Runtime Verification with Predictive Semantics

Xian Zhang[1], Martin Leucker[2], and Wei Dong[1]

[1] School of Computer, National University of Defense Technology, Changsha, China
{zhangxian,wdong}@nudt.edu.cn
[2] Institute for Software Engineering, University Lüeck, Lüeck, Germany
leucker@isp.uni-luebeck.de

Abstract. Runtime verification techniques are used to continuously check whether software execution satisfies or violates a given correctness property. In this paper, we extend our previous work of three-valued semantics for Linear Temporal Logic (LTL) to predictive semantics. Combined with the static analysis to the monitored program, the predictive semantics are capable of predicting monitored property's satisfaction/violation even when the observed execution does not convince it. We instrument the monitored program based on its Program Dependence Graph representation in order to emit "predictive word" at runtime. We also implement a prototype tool to support predictive semantics and use it to find predictive words in real, large-scale project. The result demonstrates that the predictive semantics are generally applicable in these projects.

Keywords: Runtime Verification, Three-Valued Semantics, Predictive Semantics, Program Dependence Graph.

1 Introduction

As software systems become more and more pervasive in everyday life, it is becoming increasingly necessary to guarantee their security and reliability. An approach of continuously monitoring software execution is becoming more and more popular. Runtime verification techniques do not assume the deployed software is correct, but continuous check whether software execution satisfies or violates a given correctness property. Runtime verification techniques are frequently used to prevent damage from happening when software is going to malfunction and are quite effective in practice.

We propose a predictive semantics for runtime verification in this paper. The predictive semantics enable monitors to foresee a property satisfaction or violation before the observed execution convince it. We formally define the predictive semantics used in runtime verification. We also give an algorithm on how to generate a monitor from a LTL formula and describe how to use it to check program's execution with predictive semantics.

The remainder of this paper is organized as follows: some preliminary information are first introduced in section 2; in section 3, we formally define runtime verification with predictive semantics; then in section 4, we give an algorithm to generate a monitor from a LTL formula and describe how to check monitored

A. Goodloe and S. Person (Eds.): NFM 2012, LNCS 7226, pp. 418–432, 2012.

program's execution with predictive semantics; in section 5, we implement a prototype tool to support predictive semantics; in section 6, we use this tool to do some experiments on some real projects; finally, we review related work in section 7 and conclude in section 8.

2 Preliminaries

2.1 Linear Temporal Logic and Büchi Automata

LTL is a widely used formalism for specifying and verifying correctness of computer programs. For the remainder of this section, let AP be a finite set of atomic proposition symbols and $\Sigma = 2^{AP}$ be a finite alphabet. Σ^* stands for all finite traces over Σ and Σ^ω stands for all infinite traces over Σ. Usually, u, v, u' and v' are used to denote elements in Σ^* and w and w' in Σ^ω. For a word $w \in \Sigma^\omega$, the prefix set of w is $\text{pref}(w) = \{u | \exists w' \in \Sigma^\omega : (w = uw')\}$.

The syntax of LTL is defined as follows.

$$\varphi ::= \text{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \,\mathrm{U}\, \varphi \mid \mathrm{X}\, \varphi$$

where $p \in AP$. There are some syntax sugar: $\mathrm{F}\,\varphi = \text{true}\ \mathrm{U}\,\varphi$; $\mathrm{G}\,\varphi = \neg\,\mathrm{F}\,\neg\varphi$. We define the semantics of a formula φ of LTL with respect to infinite traces. Let $w = a_0 a_1 ... \in \Sigma^\omega$ be a infinite word with $i \in \mathbb{N}$ being a position. We define the semantics of LTL formulae inductively as follows: $w, i \vDash \text{true}$; $w, i \vDash p$ iff $p \in a_i$; $w, i \vDash \neg\varphi$ iff $w, i \nvDash \varphi$; $w, i \vDash \varphi_1 \vee \varphi_2$ iff $w, i \vDash \varphi_1$ or $w, i \vDash \varphi_2$; $w, i \vDash \varphi_1 \,\mathrm{U}\, \varphi_2$ iff $\exists k \geq i$ with $w, k \vDash \varphi_2$ and $\forall i \leq l < k$ with $w, l \vDash \varphi_1$; $w, i \vDash \mathrm{X}\,\varphi$ iff $w, i+1 \vDash \varphi$.

In addition, $w, 0 \vDash \varphi$ can be abbreviated as $w \vDash \varphi$. We also use $L(\varphi) = \{w \in \Sigma^\omega \mid w \vDash \varphi\}$ to denote the set of models of a LTL formula φ. Two LTL formulae φ and ψ are called *equivalent* ($\varphi \equiv \psi$) iff $L(\varphi) = L(\psi)$. The language $L(\varphi)$ can be recognized by a corresponding *Nondeterministic Büchi Automaton*(NBA). *NBA* is defined as a tuple $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$, where: Σ is a finite alphabet; Q is a finite non-empty set of states; $Q_0 \subseteq Q$ is a set of initial states; $\delta : Q \times \Sigma \to 2^Q$ is a transition function; $F \subseteq Q$ is a set of accepting states.

In order to state conveniently, we extend the transition function $\delta : Q \times \Sigma \to 2^Q$ to $\delta' : 2^Q \times \Sigma^* \to 2^Q$ by $\delta'(Q', \epsilon) = Q'$ and $\delta'(Q', ua) = \bigcup_{q' \in \delta'(Q', u)} \delta(q', a)$ for $Q' \subseteq Q, u \in \Sigma^*$, and $a \in \Sigma$.

A *run* of an automaton $\mathcal{A}$ on a word $w = a_0 a_1 ... \in \Sigma^\omega$ is a sequence of states and input symbols $\rho = q_0 a_0 q_1 a_1 q_2 ...$, where q_0 is an initial state of $\mathcal{A}$ and $q_{i+1} \in \delta(q_i, a_i)$ for all $i \in N$. For a run ρ, we use $\text{Inf}(\rho)$ denote the states visited infinitely often. A run ρ of a NBA $\mathcal{A}$ is called *accepting* iff $\text{Inf}(\rho) \cap F \neq \emptyset$.

A *Nondeterministic Finite Automata* (NFA) $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$ is an automaton where Σ, Q, Q_0, δ and F are defined as for a Büchi automata. A NFA only accept finite words. A run of a NFA on a word $u = a_0 ... a_n \in \Sigma^*$ is a sequence of states and input symbols $\rho = q_0 a_0 q_1 a_1 ... a_n q_{n+1}$ where q_0 is an initial state and $q_{i+1} \in \delta(q_i, a_i)$ for all $i \in N$. The run is called accepting if $q_{n+1} \in F$. A NFA is called deterministic iff for all $q \in Q$, $a \in \Sigma$, $|\delta(q, a)| = 1$ and $|Q_0| = 1$.

2.2 Three-Valued Semantics for LTL in Runtime Verification

Andreas Bauer, Martin Leucker and Christian Schallhart have defined a three-valued semantics for LTL on finite traces in article [1], which is tailored to the use in runtime verification. The intuition is as follows: in theory, we observe an infinite sequence w of some system. Thus, for a given formula φ, either $w \vDash \varphi$ holds or not. In practice, however, we can only observe a finite prefix u of w. The three-valued semantics are based on whether the observed finite prefix will definitely lead to a *satisfactory* or *violation* verdict. The three-valued semantics are defined as follows.

Definition 1 (Three-valued semantics). *$u \in \Sigma^*$ is a finite word. The truth value of a LTL formula φ with respect to u, denoted by $[u \vDash \varphi]$, is an element of $\mathbb{B}_3$ ($\mathbb{B}_3 = \{\top, \bot, ?\}$) defined as follows:*

$$[u \vDash \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \vDash \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \nvDash \varphi \\ ? & \text{otherwise} \end{cases}$$

Note that in the above definition and the remainder of this paper, we use the notation $[u \vDash \varphi]$ to give a three-valued semantic to a formula based on a finite word u. Further details about three-valued semantics can be found in [1].

3 Predictive Semantics for LTL in Runtime Verification

In a lot of application areas, it is quite appreciated that the verdict can be predicted when the error prefix has not appeared. The three-valued semantics given in the previous section hold an assumption that: the observed word increase incrementally and letters arrive one at a time. When we judge the semantic value of φ with respect to the observed prefix u, we do not know what the following word is. Runtime verification is a body of verification techniques that check whether program's execution satisfies a formal property. The execution is generated by the program's code. If we can do some kinds of static analysis to program code and draw some summary information, it is possible to *predict* what the following word is. Based on this assumption, we propose a predictive semantics for LTL formulae. Although the predictive semantics are only given to LTL formulae in this paper, it can be easily extended to other property specification languages.

Our predictive semantics for LTL formulae are also based on the finite prefix word. But instead of without knowing what the future word is in three-valued semantics, future words are predictable in predictive semantics. The predictive semantics for LTL formulae are defined as follows.

Definition 2 (Predictive semantics). *$u \in \Sigma^*$ is an observed word generated by the monitored program so far, $v \in \Sigma^*$ is a finite predictive word which will be*

generated in the following time. The truth value of a LTL formula φ with respect to u and v, denoted by $[u \vDash_p^v \varphi]$, is an element of $\mathbb{B}_3$ defined as follows:

$$[u \vDash_p^v \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : uv\sigma \vDash \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : uv\sigma \nvDash \varphi \\ ? & \text{otherwise} \end{cases}$$

In runtime verification, the observed word u increases as the monitored program runs. There is a distinction between the observed word (which is *known*) and the word which is going to be generated (which is *unknown*). In the predictive semantics definition, v is the word which is going to be generated by the monitored program, but it is already known *now*. That is to say, the predictive word v can be *predicted* at present. That is why we name our semantics as *predictive semantics*. v is a predictive word.

The predictive semantics demand that the predictive word v should be identified every time the observed word u increases. That means, every time a monitored letter happens in execution, we have to predict the word which is going to be generated by the monitored program. Given the monitored program's complexity and dynamic feature, it is not realistic to predict a future word every time a monitored letter happens. In order to make predictive semantics feasible, we propose a implementation predictive semantics.

Only a fixed set of words are predictable in implementation predictive semantics. In predictive semantics definition, we are able to predict the future word every time a monitored letter happens. While in implementation predictive semantics definition, we only have to predict it when it belongs to a *fixed set*. This restriction releases the burden of predicting a future word every time a monitored letter happens. Because we only predict the future word when it falls into a fixed words set.

The implementation predictive semantics for LTL formulae are defined as follows.

Definition 3 (Implementation predictive semantics). *$u \in \Sigma^*$ is an observed word generated by the monitored program so far, $R \subseteq \Sigma^*$ is a fixed predictive words set found in the monitored program. Let v' denote the predictive word which is going to be generated by the monitored program. The truth value of a LTL formula φ with respect to u and R, denoted by $[u \vDash_i^R \varphi]$, is an element of $\mathbb{B}_3$ defined as follows:*

$$[u \vDash_i^R \varphi] = \begin{cases} \top & \text{if } v' \text{ is proven} \in R : (\forall \sigma \in \Sigma^\omega : (uv'\sigma \vDash \varphi)) \\ & \quad \text{or } (\forall \sigma' \in \Sigma^\omega : (u\sigma' \vDash \varphi)) \\ \bot & \text{if } v' \text{ is proven} \in R : (\forall \sigma \in \Sigma^\omega : (uv'\sigma \nvDash \varphi)) \\ & \quad \text{or } (\forall \sigma' \in \Sigma^\omega : (u\sigma' \nvDash \varphi)) \\ ? & \text{otherwise} \end{cases}$$

In implementation predictive semantics definition, R is a fixed predictive words set found in the monitored program through some kinds of static analysis. Different monitored programs have different predictive words sets. R contains all

predictable words for formula φ in the monitored program. The predictive word v' is quite different from the the word v in predictive semantics definition: v is an identified word in predictive semantics definition; while v' is only a place holder in implementation predictive semantics, we can only identify it when it is proven to belong to R. If v' does not belong to R, we can not predict the word which is going to be generated by the monitored program and the implementation predictive semantics degenerate into three-valued semantics. Hence, the implementation predictive semantics definition is a compromise between three-valued semantics and predictive semantics. This restriction sacrifices a kind of prediction (when v' is not proven to belong to R), but it also makes this semantics more applicable.

In implementation predictive semantics definition, not every predictive word's (the word belongs to R) occurrence in program's execution is predictable. If v' can not be *proven* $\in R$ through some kinds of static analysis, the implementation predictive semantics do not possess predictive capability. There are some situations in which v' is actually belongs to R but we can not prove it through static analysis.

4 Monitor Construction for LTL with Predictive Semantics

Predictive semantics definition is useless if we can not put it into practice. In this section, we will describe the process of generating a monitor from a LTL formula and describe how to use this monitor to check program's execution with predictive semantics. The monitor generating process is almost identical to the "monitor construction" section in article [1]. We only sketch it here.

4.1 Generating Monitor from LTL Formulae

Given a formula φ, we will construct a monitor (a Finite State Machine(FSM)) M^φ that reads finite word $u \in \Sigma^*$, and produces $[u \vDash_p^v \varphi]$ with respect to the predictive word v.

For a Nondeterministic Büchi Automata (NBA) A, we denote by $A(q)$ the NBA that coincides with A except for the set of initial states Q_0, which is redefined in $A(q)$ as q. Let us fix $\varphi \in LTL$ for the rest of this section, and let $A^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ denote the NBA that accepts all models of φ and let $A^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, F^{\neg\varphi})$ denote the NBA, which accepts all words falsifying φ. The corresponding construction is standard and explained, for example in [2].

For the automaton A^φ, we define a function $F^\varphi : Q^\varphi \to \mathbb{B}$(with $\mathbb{B} = \{\top, \bot\}$) where we set $F^\varphi(q) = \top$ iff $L(A^\varphi(q)) \neq \emptyset$. We evaluate a state q to $\top$ iff the language of the automaton starting in state q is not empty. To determine $F^\varphi(q)$, we identify in linear time the strongly connected components in A^φ which can be done using Tarjan's algorithm [3]. Using F^φ, we define the NFA $\hat{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, \hat{F}^\varphi)$ with $\hat{F}^\varphi = \{q \in Q^\varphi \mid F^\varphi(q) = \top\}$. Analogously, we set $\hat{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ with $\hat{F}^{\neg\varphi} = \{q \in Q^{\neg\varphi} \mid F^{\neg\varphi}(q) = \top\}$.

Lemma 1 (The evaluation of LTL formulae with predictive semantics). [1] *Let $u \in \Sigma^*$ is an observed word generated by the monitored program so far, $v \in \Sigma^*$ is a finite predictive word which will be generated in the following time. The truth value of a LTL formula φ with respect to u and v, denoted by $[u \vDash_p^v \varphi]$, is an element of $\mathbb{B}_3$ defined as follows:*

$$[u \vDash_p^v \varphi] = \begin{cases} \top & \text{if } uv \notin L(\hat{A}^{\neg\varphi}) \\ \bot & \text{if } uv \notin L(\hat{A}^{\varphi}) \\ ? & \text{otherwise} \end{cases}$$

The lemma yields the following procedure to evaluate the semantics of φ for a given finite word u and a predictive word v: we evaluate both $uv \notin L(\hat{A}^{\neg\varphi})$ and $uv \notin L(\hat{A}^{\varphi})$ and use the above lemma to determine whether $[u \vDash_p^v \varphi]$. Similarly, we have the same conclusion to the evaluation of LTL formulae with implementation predictive semantics as follows.

Lemma 2 (The evaluation of LTL formulae with implementation predictive semantics). *$u \in \Sigma^*$ is an observed word generated by the monitored program so far, $R \subseteq \Sigma^*$ is a fixed predictive words set found in the monitored program. Let v' denote the future word which is going to be generated by the monitored program. The truth value of a LTL formula φ with respect to u and R, denoted by $[u \vDash_i^R \varphi]$, is an element of $\mathbb{B}_3$ defined as follows:*

$$[u \vDash_i^R \varphi] = \begin{cases} \top & \text{if } (v' \text{ is proven} \in R \wedge uv' \notin L(\hat{A}^{\neg\varphi})) \vee (u \notin L(\hat{A}^{\neg\varphi})) \\ \bot & \text{if } (v' \text{ is proven} \in R \wedge uv' \notin L(\hat{A}^{\varphi})) \vee (u \notin L(\hat{A}^{\varphi})) \\ ? & \text{otherwise} \end{cases}$$

As a final step, we now define a (deterministic) FSM M^{φ} that outputs for each finite word u and predictive word v their associated predictive semantical evaluation. Let $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$ be the deterministic versions of $\hat{A}^{\varphi}$ and $\hat{A}^{\neg\varphi}$, which can be computed in a standard manner using the power-set construction. Then, we define $\bar{A}^{\varphi}$ as a product of $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$.

Definition 4 (Monitor M^{φ} for a LTL formula φ with predictive semantics). *Let φ be a LTL formula and $\hat{A}^{\varphi}$ be a NFA. Let $\tilde{A}^{\varphi} = (\Sigma, Q^{\varphi}, \{q_0^{\varphi}\}, \delta^{\varphi}, \tilde{F}^{\varphi})$ be a deterministic automaton with $L(\tilde{A}^{\varphi}) = L(\hat{A}^{\varphi})$, $\tilde{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, \{q_0^{\neg\varphi}\}, \delta^{\neg\varphi}, \tilde{F}^{\neg\varphi})$ be a deterministic automaton with $L(\tilde{A}^{\neg\varphi}) = L(\hat{A}^{\neg\varphi})$. The product automaton $\bar{A}^{\varphi} = \tilde{A}^{\varphi} \times \tilde{A}^{\neg\varphi}$ is a FSM $(\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{\lambda})$ where*

- $\bar{Q} = Q^{\varphi} \times Q^{\neg\varphi}$,
- $\bar{q}_0 = (q_0^{\varphi}, q_0^{\neg\varphi})$,
- $\bar{\delta}((q, q'), a) = (\delta^{\varphi}(q, a), \delta^{\neg\varphi}(q', a))$ *and*
- $\bar{\lambda} : \bar{Q} \to \mathbb{B}_3$ *is defined by*

$$\bar{\lambda}((q, q'), a) = \begin{cases} \top & \text{if } q' \notin \tilde{F}^{\neg\varphi} \\ \bot & \text{if } q \notin \tilde{F}^{\varphi} \\ ? & \text{if } q \in \tilde{F}^{\varphi} \wedge q' \in \tilde{F}^{\neg\varphi} \end{cases}$$

[1] The proof is included in section 2.3 in article [1].

The monitor M^{φ} for a LTL formula φ is the unique FSM obtained by minimizing the product automaton $\bar{A}$.

Just as proved in article [1], the following theorem holds (there are similar theorem holds for implementation predictive semantics). According to this theorem, if we want to check whether finite word u and predictive word v satisfy or violate the property φ, we just need to check whether uv drives the monitor M^{φ} into conclusive states (states labeled by $\top$ or $\bot$ according to $\bar{\lambda}$).

Lemma 3 (LTL monitor correctness). *Let φ be a LTL formula, let $M^{\varphi} = (\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{\lambda})$ be the corresponding monitor and let v be a predictive word. Then, for all $u \in \Sigma^*$, the following holds:*

$$[u \vDash_p^v \varphi] = [uv \vDash \varphi] = \bar{\lambda}(\bar{\delta}(\bar{q_0}, uv))$$

4.2 Checking Process

Once we generate the monitor from LTL formula, we can use it to check whether the monitored program's execution satisfies (driving the monitor into $\top$ state) or violates (driving the monitor into $\bot$ state) the formula. The checking process of monitor with predictive semantics is slightly different from the traditional runtime monitors. The traditional runtime monitors receive letters while in predictive semantics, they receive predictive words. When a monitor receives a predictive word, it first checks whether the run of predictive word goes through a conclusive state. If so, the monitor can give a verdict before the observed word driving it into conclusive states. If not, the monitor changes its current state $\bar{q}$ to $\bar{\delta}(\bar{q}, v)$ ($\bar{\delta}$ is the transition function, v is the predictive word). For runtime monitors in implementation predictive semantics, there are two different types of letters arising in program's execution: an ordinary letter and a predictive word. When a monitor receives an ordinary letter, it just changes its current state according to the transition function and judge whether it arrives the conclusive states. When receives a predictive word, it acts the same as the monitor in predictive semantics.

4.3 A Demonstration Example

In Java library, `Vector` is a common class to hold a variable collection of objects. The `Vector` class provides an `iterator` method to get an `Iterator` instance. The `Iterator` instance allows users to enumerate all the elements belonging to `Vector`. There is usage demand that once the `Vector` creates an `Iterator` instance, `Iterator`'s `next` method should not be called if the underlying `Vector` is modified though its own methods, such as `remove` etc, . If we use `create` to stand for the letter of creating an `Iterator` instance, `update` for modifying the `Vector`, and `next` of enumerating the `Vector`, the usage rule can be depicted as the formula: G(`create` $\rightarrow$ G(`update` $\rightarrow \neg$ F(`next`))). Using the algorithm described in the preceding section, we get the monitor in Fig. 1.

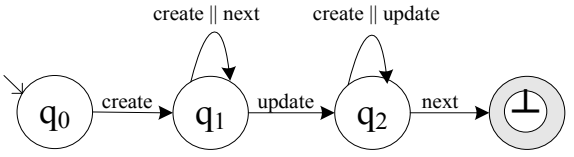


Fig. 1. A monitor generated from G(`create` → G(`update` →! F(`next`)))

If the predictive words set found in the monitored program is `update.next` (where . means concatenation). Then, the monitor can find the violation at state q_1 when it receives the predictive word. While for monitors without predictive semantics, they can only find the violation at the state labeled with $\perp$.

5 Implementation

We have implemented a prototype tool to support the implementation predictive semantics. Our tool integrates our previous work of generating monitors from LTL formulae and extends the work from Hossein Sadat-Mohtasham [4] of finding and instrumenting predictive word in monitored program.

Fig. 2 depicts a typical usage scenario of our tool. Users first specify the properties they want to monitor through LTL formulae. LTL formulae are then converted into monitors through *LTL3*[2] tool. Predictive words sets are found in monitored program according to the monitor's alphabet. Finally, monitors are injected into the monitored program by an AspectJ compiler *abc* [5] and predictive words set are injected by a modified version of *Transcut* [4]. The result program is the desired program with predictive runtime monitoring capability. It is able to detect property's satisfaction or violation before the software execution driving monitor into conclusive states.

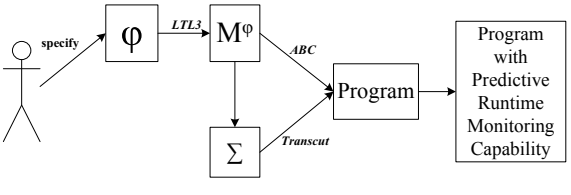


Fig. 2. A typical usage scenario of our prototype tool

5.1 Converting a LTL Formula into a Monitor

We have described the process of generating a monitor from a LTL formula in section 4. The monitor generation process is almost the same as the work

[2] http://ltl3tools.sourceforge.net/

in [1] and we have already implemented a *LTL3* tool to realize this process. We also adopt this tool to generate a monitor from a LTL formula. For example, the monitor generated from formula G(`create` → G(`update` → ¬ F(`next`))) by *LTL3* tool is depicted in Fig. 3.

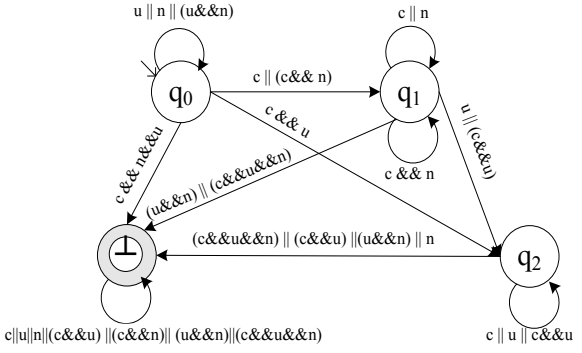


Fig. 3. A monitor generated by *LTL3* tool from G(`create` → G(`update` →! F(`next`))), where: c denotes `create`, u denotes `update` and n denotes `next`

In Fig. 3, edges are labeled with a set of APs. The || operator means there are two edges between these two states. Each is labeled by one operand. The && operator means the two operands have to be true at same time when the transition happens. For example, the edge labeled with $(c||(c\&\&n))$ between state q_0 and q_1 denotes that there are actually two edges connecting these two states. One is labeled with c and the other is labeled by $c\&\&n$.

In *LTL3* tool, the alphabet of the monitor generated from φ is 2^{AP}, where AP is the set of automatic propositions in φ. In our implementation, an automatic proposition stands for a certain kind of events in program execution and is specified by a pointcut expression in AspectJ[6]. The problem of judging whether two pointcut expressions would select a common event can be partly solved through pointcut's syntax analysis. We use this analysis to further simplify the monitor. The edges whose label are the intersection of two or more automatic propositions can be simply dropped off if the corresponding pointcut expressions can not select a common event. For instance, the edge labeled by $c\&\&n$ in Fig. 3 denotes events which belong to `create` and `next` at the same time. This is actually impossible because the two pointcut expressions for `create` and `next` (they are described in Fig. 4) can not select any common events. We also omit the self-loop edges in the initial state q_0 and the conclusive states. The self-loop edge in the initial state can be omitted as we check the program's whole trace (instead of suffix traces). The self-loop edge in the conclusive states can be omitted as there are no edges going out it. Through this kind of simplification, we get the monitor in Fig. 1.

5.2 Instrumenting Monitored Program

Once getting the monitor, we can use its alphabet to find a predictive words set from monitored program and then instrument the monitored program according to the predictive words set. As a letter in monitor's alphabet corresponds to an event in program's execution. We first briefly introduce the map between a letter in alphabet and an event in execution.

The Map Between a Letter in Alphabet and an Event in Execution. A letter in monitor's alphabet corresponds to a certain kind of event in program's execution. We use AspectJ's pointcut to build this connection. A pointcut is a set of well-defined points in the execution of the program and corresponds to a letter in alphabet. A pointcut is defined in terms of an enumeration of method signatures, wildcards and control flow relations [6]. For example, The pointcut expressions for letters (`create`, `update` and `next`) in Fig. 1 are described in Fig. 4. The first line is a pointcut expression denoting the event of calling `iterator` method on *Collection* class or its subClass and corresponding to the letter `create`. The following two `pointcut` expressions have similar meanings and denote `updage` and `next` respectively.

```
pointcut create : call(* java.util.Collection+.iterator()) && target(c) ;
pointcut update : call(* java.util.Collection+.add*(..)) && target(c) ||
         call(* java.util.Collection+.clear()) && target(c) ||
             call(* java.util.Collection+.remove*(..) ) && target(c);
pointcut next :  call(* java.util.Iterator+.next());
```

Fig. 4. A map between letters in monitor's alphabet and events in program's execution

Finding Predictive Words Set on Monitored Program's PDG Representation. While it is easy for AspectJ compiler to find the code place where a letter (event) happens, it is not easy to find the code place where the subsequent happening letter sequence (word) belongs is a predictive word (its length is bigger than 1). We find the predictive words set from monitored program's *control flow graph* (CFG) and *program dependence Graph* (PDG) representation.

A *CFG* is a directional graph in which nodes represent basic blocks. A CFG can be depicted as $< V, E >$, where V denotes the nodes set, E denotes the control flow edge set. A *region* includes the instructions in a program that execute under the same control conditions. Two nodes in a CFG are in the same region if they have the same set of control dependence predecessors [7]. Regions have an important property that makes them very ideal for finding predictive words: if the normal flow of control enters the region (which occurs only through the head node of the region), it will go through all the nodes in the region, and eventually exit through the tail node of the region. This is similar to the property of basic blocks [8] with the difference that regions can consist of non-contiguous pieces of code.

There are two different types of region: *strong region* and *weak region* [7]. We only concern strong region in this paper. Two nodes n_1 and n_2 are in the same *strong region* iff n_1 and n_2 occur the same number of times in any complete control-flow path. Thomas Ball provides an algorithm to find strong regions in $O(V + E)$ time for all CFGs [7]. Predictive words are found in strong regions. Fig. 5 illustrates the concrete finding process. In this figure, the dashed ellipse stands for a strong region. `b1`,`b2`,`b3` and `b4` are four basic blocks or statements. They belongs to the same strong region. `r1`, `r2` and `r3` are three potential finding area. Predictive words are found in these potential areas. For example, if `b1` and `b4` generate letter a and b respectively, and `b3`, `b4` do not generate monitored letters, then we can find the predictive word ab in this strong region.

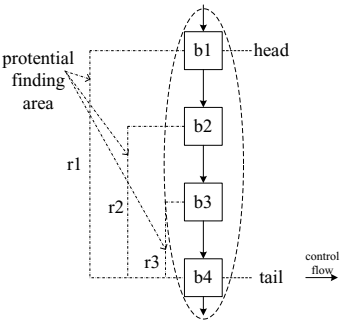


Fig. 5. Finding predictive word in strong region

The predictive word finding process is complicated by the fact that strong region are non-contiguous pieces of code. We not only have to ensure that the code fragments surrounded by the predictive word in the same strong region (such as `b2`, `b3` in Fig. 5) do not generate monitored letters, but also that the code fragments surrounded by the predictive word in CFG also do not generate monitored letters. We ensure the latter condition based on the monitored program's *PDG* representation. *PDG* is an intermediate program representation. It gives a hierarchial relation among regions. The code fragments surrounded by predictive word in CFG are the subregions of the region generating predictive word. If a predictive word is found in a region and the region has subregions, we have to check whether the subregions are surrounded by predictive word and generating monitored letters.

Our implementation for finding predictive words set in monitored program is based on a modified version of the work from Hossein Sadat-Mohtasham [4] and can only find predictive word in each method at present. The major modifications includes matching predictive words in strong region instead of weak region and checking whether the code fragments surrounded by predictive word generates monitored letters. In the future, we are planing to find predictive words across methods.

6 Experiments

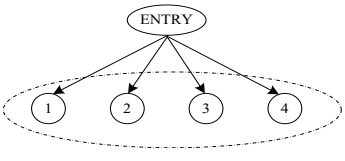
6.1 Experiments on Illustration Example

We have described a monitor in Fig. 1 and its alphabet is {`create`, `update`, `next`}. We can use this alphabet to find predictive words in monitored program's PDG representation. For instance, to the code fragment in Fig. 6a and its corresponding PDG representation in Fig. 6b, the found predictive words set is {`create.update.next`}. That is because the code generating `create` (statement 2), `update` (statement 3) and `next` (statement 4) are in the same strong region and there are no extra code surrounded by them. Fig. 7 describes another code fragment (7a) and its PDG representation (7b), whose predictive words set is also {`create.update.next`}. The code generating monitored letters (statements 2, 3, 7) are in the same strong region but are not contiguous in CFG. The code (statements 4, 5) surrounded by predictive word do not generate monitored letters. If we instrument the monitored program according to the predictive words set, the monitor can find the violation at statement 2 in Fig. 6a and statement 3 in Fig. 7a.

It is necessary to point out that the code fragments in Fig. 6a, 7a only denote code templates rather than concrete code fragments. Users can scatter any statements in these code fragments as long as they neither contain the monitored letters nor change the monitored program's PDG structure.

```
1 Vector v = new Vector(collection);
2 Iterator it = v.iterator();
3 v.add(object);
4 it.next();
```

(a) code fragment



(b) pdg representation

Fig. 6. A code fragment and its PDG representation whose predictive words set is `create.update.next`

6.2 Experiments on Real Projects

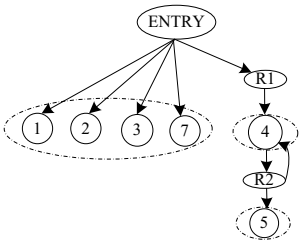
We also use the monitor in Fig. 1 to find predictive words set on several big open-source projects. These projects are chosen from Dacapo benchmark [9] and quite representative for real projects.

We summarize the project size (class number, method number) and the ratio of predictable shadows [3] and predictable regions in table 1. The row of predictable shadow's ratio describes the percentage of the shadows for letters in predictive words (compared to all the shadows for monitored letters). The fraction in parentheses list the shadow's actual number. The denominator is the number of monitored letter's shadow found in monitored program, while the

[3] A shadow is a code area which may generate monitored letter at runtime.

```
1 Vector v = new Vector(col)
2 Iterator it = v.iterator( );
3 v.add(object);
4 for(each element in col){
5     process each element ;
6 }
7 it.next();
```

(a) code fragment



(b) pdg representation

Fig. 7. A code fragment and its PDG representation whose predictive words set is also `create.update.next`

Table 1. The result of finding predictive words on real, large-scale projects

	antlr	eclipse	fop	hsqldb	bloat	lucene
class number	224	344	967	385	263	311
method number	2972	3978	6889	5859	3986	3013
predictable shadow ratio	0% (0/23)	7.92% (53/391)	24.65% (83/288)	28.23% (45/124)	17.06% (608/1495)	25% (61/224)
predictable region ratio	0% (0/23)	3.33% (22/360)	7.86% (24/229)	11.83% (14/93)	7.88% (204/1091)	7.3% (15/178)

numerator is the number of the shadow for letters which are included in predictive words. Generally speaking, the percentage is more bigger, there are more letters belongs to the predictive words and the predictive semantics are more applicable. From the table's fourth row, we can see that if the project contains a large number of shadows, there are a considerable percentage (from 7.92% to 28.23%) of letters (shadows) are predictable. The percentage of predictable letters in *antlr* project is 0%. That is because the antlr project seldom generate monitored letters (there are only 23 places generating monitored letters, and they belong to 23 different regions).

We also list the percentage of the predictable region (the region which contains predictive word) compared to the region which contains monitored letter. The percentage of predictable regions is a bit low (less than 12%). That is because we only find the predictive word in methods at present. In the future, we plans to inline more methods and find predictive word across methods.

7 Related Work

As runtime verification is becoming more and more popular to guarantee software's reliability, many runtime verification frameworks are proposed. The Tracematches [10] framework uses regular expression to specify verification property.

The Java-MOP [11] framework is designed in a way that is neutral among different logical formalisms. Regular Expressions, Context-free Grammars and LTL are all supported in Java-MOP. The Java-MaC [12] framework defines its own property specification languages: Primitive Event Definition Language(PEDL) and Meta Event Definition Language(MEDL). However, the semantics of above frameworks' property specification languages are not predictive. They all match the observed word against the property. What the future word would like is totally ignored.

Let us take *Tracematches* as an example. *Tracematches* is a popular runtime verification framework developed by Oxford and McGill university [10]. Tracematches gives the verdict when the observed letter sequence satisfies/violates the property. Hence, it catches the violation at statement *4* in Fig. 6a and at statement *7* in Fig. 7a respectively. While our implementation catches the violation at statement *2* in these two figures. Obviously, our implementation can catch the problem more earlier.

The semantics used in runtime verification which have a kind of prediction are Feng Chen etc's work [13]. Their work is mainly used in concurrent programs. The prediction in their work means an un-execution trace can be inferred from an execution path based on the program casual model. Hence, it is able to "predict" potential violations of monitored property even when the violations are not encountered in the observed executions. This kind of prediction is not about the events which will happen in the future and is totally different from our work.

8 Conclusions

In this paper, we propose predictive semantics for LTL formulae in runtime verification. We give an algorithm on how to convert a LTL formulae into a monitor and describe how to use it to check monitored program's execution with predictive semantics. We also implement a prototype tool to support predictive semantics. We use it to do some experiments on several real, large-scale projects. The result demonstrates that our predictive semantics is generally applicable in these projects. As far as we know, this is the first predictive semantics definition in runtime verification.

Acknowledgment. The work is supported by National Natural Science Foundation of China under Grant No.60970035, No.90818024, No.91018013, by the Hi-Tech Research and Development Program of China (863 Plan, 2011AA010106), and by the German BMBF Grant CHN 09/003.

References

1. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology (TOSEM) (2009) (in press)

2. Vardi, M.Y.: An Automata-Theoretic Approach to Linear Temporal Logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
3. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
4. Sadat-Mohtasham, H.: Transactional Pointcuts for Aspect-Oriented Programming. phd thesis, Department of Computer Science, University of Alberta (2010)
5. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: *abc*: An Extensible AspectJ Compiler. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 293–334. Springer, Heidelberg (2006)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
7. Ball, T.: What's in a region?: or computing control dependence regions in near-linear time for reducible control flow. ACM Lett. Program. Lang. Syst. 2, 1–16 (1993)
8. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1986)
9. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006, pp. 169–190. ACM, New York (2006)
10. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA 2005: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications, pp. 345–364. ACM Press, New York (2005)
11. Chen, F., Jin, D., Meredith, P., Roşu, G.: Monitoring oriented programming - a project overview. In: Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS 2009), pp. 72–77. ACM (2009)
12. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A runtime assurance approach for java programs. Form. Methods Syst. Des. 24, 129–155 (2004)
13. Chen, F., Şerbănuţă, T.F., Roşu, G.: jPredictor: a predictive runtime analysis tool for Java. In: ICSE 2008: Proceedings of the 30th International Conference on Software Engineering, pp. 221–230. ACM, New York (2008)