Soft Verifying Dynamic Features in Dynamic Programs

1Hou Honglin, 2Lv Jia and 3Wu Minghui
1Department of Computer Science and Engineering, Zhejiang University City College, 310015, Hangzhou, China
2College of Internet of Things Engineering, Hohai University, 213022, Changzhou, China

Abstract: Due to the dynamic language features, it is very difficult to verify the correctness of dynamic programs statically. In this study, we introduce a new method named soft verification, which combines static verification and dynamic run-time checking. The dynamic programs are divided into two parts: static components and dynamic components. Static components will be verified statically and dynamic components will be inserted into run-time checks, which will be executed at run-time to ensure the correctness of dynamic programs. We also use open temporal logic to specify the correctness of dynamic programs.

Key words: Dynamic programs, verification, temporal logic, program proof

INTRODUCTION

The advent of Web 2.0 has led to the proliferation of dynamic languages. These dynamic features include dynamically typed or weakly typed, run-time evaluation and reflection and not supporting module encapsulation enough. Due to these dynamic features, it is very difficult to verify dynamic programs statically.

We introduce a method named soft verification, which combined static verification and dynamically checking. Soft verification is a compositional method, which allows the satisfaction of a specification by a system be verified on the basis of specifications of its constituent components, without knowing the interior construction of those components.

The soft verification method model the programs as an open system, in which some components are static and will be statically verified and other components are dynamic and will be checked at run-time. The interference between static components and dynamic should be constrained, the whole program can be verified before run-time. Those run-time checks will be executed at run-time to ensure behavioral constrains specified before run-time.

In this study, we specify the correctness of dynamic programs with Open Temporal Logic (OTL) (Lv et al., 2009). We define a core language for dynamic programs and introduce a proof system to statically prove the correctness of static components and the composition of static and dynamic components.

This study consists of seven sections. The next section introduces the syntax of open temporal logic. Section 3 introduces the syntax and semantic model of open temporal logic. Section 4 represents a proof system to verify dynamic programs with the core language. Section 5 introduce our soft verification and illustrates how to soft verify dynamic programs. Section 6 discusses the related work. The last section is the conclusion and future work.

OPEN TEMPORAL LOGIC

Temporal logic (Rescher and Garson, 1968) is used to describe any system of rules and symbolism for specifying and reasoning about logic propositions qualified in terms of time. Temporal logic is the formal basis of formal verification methods such as model checking (Emerson and Clarke, 1980; Clarke and Emerson, 1981; Clarke et al., 1986; Queille and Sifakis, 1982). The semantic model of most temporal logic is a state transition system which is closed and symmetrical, which means the states of the system are fixed and same to each other. In our previous study (Lv et al., 2009), we introduce a new temporal logic named open temporal logic. The semantic model of OTL is an open system, in which come components are undetermined and variable before running and those components can added into or removed from those systems freely.

Corresponding Author: Wu Minghui, Department of Computer Science and Engineering, Zhejiang University City College, 310015, Hangzhou, China
SYNTAX OF OPEN TEMPORAL LOGIC

The execution paths in OTL are divided into two parts: static part and dynamic part. Static part specifies the static execution process and dynamic part specifies the dynamic execution process. The interference between static part and dynamic part occurs when one or both of them updates a common state. In order to distinguish the dynamic execution process from the static execution process, OTL introduces two path operators: internal part and external part.

Definition 1: Internal Part (denoted by the letter I): Static parts of a path, they are modeled as static components of the system.

Definition 2: External Part (denoted by the letter O): Dynamic parts of a path, they are modeled as dynamic components of the system.

The division of internal part and external part of the execution path ensures to specify both static part and dynamic part of the execution path. The static components can be modeled as internal parts of the execution path and dynamic components can be modeled as external parts of the execution path. Combined with other path operators inherited from CTL, OTL includes following path operators:

- A: All paths
- E: Exists a path
- AI: All internal parts in all paths
- EI: Exists an internal part in some path
- AO: All external parts in all paths
- EO: Exists an external part in some path

Since the external part of the execution path may influence the state of the whole system, some stable properties should be ensured. Similar to stable function (Xu et al., 1997) in rely-guarantee method, OTL introduces a new temporal operator named stable (denoted by the letter S) to specify the stable properties of the execution path.

Definition 3: Stable (denoted by the letter S): The system should ensure some properties when interfered by some external parts of the execution path.

For example, an open system consists of four states (S1→S4) and two execution paths S1→S2→S4 and S1→S3→S4. The execution path S1→S2 is dynamic. To keep stable properties of path S1→S2, we add constraint \( \varphi_2 \) to the dynamic path S1→S2 and the state S2 satisfy behavior constraint \( \varphi_1 \). Then we can specify this constraint by the formula \( \varphi_1 \land S \varphi_2 \) (shown as Fig. 1).

Based on these two path operators and one temporal operator, we define the syntax of OTL as follow:

Definition 4: Open temporal logic has the following syntax given in Backus Naur form:

\[
\begin{align*}
\varphi &::= \text{TRUE} \mid \text{FALSE} \mid p \mid (\neg \varphi) \mid (\varphi \land \varphi) \mid (\varphi \lor \varphi) \mid (\varphi \rightarrow \varphi) \mid \\
&\hspace{1cm} \text{AX} \varphi \mid \text{EX} \varphi \mid \text{AF} \varphi \mid \text{EF} \varphi \mid \text{AG} \varphi \mid \text{EG} \varphi \mid \\
&\hspace{1cm} \text{Ai}(\varphi) \mid \text{Ei}(\varphi) \mid \text{Ai}(\varphi, S) \mid Ei(\varphi, S) \mid \text{A}(\varphi) \mid \text{E}(\varphi) \mid \text{Ai}(\varphi) \mid \text{Ei}(\varphi) \mid \\
&\hspace{1cm} \text{Ai}(\varphi) \mid \text{Ei}(\varphi) \mid \text{Ai}(\varphi) \mid \text{Ei}(\varphi) \mid \text{Ai}(\varphi) \mid \text{Ei}(\varphi)
\end{align*}
\]

where, \( p \) is any propositional atom from some set Atoms.

Semantic of open temporal logic: The semantic model of OTL is an open system, internal actions specify the static part of execution process, while external actions specify the dynamic part of execution process. According to the division of internal actions and external actions, the open system distinguishes two kinds of transition.

Definition 5: Internal transition: A state change is caused by internal actions of the open system.

Definition 6: External transition: A state change is caused by external actions of the open system.

OTL formulae can be interpreted over open systems. Internal parts of the execution process are modeled as internal transitions in the open system and external parts of the execution process are modeled as external transitions in the open system. The stable operator is interpreted as some behavior properties of the external transitions.

Definition 7: The semantics of stable operator is interpreted as following:

\( \varphi \land S \varphi \) is true when:
1. No new external parts are added to any execution processes.
2. Some new external parts which satisfy formula \( \varphi \), are added into or removed from one execution process and the return state \( \sigma' \) satisfies formula \( \varphi \).
\( \varphi, S \varphi, \) is false when: (1) Some new external parts which do not satisfy formula \( \varphi, \) are added to any execution processes. (2) Some new external parts which satisfy formula \( \varphi, \) are added into or removed from one execution process and the return state \( \sigma' \) does not satisfy formula \( \varphi, \) any longer.

Stable operator can ensure some behavior properties of the open system. If the open system is interfered by some external actions which satisfies some behavioral constraints and if the result system still satisfies some special behavioral constraints after interfering, then the result system will ensure some behavior properties of the open system, which can ensure some correctness of the open system.

An open system \( M = (S, \rightarrow, L) \) includes a set of states \( S \) endowed with internal transitions \( \rightarrow \) or external transitions \( \rightarrow \) (binary relations on \( S \)), such that every state \( s \in S \) has some state \( s' \in S \) with \( s \rightarrow s' \) or \( s \rightarrow s' \) and a labeling function \( L : S \rightarrow P \) (Atoms). The distinction of internal transitions and external transitions leads to a compositional semantics.

**Definition 8:** Let \( M = (S, \rightarrow, L) \) be an open system model for open temporal logic, state \( s \in S \) and \( \varphi \) is an open temporal logic formula. If an external transition is added into the open system, the return state is \( s' \). The relation \( M, s = \varphi \) is defined by structural induction on \( \varphi. \)

**A CORE LANGUAGE**

Similar to the proof system of rely-guarantee method (Xu et al., 1997), we use Dijkstra's guarded command language as core language of this article.

**Syntax:** In order to prove the correctness of dynamic programs, we present a core language for dynamic programs, which syntax is defined as follows:

\[
P := \overline{x} = e \mid P_1 ; P_2 \mid P_1 \prec P_2 \mid \text{if } c \text{ then } P_1 \text{ else } P_2 \mid \text{if } c \text{ then } P_1 \text{ while } c \text{ do } P
\]

In the assignment statement, \( \overline{x} \) represents a vector of variables \( (x_1, \ldots, x_n) \) and \( e \) represents a vector of expressions \( (e_1, \ldots, e_n) \); both of them are of the same length. In a sequential composition \( \{P_1, P_2\}, P_1 \) executes first and if it terminates and \( P_2 \) then executes. In a dynamic composition \( \{P_1 \prec P_2\}, P_1 \) is the static code and \( P_2 \) is the dynamic code.

**Operational semantics:** Similar to rely-guarantee method, the program variables in our core language can be modeled as a state space. A state is a mapping from program variables to some values. A configuration is a pair \( (P, \sigma) \), where \( P \) is either a segment of program, or a special symbol \( E \) standing for the end of a program and \( \sigma \) is a state of program \( P \) (Xu et al., 1997). The actions are represented by an arrow connecting the beginning configuration and ending configuration. The actions of the dynamic programs can be divided into two kinds as follow:

- **Internal Transition** \((P, \sigma) \rightarrow (P', \sigma')\): A step from static components of the program, program \( P \) changes to \( P' \) and state \( \sigma \) changes to \( \sigma' \).
- **External excitation** \((P, \sigma) \rightarrow^e (P, \sigma')\): A step from dynamic components of the program and state \( \sigma \) changes to \( \sigma' \). Note program \( P \) does not change.

**Definition 9:** Computation: For a program \( P, \) a computation is any finite or infinite sequence:

\[
(P_0, \sigma_0) \rightarrow^e (P_1, \sigma_1) \rightarrow^e (P_2, \sigma_2) \rightarrow^e \ldots
\]

where, each action \( \rightarrow^e \) is the internal transition relation or the external excitation relation and \( P_0 = P \).

**A proof system:** In this section, we propose a proof system to verify a segment of program written in core language. Similar to Floyd-Hoare logic, if we can verify a segment of program in our core language, we can incrementally verify a program written in actual dynamic programs.

**Specification:** A specification describes conditions (called assumptions) under which the program is used and the expected behavior (called commitments) of the program when it is used under these conditions (Xu et al., 1997). Similar to the specification of rely-guarantee method (Xu et al., 1997), we introduce the rely-condition to constrain the interference of dynamic code.

**Definition 10:** Rely-condition: An assertion over two states, one is the state before dynamic code and the other is the state getting control back from dynamic code.

When reasoning about the behavior of a segment of program, we assume that any interleaved actions that the aspects may change the state of the underlying system should be within the constraints specified by rely-conditions.

The specification of a segment of program consists of three parts: pre-condition, rely-condition and post-condition. Pre-condition and rely-condition are assumptions of the program and post-condition is commitment of the program.
Informally, a program $P$ satisfies such a specification $(\text{pre}, \text{rely}, \text{post})$, denoted by the formula:

$$\text{Psat}(\text{pre}, \text{rely}, \text{post}), \quad \text{if:}$$

- $P$ is invoked in a state which satisfies $\text{pre}(\text{pre-condition})$ and
- any external excitation satisfies $\text{rely}(\text{rely-condition})$, then
- if a computation terminates, the final state satisfies $\text{post}(\text{post-condition})$

Proof rules: In this section, we present a set of the proof rules to verify a segment of programs written in our core language.

Assigning axiom: The assigning axiom is atomic, which means no dynamic code may interfere during the execution of it. But some dynamic code may happen before or after this statement, so the pre-condition $\text{pre}$ and post-condition $\text{pre} \ (x\leftarrow e)$ should be stable with respect to rely-condition rely. Formula $A(\text{pre} S \text{rely})$ means the pre-condition $\text{pre}$ will be stable when any dynamic code satisfies rely-condition rely inserted before the assignment $x \leftarrow e$. Formula $A(\text{pre} (x\leftarrow e) S \text{rely})$ means the post-condition $\text{pre} \ (x\leftarrow e)$ will be stable when any dynamic code satisfies rely-condition rely inserted after the assignment $x \leftarrow e$:

$$\frac{A(\text{pre} S \text{rely})}{A(\text{pre} (x\leftarrow e) S \text{rely})} (x \leftarrow e) \text{sat}(\text{pre, rely, pre}(x\leftarrow e))$$

Consequence rule: Consequence rules allow one to strengthen the assumptions or weaken the commitments of a specification. The program used to satisfy a specification $\text{Psat}(\text{pre}', \text{rely}', \text{post}')$. The pre-condition $\text{pre}'$, the rely-condition $\text{rely}'$ implies $\text{pre}'$ and the post-condition post implies $\text{post}'$. Then the specification of program $P$ can be replaced by $\text{Psat}(\text{pre}', \text{rely}', \text{post}')$:

$$\frac{\text{Psat}(\text{pre}', \text{rely}', \text{post}')}{\text{Psat}(\text{pre}, \text{rely}, \text{post})} \quad \text{pre} \rightarrow \text{pre}', \quad \text{rely} \rightarrow \text{rely}', \quad \text{post} \rightarrow \text{post}'$$

Sequential composition rule: Program $P_1$ satisfies $(\text{pre}, \text{rely}, g)$ and program $P_2$ satisfies $(g, \text{rely}, \text{post})$, in which the post-condition of program $P_1$ equals the pre-condition of program $P_2$. Then the result program $\{P_1, P_2\}$ satisfies $P_1; P_2 \text{sat}(\text{pre, rely, post})$:

$$\frac{P_1 \text{sat}(\text{pre, rely, g})}{P_2 \text{sat}(g, \text{rely}, \text{post})} \quad P_1; P_2 \text{sat}(\text{pre, rely, post})$$

Dynamic composition rule: The program $P_1$ satisfies $(\text{pre}, \text{rely}, \text{post})$ and the program $P_2$ satisfies $(\text{pre}_2, \text{rely}_2, \text{post}_2)$. The rely-condition of base-code $(P_1)$ implies the rely-condition of the dynamic code $(P_2)$ inserted into it and the rely-condition of $P_1$ should be stable when $P_2$ is inserted into it. The result program $P_1; P_2$ satisfies $P_1 \times P_2 \text{sat}(\text{pre}, \text{rely}, \text{post})$:

$$\frac{P_1 \text{sat}(\text{pre}, \text{rely}, \text{post})}{P_2 \text{sat}(\text{pre}_2, \text{rely}_2, \text{post}_2)} \quad \text{A(rely}, S \text{rely}_1), \ \text{rely}_1 \rightarrow \text{rely}_2, \ P_1 \times P_2 \text{sat}(\text{pre}, \text{rely}, \text{post})$$

Conditional rule: The Boolean test is atomic, which cannot be interrupted by any dynamic code, but some dynamic code may happen before or after the statement IF. The post-condition post of the statement IF is already stable according to the condition $P_1 \text{sat}(\text{pre} \land c, \text{rely}, \text{post})$. The pre-condition $\text{pre}$ of statement IF should also be stable. Then the result program if $c$ then $P_1$ else $P_2$ satisfies if $c$ then $P_1$ else $P_2 \text{sat}(\text{pre, rely, post})$:

$$\frac{P_1 \text{sat}(\text{pre} \land c, \text{rely}, \text{post})}{P_2 \text{sat}(\text{pre} \land \neg c, \text{rely}, \text{post})}, \quad \text{A(pre S rely)}) \quad \text{if c then P_1 else P_2 sat(pre, rely, post)}$$

If $P_1$ is null, the Conditional Rule is as follows:

$$\frac{P_1 \text{sat}(\text{pre} \land c, \text{rely}, \text{post})}{\text{A(pre S rely)})} \quad \text{if c then P_1 sat(pre, rely, post)}$$

Iteration rule: Similar to the statement IF, every boolean test in statement Iteration is atomic and it cannot be interrupted by any dynamic code but some dynamic code may happen before or after every iteration. So the pre-condition $\text{pre}$ of statement Iteration before ith iteration and the post-condition post of statement Iteration should be stable respect to rely-condition rely after ith iteration:

$$\frac{\text{Psat}(\text{pre} \land c, \text{rely}, \text{pre})}{\text{A(pre S rely)})}, \quad \text{pre} \land \neg c \rightarrow \text{post}, \quad \text{A(pre S rely)})} \quad \text{if c then P_1 sat(pre, rely, post)}$$

$$\frac{\text{Psat}(\text{pre} \land c, \text{rely}, \text{pre})}{\text{A(pre S rely)})} \quad \text{A(pre} \land \neg c, S \text{rely})} \quad \text{while c do P sat(pre, rely, pre} \land \neg c)$$
STATIC VERIFICATION AND RUN-TIME CHECKS

In this study we introduce a new method named Soft verification which combined static verification and dynamic run-time checking. The static components of dynamic programs will be verified before run-time and the dynamic components of dynamic programs will be specified by OTL and inserted into run-time checking code, these code run-time checking will be checked at run-time to ensure the correctness of dynamic components. We illustrate the method of softly verification with a toy example. In Web 2.0 application, the SNS application always support client supplying code, which are always written in dynamic languages. A segment of dynamic program P consists of two components (shown as Fig. 2), a static component P₁, is developed by application developer and static and another dynamic component P₂ which is supplied by client and dynamic, which is inserted into static components P₁. The specification of static component P₁ is:

\[(\text{var}1 > 1, \text{var}1 > \text{var}1', \text{var}2 > \text{var}2', \text{var}1 > 1 \land \text{var}2 > 4)\]

**Static verification:** We can use upper proof system to statically verify the correctness of dynamic program P. We should statically verify whether the static component P₁ satisfies its specification or not and whether the composition of dynamic component P₂ satisfies the rely specification of P₁, or not.

According to Eq. 1, we can infer:

\[
\begin{align*}
\text{var}1 &= \text{var}1 + 1 \land \text{var}1 > \text{var}1' \\
&\land \text{var}2 > 2, \text{var}1 > 1 \land \text{var}2 > 2)
\end{align*}
\]

and:

\[
\begin{align*}
\text{var}2 &= \text{var}2 + 2 \land \text{var}1 > 1 \land \text{var}2 > 2, \\
&\land \text{var}1 > 1 \land \text{var}2 > 4)
\end{align*}
\]

If the dynamic component P₂ is inserted into static component P₁, the rely-condition of P₁ is stable during composition which satisfies the rely-condition of P₁:

\[
A(\text{var}1 > \text{var}1' \land \text{var}2 > \text{var}2' \land \text{var}1 > 1 \land \text{var}2 > 2')
\]

So according to Eq. 4, we have:

\[
\text{P vac}(\text{var}1 > 1, \text{var}1 > 1' \land \text{var}2 > 2', \text{var}1 > 1 \land \text{var}2 > 4)
\]

The static component P₁ satisfies its specification.

![Fig. 2: A toy example](image)

![Fig. 3: Different editions of unknown P₂](image)

**Run-time checks:** According to the specification of static components, if a dynamic component satisfies rely condition rely is inserted into the static component, the execution process of dynamic component ensures the pre-condition and post-condition stable. To ensure the pre-condition and post-condition stable, some run-time checks are inserted into dynamic components, which ensure safe composition of dynamic components and static components.

At run-time, we will check the behavior constraint of unknown program P₂. The unknown program P may include different dynamic language features as following Fig. 3. If all these run-time checks satisfy the behavior constraint (var₁ > 1, var₁ > var₁', var₁ > 1, var₁ > 1), the program will be correct at run-time.

**RELATED WORK**

Researches (Zhang and Cheng, 2006; Zhang et al., 2009) In order to facilitate the development and verification of dynamically adaptive systems, they separate functional concerns from adaptive concerns. Specifically, they model a dynamically adaptive program as a collection of (non-adaptive) steady-state programs and a set of adaptations that realize transitions among steady state programs in response to environmental changes. They use Linear Temporal Logic to specify properties of the non-adaptive portions of the system and we use A-LTL (an adapt-operator extension to LTL) to concisely specify properties that hold during the adaptation process. They propose a modular model checking approach to verifying that a formal model of an adaptive program satisfies its requirements specified in LTL and A-LTL, respectively.

Researches (Jonsson and Yih-Kuen, 1996; Pasareanu et al., 1999; Giannakopoulou et al., 2004; Khatchadourian and Soundarajan, 2007) present
assume-guarantee model checking, which is a novel technique for verifying correctness properties of loosely-coupled multithreaded software systems. Assume-guarantee model checking verifies each thread of a multithreaded system separately by constraining the actions of other threads with an automatically inferred environment assumption. Separate verification of each thread allows the enumeration of the local state of only one thread at a time, thereby yielding significant savings in the time and space needed for model checking.

Articles (Khatchadourian and Soundarajan, 2007; Khatchadourian et al., 2008) use rely-guarantee approach to support verifying and modular reasoning about aspect-oriented programs. Article (Katz and Katz, 2008) uses assume-guarantee method to settle the problem of behavioral problems caused by some aspects woven into a same pointcut. They have defined semantic interference among aspects relative to their specifications and shown an effective way to detect interference or prove interference-freedom of multiple aspects in a library.

CONCLUSION

Owing to the introduction of dynamic language features, it is very difficult to statically verify the correctness of dynamic programs. In this study we introduce a method which combined static program proof and run-time behavior checking. We also use OTL to specify the correctness of dynamic program and introduce a proof system to verify the static components and composition of static and dynamic programs.

We plan to user soft verification to verify more computer system, such as adaptive systems. Since temporal logic is the formal basic of model checking, we also plan to research how to model checking dynamic programs based on OTL in our future.

ACKNOWLEDGMENTS

Our research work is partly supported by the Science Foundation of Zhejiang Province (No. 2010RS0009).

REFERENCES


