriting," in *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on*, Aug 2010, pp. 77–86.

[10] F. Bacchus and F. Kabanza, "Planning for temporally extended goals," *Ann. of Mathematics and Artificial Intelligence*, vol. 22, pp. 5–27, 1998.

[11] E. Kindler, "Safety and liveness properties: A survey," *Bulletin of the EATCS*, vol. 53, pp. 268 – 272, 1994.

[12] M. Vardi, "An automata-theoretic approach to linear temporal logic," in *Logics for Concurrency: Structure versus Automata*, ser. Lecture Notes in Computer Science, F. Moller and G. Birtwistle, Eds., vol. 1043. Springer, 1996, pp. 238–266.

[13] Z. Wang, J. Li, Y. Zhao, Y. Qi, G. Pu, J. He, and B. Gu, "Spardl: A requirement modeling language for periodic control system," in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2010, vol. 6415, pp. 594–608.

[14] J. Li, G. Pu, Z. Wang, Y. Chen, L. Zhang, Y. Qi, and B. Gu, "An approach to requirement analysis for periodic control systems," *2012 35th Annual IEEE Software Engineering Workshop*, vol. 0, pp. 130–139, 2012.

[15] K. Rozier and M. Vardi, "LTL satisfiability checking," *Int'l J. on Software Tools for Technology Transfer*, vol. 12, no. 2, pp. 1230–137, 2010.

[16] N. Decker, M. Leucker, and D. Thoma, "*jUnit*^{rv}: adding runtime verification to junit," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7871, pp. 459–464.