Softw Syst Model
DOI 10.1007/s10270-015-0473-1

THEME SECTION PAPER

Model-checking software library API usage rules

Fu Song[1] · **Tayssir Touili**[2]

Received: 5 November 2013 / Revised: 17 March 2015 / Accepted: 30 April 2015

Abstract Modern software increasingly relies on using third-party libraries which are accessed via application programming interfaces (APIs). Libraries usually impose constraints on how API functions can be used (API usage rules) and programmers have to obey these API usage rules. However, API usage rules often are not well documented or documented informally. In this work, we show how to use the SCTPL and SLTPL logics to precisely and formally specify API usage rules in libraries, where SCTPL/SLTPL can be seen as an extension of the branching/linear temporal logic CTL/LTL with variables, quantifiers and predicates over the stack. This allows library providers to formally describe API usage rules without knowing how their libraries will be used by programmers. We propose an automated approach to check whether programs using libraries violate API usage rules or not. Our approach consists in modeling programs as pushdown systems (PDSs) and checking API usage rules by SCTPL/SLTPL model-checking for PDSs. To make the model-checking procedure more efficient and precise, we propose an abstraction that reduces drastically the size of the program model and integrate may-alias analysis into our approach to reduce false alarms. Moreover, we characterize two sublogics rSCTPL and rSLTPL of SCTPL and SLTPL that are preserved by the abstraction. We implement our techniques in a tool and apply the tool to check several open-source programs. Our tool finds several previously unknown bugs in several programs. The may-alias analysis avoids most of the false alarms that occur using SCTPL or SLTPL model-checking techniques without may-alias analysis.

Keywords Pushdown systems · Model-checking · Software API usage rules

Communicated by Prof. Einar Broch Johnsen and Luigia Petre.

✉ Fu Song
fsong@sei.ecnu.edu.cn

Tayssir Touili
touili@liafa.univ-paris-diderot.fr

[1] Shanghai Key Laboratory of Trustworthy Computing, National Trusted Embedded Software Engineering Technology Research Center, East China Normal University, Shanghai, China

[2] LIAFA, CNRS and Université Paris Diderot, Paris, France

1 Introduction

Most modern software increasingly relies on using third-party libraries and frameworks provided by organizations in order to shorten time to market. These libraries or frameworks are accessed via Application Programming Interfaces (APIs) which are sets of library functions (called API functions) and usually impose constraints (API usage rules) on how API functions can be used. Programmers have to obey these constraints when calling API functions. However, most of API usage rules are not well documented or documented informally in the API documentation. It is easy to introduce bugs using API functions. So, it is important to formally describe and automatically check API usage rules.

Many works addressed this problem [7,25,27,28,31,33,35–38,43,50,64,65,68]. However, their approaches either cannot formally describe API usage rules in a precise way or cannot automatically check API usage rules. In this work, we propose a novel formalism that can formally and precisely specify API usage rules without knowing how API functions will be used by programmers and present an approach that can automatically verify whether a program satisfies the API usage rules or not. Our approach consists of (1) modeling

Published online: 19 May 2015

n_1 : FILE* f_1=fopen("t1","w");
n_2 : FILE* f_2=fopen("t2","w");
n_3 : FILE* f_3=fopen("t3","w");
n_4 : if(f_1) then
n_5 : fclose(f_1);
n_6 : fclose(f_3);

Fig. 1 File operations

n'_1 : FILE* f_1=fopen("t1","r");
n'_2 : if(!f_1) then{
n'_3 : fread(buf,1,4,f_1);
n'_4 : fclose(f_1);}

Fig. 2 Modified file operations

programs as pushdown systems (PDSs), since PDSs are a natural model of sequential programs [23] (the stack of PDSs stores the calling procedures which allows us to check API usage rules context sensitively), (2) formally specifying in a precise manner API usage rules in SCTPL and SLTPL and (3) automatically checking whether programs violate or not API usage rules by SCTPL and SLTPL model-checking for PDSs.

SCTPL can be seen as an extension of the CTPL logic [32] with predicates over the stack content. CTPL logic is an extension of the computation tree logic (CTL) [8] with variables and quantifiers. In CTPL, propositions can be predicates of the form $p(x_1, \ldots, x_m)$, where $x_1, \ldots, x_m$ are constants or free variables that can get values from a finite domain and can be universally or existentially quantified. CTPL can specify API usage rules without knowing how API functions will be used by programmers. For example, consider the file operation API usage rule "The file should be closed by calling the API function *fclose* whenever this file is opened by calling *fopen*". Closing opened files is very important. Indeed, longtime running programs, such as web servers, will occupy a huge amount of resources if opened files are not closed. This API usage rule can be expressed in CTL as $\psi_1 \equiv \mathbf{AG}(fopen \Longrightarrow \mathbf{EF}\, fclose)$ (note that the formula $\mathbf{AG}(fopen \Longrightarrow \mathbf{AF}\, fclose)$ does not express the above statement, since if *fopen* returns a *null*[1] file pointer, then *fclose* should not be called). However, ψ_1 cannot detect the bug in Fig. 1, where the file pointed to by f_2 will never be closed. This is due to the fact that we cannot specify the relation between the return value of *fopen* and the parameter of *fclose*. To detect this bug, one approach is to specify this rule as $\psi_2 \equiv \mathbf{AG}(\bigwedge_{i=1}^{3}(f_i = fopen \Longrightarrow \mathbf{EF}\, fclose(f_i)))$. However, this formula is too special to specify this rule in the library, since, e.g., replacing the variable f_1 by f'_1 breaks ψ_2. Using CTPL, we can specify this rule as $\psi_3 \equiv \forall x\ \mathbf{AG}(x = fopen(-,-) \Longrightarrow \mathbf{EF}\, fclose(x))$ stating that whenever a file is opened and pointed to by some variable x, it should be closed in the future, where $-$ denotes a non-important variable.

Yet, ψ_3 cannot express the constraint that *fclose* is only called when *fopen* returns a pointer to some file. Indeed, *fopen* returns a *null* pointer when the file does not exist. In this case, calling *fclose* induces an error. Therefore, the API specification should be able to express the checking of return value and *fclose* is called when the return value is not null. The properties for checking return values are called error handle in [2]. For this, we introduce an additional predicate *Test(x)* which holds at a control point n iff x is tested at the control point n. We can refine ψ_3 into $\psi_4 \equiv \forall x\ \mathbf{AG}\big(x = fopen(-,-) \Longrightarrow \mathbf{AF}(Test(x) \wedge \mathbf{EXAF}\, fclose(x))\big)$. ψ_4 states that whenever the function $fopen$ is called and x stores its return value, one has to check the return value x (i.e., *Test(x)*). After this, the file has to be closed in all the future paths. The motivation of using *Test(x)* is that we cannot predict how the return value will be checked. We therefore coarsely specify that the return value is checked. But the existential path quantifier **E** in ψ_4 cannot distinguish which path is selected to validate $\mathbf{AF}\, fclose(x)$. This may induces false alarms. Let us consider the program shown in Fig. 2, which contains a bug at line n'_4, as $fclose(f_1)$ will be called if f_1 points to null. But the program satisfies ψ_4. To overcome this problem, we will add Boolean constraints into CTPL. For this example, we refine ψ_4 to the following formula ψ_5. ψ_5 states that whenever the function call $x = fopen$ is made, whether x is 0 or not should be checked. If $x \neq 0$, then $fclose(x)$ should eventually be called in all the future paths. In a similar way, we also can specify that $fclose(x)$ should not be called if x is 0.

$$\psi_5 \equiv \forall x\ \mathbf{AG}\Big(x = fopen(-,-) \Longrightarrow \mathbf{AF}(Test(x) \wedge \big(\mathbf{EX}(x \neq 0 \Longrightarrow \mathbf{AF}\, fclose(x))\big)\Big).$$

Nonetheless, CTPL extended with Boolean constraints cannot specify stack inspection properties which are important [5]. Consider the API usage rule "Calling a function $proc_1$ in some procedure $proc$ must be followed by a call to the function $proc_2$ before the procedure $proc$ returns". This API usage rule cannot be specified in CTPL. To overcome this problem, we use the SCTPL logic [52,53] (extended with Boolean constraints) to precisely describe API usage rules. SCTPL extends CTPL with predicates over the stack. Such predicates are given in the form of regular expressions over the stack alphabet and some free variables (which can also be existentially and universally quantified). Using SCTPL, the above rule can be specified as $\forall l\ \mathbf{AG}\big((proc_1 \wedge \Gamma l \Gamma^*) \Longrightarrow \mathbf{AF}(proc_2 \wedge \Gamma^+ l \Gamma^*)\big)$, where $\Gamma l \Gamma^*$ and $\Gamma^+ l \Gamma^*$ are regular predicates. The subformula $(proc_1 \wedge \Gamma l \Gamma^*)$ expresses that

[1] Note that *null* is regarded as the constant 0.

$proc_1$ is called inside some procedure $proc$ whose return address is l (since the return addresses of the called procedures are put into the stack when executing the program). The above formula states that whenever $proc_1$ is called in some procedure $proc$ whose return address is l (ensured by $\Gamma l \Gamma^*$), a function call to $proc_2$ should be made where the return address l is still in the stack, i.e., before the procedure $proc$ returns (this is ensured by $\Gamma^+ l \Gamma^*$). Note that, in our modeling, the topmost symbol of the stack of the PDS stores the current control point and the rest of the stack stores the return addresses of the calling procedures, i.e., the procedures that have not returned yet.

Similarly, the Stack Linear Temporal Predicate Logic (SLTPL) is an extension of LTL with variables, quantifiers, Boolean constraints and predicates over the stack. SLTPL is incomparable with SCTPL. The SCTPL formula ψ_5 cannot be expressed in SLTPL, while SCTPL also cannot express some SLTPL formulas. Moveover, the complexity of SLTPL model-checking is better than the complexity of SCTPL model-checking [53,54]. From a practical point of view, since LTL is considered by some users as more intuitive than CTL [17,22], we believe that SLTPL will also be considered more intuitive than SCTPL.

It is shown in [53,54] that model-checking of SCTPL and SLTPL for PDSs is decidable. Thus, we can automatically check whether a program violates or not API usage rules by SCTPL/SLTPL model-checking for PDSs. To make the verification of API usage rules more efficient, we introduce the *procedure-cutting abstraction*, which is an abstraction that drastically reduces the size of the program model by removing some procedures that do not use the API functions specified in the SCTPL/SLTPL formula. We also consider rSCTPL and rSLTPL, two sublogics of SCTPL and SLTPL, respectively, and show that the procedure-cutting abstraction preserves all rSCTPL/rSLTPL formulas when the removed procedures are terminating. rSCTPL together with rSLTPL is sufficient to express all the API usage rules we met. The procedure-cutting abstraction makes our approach more efficient and scalable.

With SCTPL model-checking for PDSs without Boolean constraints, we carried out preliminary experiments in [55]. In our preliminary experiments, several false alarms occurred due to variable aliasing. For example, our techniques reported a false negative when checking the program shown in Fig. 3 against the following file operation property

$$\psi_6 \equiv \forall x \mathbf{A}[x = fopen(-,-) \mathcal{R} \neg fread(-,-,-,x)].$$

ψ_6 states that any file can be read by calling $fread$ only if this file is previously opened by calling $fopen$. We can see that this program does not satisfy ψ_6. To solve this problem, one has to apply context-sensitive alias analysis which computes an over-approximation of context-sensitive alias pairs, as context-insensitive alias analysis does not make any sense for this example. Thus, we integrate may-alias analysis of [44] into our techniques. In [44], may-alias pairs are computed via solving the generalized reachability problem of weighted pushdown systems [44]. May-alias result obtained by solving the generalized reachability problem of weighted pushdown systems are context-sensitive and is finitely represented by a weighted finite-state automaton. This is essential to easily integrate the resulting context-sensitive may-alias into our techniques due to the stack inspection capability of SCTPL and SLTPL model-checking. In the experiment of this work, most of the false alarms reported in [55] are avoided thanks to the integration of may-alias analysis.

```
l1 : void p1(){
l2 :    ...
l3 :    FILE* f1=fopen("t1","r");
l4 :    if(f1) then{
l5 :      stringread(buf1,f1);
l6 :      fclose(f1);
       } }
l7 : void p2(){
l8 :    ...
l9 :    FILE* f2=fopen("t2","r");
l10 :   if(f2) then{
l11 :     stringread(buf2,f2);
l12 :     fclose(f2);
       } }
l13 :void stringread(char* buf, FIlE* f){
l14 :   fread(buf,1,10,f);}
l15 :void main(){
l16 :   p1();
l17 :   p2();
l18 :}
```

Fig. 3 A simplified program from verbs

The main contributions of this paper are as follows:

1. We propose a novel approach to formally and precisely specify API usage rules using two logics SCTPL and SLTPL. SCTPL and SLTPL allow library providers to formally describe API usage rules when implementing the libraries.
2. Our approach can automatically check programs against API usage rules by SCTPL and SLTPL model-checking for PDSs and allow program developers to automated verify API usage rules without any additional inputs, nor program annotation, nor environment abstractions.
3. We propose a procedure-cutting abstraction. We show that this abstraction preserves all rSCTPL and rSLTPL formulas when the cut procedures are terminating. Our abstraction reduces drastically the size of the program model, which makes API usage rules verification more efficient.
4. We implemented our techniques in a tool and applied it to check several API usage rules on several open-source programs. Our tool was able to find several previously

unknown bugs in several well-known open-source programs such as Nssl, Verbs, Acacia+, Walksat and Getafix.

Outline. Section 2 introduces the definitions of (weighted) pushdown systems and shows how to model programs as PDSs. Section 3 introduces the definitions of SCTPL and SLTPL, gives an example that illustrates the way in which properties can be expressed in SCTPL/SLTPL and compares SCTPL/SLTPL to QBEC and finite-state automata. Section 4 briefly recalls how to perform may-alias analysis via solving the generalized pushdown reachability problem of weighted pushdown systems and shows how to integrate the may-alias result into model-checking. Section 5 describes the procedure-cutting abstraction and two sublogics rSCTPL and rSLTPL. Section 6 discusses the experimental results. The related work and the conclusion are given in Sects. 7 and 8.

This paper is the full version of [55]. In this full version, we introduce Boolean constraints into SCTPL in order to overcome the limitation of the existential path quantifier **E** and integrate may-alias analysis into model-checking to avoid false alarms that are found in our preliminary experiments [55]. These efforts make our approach more precise. Besides, we show how to use SLTPL to specify API usages which is considered as more intuitive than SCTPL in practice. We also compare SLTPL/SCTPL to two classic API specification formalisms, QBEC [37] and finite-state automata.

2 Formal model

This section recalls the definitions of pushdown systems, weighted pushdown systems and its generalized pushdown reachability problem [44]. We use the approach of [23] to model a sequential program as a pushdown system.

2.1 Pushdown systems

A *Pushdown System* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations, Γ is the stack alphabet and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules.

A configuration $\langle p, \omega \rangle$ of $\mathcal{P}$ is an element of $P \times \Gamma^*$. We write $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ instead of $((p, \gamma), (q, \omega)) \in \Delta$. The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: If $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$.

A *path* $\pi = c_0 c_1 c_2 \ldots$ of the PDS is a sequence of configurations such that c_{i+1} is an *immediate successor* of the configuration c_i, i.e., $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$, for every $i \geq 0$. Let $\pi(i)$ denote the configuration c_i and π^i denote the suffix $c_i c_{i+1} \ldots$. Let $\mathcal{C}_{\mathcal{P}}$ denote the set of all the configurations of $\mathcal{P}$.

An *idempotent semiring* $\mathcal{W}$ is a tuple $(S, \oplus, \otimes, \overline{0}, \overline{1})$, where S is a set, $\overline{0}$ and $\overline{1}$ are elements of S and the *combine operation* $\oplus$ and the *extend operation* $\otimes$ are binary operators on S such that

- $(S, \oplus)$ is a commutative monoid with neutral element $\overline{0}$, and $\oplus$ is idempotent, i.e., for all $s \in S$, $s \oplus s = s$
- $(S, \otimes)$ is a monoid with neutral element $\overline{1}$,
- $\otimes$ distributes over $\oplus$, i.e., for all $s, s_1, s_2 \in S$,

$$s \otimes (s_1 \oplus s_2) = (s \otimes s_1) \oplus (s \otimes s_2),$$
$$(s_1 \oplus s_2) \otimes s = (s_1 \otimes s) \oplus (s_2 \otimes s),$$

- $\overline{0}$ is an annihilator for $\otimes$, i.e., for all $s \in S$:

$$s \otimes \overline{0} = \overline{0} = \overline{0} \otimes s.$$

We define a binary relation $\sqsubseteq$ on the semiring S: for all $s_1, s_2 \in S$, $s_1 \sqsubseteq s_2$ iff $\exists s \in S : s_1 \oplus s = s_2$. A semiring is *bounded* if there is no infinite ascending chains in the relation $\sqsubseteq$. In a bounded semiring $(S, \oplus, \otimes, \overline{0}, \overline{1})$, $(S, \oplus)$ is a meet semilattice with no infinite ascending chains.

Definition 1 A *weighted pushdown system* (WPDS) is a tuple $\mathcal{WP} = (\mathcal{P}, \mathcal{W}, l)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS, $\mathcal{W} = (S, \oplus, \otimes, \overline{0}, \overline{1})$ is a semiring, and $l : \Delta \to S$ is a function that assigns to each transition rule in Δ an element of S.

The transition relation $\Longrightarrow \subseteq \mathcal{C}_{\mathcal{P}} \times \Delta \times \mathcal{C}_{\mathcal{P}}$ of $\mathcal{WP}$ is defined as for every transition rule $r = p\gamma \hookrightarrow p'\omega \in \Delta$ and every word $\omega' \in \Gamma^*$,

$$p\gamma\omega' \overset{r}{\Longrightarrow} p'\omega\omega'.$$

Intuitively, suppose $\mathcal{WP}$ is at the configuration $p\gamma u$ and there is a transition rule $r = p\gamma \hookrightarrow p'\omega \in \Delta$, then $\mathcal{WP}$ can move from the control state p to p' and pop the topmost symbol γ from the stack, push the word ω onto the stack by the transition rule r. The weight of this moving is $l(r)$.

Given a sequence $\sigma = r_1 \ldots r_n \in \Delta^*$ of transition rules, by abuse of notation, let $l(\sigma) = l(r_1) \otimes \cdots \otimes l(r_n)$ denote the weight of the sequence σ.

Let $\Longrightarrow^* \subseteq \mathcal{C}_{\mathcal{P}} \times \Delta^* \times \mathcal{C}_{\mathcal{P}}$ be the reachability relation defined as the smallest relation such that

- $c \overset{\epsilon}{\Longrightarrow}{}^* c$, for all $c \in \mathcal{C}_{\mathcal{P}}$;
- $c \overset{r\sigma}{\Longrightarrow}{}^* c_2$ if there exists $c_1 \in \mathcal{C}_{\mathcal{P}}$ such that $c \overset{r}{\Longrightarrow} c_1$ and $c_1 \overset{\sigma}{\Longrightarrow}{}^* c_2$.

Given two configurations c and c', let $trace(c, c')$ be the set of sequences of transition rules such that

$c \overset{\sigma}{\Longrightarrow}^* c'$ iff $\sigma \in trace(c, c')$.

Given a *regular set of configurations* (defined hereafter) C (i.e., C can be represented by a finite-state automata [13]), the *generalized pushdown successor (GPS) problem* is to find for every configuration $c \in \mathcal{C}_{\mathcal{P}}$,

$$\mathcal{F}_{\mathcal{WP}}(C, c) = \bigoplus\{l(\sigma) \mid \sigma \in trace(c', c), c' \in C\}.$$

2.2 $\mathcal{P}$-automata

Definition 2 Given a weighted pushdown system $\mathcal{WP} = (\mathcal{P}, \mathcal{W}, l)$, a $\mathcal{P}$-automaton is a structure $\mathcal{WA} = (\mathbf{A}, \mathcal{W}, l')$ such that $\mathbf{A} = (Q, \Gamma, \delta, I, Q_f)$ is a finite-state automaton, where Q is a finite set of states, $\delta \subseteq Q \times \Gamma \times Q$ a transition relation, I and Q_f are sets of initial and final states, respectively, $\mathcal{W} = (S, \oplus, \otimes, \overline{0}, \overline{1})$ is a semiring and $l' : \delta \to S$ is a labeling function that assigns to each rule in δ an element of S.

Given a sequence $r_0 \cdots r_n$ of rules of δ, by abuse of notation, let $l'(r_0 \cdots r_n) = r_n \otimes \cdots \otimes r_0$. Given a configuration $c = p\gamma_1 \cdots \gamma_n \in P \times \Gamma^*$, let $trace_{\mathcal{WA}}(c)$ denote the set of all the sequences of transitions $\{r_0 \cdots r_{n-1} \mid r_0 = (q_0, \gamma_1, q_1), \ldots, r_{n-1} = (q_{n-1}, \gamma_n, q_n) \in \delta, q_0 = p \in I, q_n \in Q_f\}$.

We define

$$\mathcal{B}_{\mathcal{WA}}(c) = \bigoplus\{l'(\sigma) \mid \sigma \in trace_{\mathcal{WA}}(c)\},$$

to be the weight of the configuration c. A configuration c is *accepted* by $\mathcal{WA}$ if $\mathcal{B}_{\mathcal{WA}}(c) \neq \overline{0}$. Let $L(\mathcal{WA})$ denote the set of all the accepted configurations of $\mathcal{WA}$. A set of configurations C is called *regular* if there a is $\mathcal{P}$-automaton such that $L(\mathcal{WA}) = C$.

Theorem 1 [45,56] *Given a WPDS $\mathcal{WP} = (\mathcal{P}, \mathcal{W}, l)$ such that $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS, $\mathcal{W} = (S, \oplus, \otimes, \overline{0}, \overline{1})$ is a bounded idempotent semiring, if C is a regular set of configurations, we can compute a $\mathcal{P}$-automaton $\mathcal{WA}$ in polynomial time such that for every configuration $c \in L(\mathcal{WA})$,*

$$\mathcal{B}_{\mathcal{WA}}(c) = \mathcal{F}_{\mathcal{WP}}(C, c).$$

2.3 From programs to pushdown systems

Given a sequential program represented by a control flow graph (CFG) such that statements are in the form of static single assignments, we construct a pushdown system using the standard approach [23] where the stack alphabet Γ corresponds to the control points of the program (i.e., the nodes of the CFG); the set P of control locations is a singleton set containing p_0 (since we do not keep information about the variables of the program); and every edge $n \xrightarrow{stmt} n'$ in the CFG is represented by the following transition rule of the PDS:

- $\langle p_0, n\rangle \hookrightarrow \langle p_0, n'\rangle$ if the statement $stmt$ is neither a function call nor a return;
- $\langle p_0, n\rangle \hookrightarrow \langle p_0, f_0 n'\rangle$, if the statement $stmt$ is a function call $y = f(p_1, \ldots, p_m)$, where f_0 is the entry point of the function f. n' is regarded as the return address;
- $\langle p_0, n\rangle \hookrightarrow \langle p_0, \epsilon\rangle$ if the statement $stmt$ is a return.

Intuitively, a configuration $\langle p_0, n\omega\rangle$ where n is a control point expresses that the run of the program is at the control point n and $\omega \in \Gamma^*$ represents the return addresses of the calling procedures. Using this translation, a run of the PDS mimics a run of the program.

3 API usage rules specification

In this section, we introduce the extensions of SCTPL [53] and SLTPL [54] with Boolean constraints and show how to specify API usage rules in these two logics.

3.1 Environments, predicates and regular variable expressions

Hereafter, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, \ldots\}$ be a finite set of abstract variables ranging over a finite domain $\mathcal{D}$. In our setting, abstract variables in $\mathcal{X}$ are used in SCTPL/SLTPL formulas. While $\mathcal{D}$ is a set of program variables occurring in the given program. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment function that assigns a value $v \in \mathcal{D}$ to each variable $x \in \mathcal{X}$ and such that $B(v) = v$ for every $v \in \mathcal{D}$. $B[x \leftarrow v]$ denotes the environment function such that $B[x \leftarrow v](x) = v$ and $B[x \leftarrow v](y) = B(y)$ for every $y \neq x$. Let $\mathcal{B}$ be the set of all the environment functions.

Let AP be a finite set of atomic propositions, $AP_{\mathcal{X}}$ be a finite set of atomic predicates in the form of $a(x_1, \ldots, x_m)$ such that $a \in AP$, $x_i \in \mathcal{X} \cup \mathcal{D}$ for every $1 \leq i \leq m$, or in the form of $x \diamond n$ such that $x \in \mathcal{X}$, n is an integer, and $\diamond \in \{==, >\}$. Let $AP_{\mathcal{D}}$ be a finite set of atomic predicates of the form $a(\alpha_1, \ldots, \alpha_m)$ such that $a \in AP$, $\alpha_i \in \mathcal{D}$ for every $1 \leq i \leq m$, or in the form of $d \diamond n$ or $\neg(d \diamond n)$ such that $d \in \mathcal{D}$, n is an integer, and $\diamond \in \{==, >\}$. $x \diamond n \in AP_{\mathcal{X}}$ is called Boolean constraint, and $d \diamond n \in AP_{\mathcal{D}}$ is called Boolean expression.

W.l.o.g., we let $\neg\neg(y \diamond n)$ denote $(y \diamond n)$, $y \neq n$ denote $\neg(y == n)$ and $y \geq n$ denote $y > n-1$. For example, $x \neq 0$ used in ψ_i in Sect. 1 denotes $\neg(x == 0)$, where $x == 0$ represents that the value of the program variable $B(x)$ is 0. For every boolean expressions $b, b' \in AP_{\mathcal{D}}$, $b \wedge b'$ is satisfiable

iff $b_1 \wedge b_1'$ is true, where b_1, b_1' are two Boolean expressions obtained from b, b' by replaced all the program variables d by some integers c_d. The satisfiability of a Boolean expression can be checked by a SAT Solver.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, let $\mathcal{R}$ be a finite set of regular variable expressions over $\mathcal{X} \cup \Gamma$ defined by:

$$e ::= \emptyset \mid \epsilon \mid a \in \mathcal{X} \cup \Gamma \mid e + e \mid e \cdot e \mid e^*.$$

The language $L(e)$ of a regular variable expression e is a subset of $P \times \Gamma^* \times \mathcal{B}$ defined inductively as follows:

- $L(\emptyset) = \emptyset$;
- $L(\epsilon) = \{(\langle p, \epsilon\rangle, B) \mid p \in P, B \in \mathcal{B}\}$;
- $L(x)$, where $x \in \mathcal{X}$ is the set $\{(\langle p, \gamma\rangle, B) \mid p \in P, \gamma \in \Gamma, B \in \mathcal{B} : B(x) = \gamma\}$;
- $L(\gamma)$, where $\gamma \in \Gamma$ is the set $\{(\langle p, \gamma\rangle, B) \mid p \in P, B \in \mathcal{B}\}$;
- $L(e_1 + e_2) = L(e_1) \cup L(e_2)$;
- $L(e_1 \cdot e_2) = \{(\langle p, \omega_1\omega_2\rangle, B) \mid (\langle p, \omega_1\rangle, B) \in L(e_1);\ (\langle p, \omega_2\rangle, B) \in L(e_2)\}$;
- $L(e^*) = \{(\langle p, \omega\rangle, B) \mid B \in \mathcal{B}$ and $\omega = \omega_1 \cdots \omega_m$, s.t. $\forall i, 1 \leq i \leq m, (\langle p, \omega_i\rangle, B) \in L(e)\}$.

For example, $(\langle p, \gamma_1\gamma_2\gamma_2\rangle, B)$ is an element of $L(\gamma_1 x^*)$ when $B(x) = \gamma_2$.

3.2 Stack computation tree predicate logic

A SCTPL formula is a CTL formula where predicates and regular variable expressions are used as atomic propositions and variables can be quantified. Regular variable expressions are used to express predicates on the stack content of the PDS. Formally, the set of *SCTPL formulas* is given by (where $x \in \mathcal{X}, a(x_1, \ldots, x_m) \in AP_{\mathcal{X}}, b \in AP_{\mathcal{X}}$ and $e \in \mathcal{R}$):

$$\varphi ::= a(x_1, \ldots, x_m) \mid e \mid b \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall x\, \varphi \mid \mathbf{EX}\varphi \mid \mathbf{E}[\varphi \mathbf{U} \varphi].$$

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ s.t. $\Gamma \subseteq \mathcal{D}$, let $\lambda : AP_{\mathcal{D}} \rightarrow 2^{\Gamma^*}$ be a labeling function that assigns a regular set of words over Γ to a predicate. Let $c \in P \times \Gamma^*$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SCTPL formula ψ in c, denoted by $c \models_\lambda \psi$, iff there exists an environment $B \in \mathcal{B}$ s.t. $c \models_\lambda^B \psi$, where $c \models_\lambda^B \psi$ is defined by induction as follows: where $\diamond \in \{==, >\}$,

- $c \models_\lambda^B a(x_1, \ldots, x_m)$ iff $\omega \in \lambda(a(B(x_1), \ldots, B(x_m)))$ and $c = \langle p, \omega\rangle$;
- $c \models_\lambda^B e$ iff $(c, B) \in L(e)$;
- $c \models_\lambda^B x \diamond n$ iff for every Boolean constraint b such that $\omega \in \lambda(b), B(x) \diamond n \wedge b$ is satisfiable, where $c = \langle p, \omega\rangle$;
- $c \models_\lambda^B \psi_1 \wedge \psi_2$ iff $c \models_\lambda^B \psi_1$ and $c \models_\lambda^B \psi_2$;
- $c \models_\lambda^B \forall x\ \psi$ iff $\forall v \in \mathcal{D},\ c \models_\lambda^{B[x \leftarrow v]} \psi$;
- $c \models_\lambda^B \neg\psi$ iff $c \not\models_\lambda^B \psi$;
- $c \models_\lambda^B \mathbf{EX}\ \psi$ iff there exists a successor c' of c s.t. $c' \models_\lambda^B \psi$;
- $c \models_\lambda^B \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ iff there exists a path $\pi = c_0 c_1 \ldots$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\exists i \geq 0,\ c_i \models_\lambda^B \psi_2$ and $\forall 0 \leq j < i, c_j \models_\lambda^B \psi_1$.

Intuitively, $c \models_\lambda^B \psi$ holds iff the configuration c satisfies ψ under the environment B. We will freely use the following abbreviations: $\mathbf{AX}\psi = \neg\mathbf{EX}(\neg\psi)$, $\mathbf{EG}\psi = \mathbf{E}[\psi \mathbf{U} false]$, $\mathbf{EF}\psi = \mathbf{E}[true \mathbf{U} \psi]$, $\mathbf{AG}\psi = \neg\mathbf{EF}(\neg\psi)$, $\mathbf{AF}\psi = \neg\mathbf{EG}(\neg\psi)$, $\mathbf{A}[\psi_1 \mathbf{U} \psi_2] = \neg\mathbf{E}[\neg\psi_2 \mathbf{U}(\neg\psi_1 \wedge \neg\psi_2)] \wedge \neg\mathbf{EG}\neg\psi_2$, $\mathbf{A}[\psi_1 \mathbf{R} \psi_2] = \neg\mathbf{E}[\neg\psi_1 \mathbf{U} \neg\psi_2]$, $\mathbf{E}[\psi_1 \mathbf{R} \psi_2] = \neg\mathbf{A}[\neg\psi_1 \mathbf{U} \neg\psi_2]$, $\psi_1 \vee \psi_2 = \neg(\neg\psi_1 \wedge \neg\psi_2)$ and $\exists x \psi = \neg\forall x \neg\psi$.

Remark 1 In the model-checking community [23], the labeling function for the program model is usually defined in the form of $\lambda : \Gamma \rightarrow 2^{AP}$ which assigns to each program location a set of atomic propositions. In this setting, the validity of atomic propositions depends only on the locations. In order to integrate the context-sensitive may-alias analysis into SCTPL/SLTPL model-checking in which the may-alias pairs depend on the calling history (cf. Sect. 4), we define the labeling function as $\lambda : AP_{\mathcal{D}} \rightarrow 2^{\Gamma^*}$ which is more general than $\lambda : \Gamma \rightarrow 2^{AP}$ (cf. [24]).

Theorem 2 *SCTPL model-checking for PDSs is decidable* [53].

Intuitively, given a SCTPL formula and a PDS, checking whether the PDS satisfies the formula or not is reduced to the emptiness problem of an alternating PDS with Büchi acceptance (ABPDS), where each valuation of variables are represented by a relation. Each computation of the ABPDS mimics the checking procedure of the SCTPL formula on the transition system of the PDS. This is an extension of the automata-theoretic approach of CTL model-checking on Kripke structures to PDSs. The emptiness problem of the ABPDS is solved by computing a kind of finite automaton by readapting the saturation procedure of [13], such that the finite automaton accepts exactly all the configurations of the ABPDS from which the ABPDS has an accepting run. More details can be found in [53].

3.3 The stack linear temporal predicate logic

A SLTPL formula is a LTL formula where predicates and regular variable expressions are used as atomic propositions and variables can be quantified. Formally, the set of *SLTPL formulas* is given by (where $x \in \mathcal{X}, a(x_1, \ldots, x_m) \in AP_{\mathcal{X}}$, $b \in AP_{\mathcal{X}}$ and $e \in \mathcal{R}$):

$$\varphi ::= a(x_1, \ldots, x_m) \mid b \mid e \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall x\, \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi.$$

The other standard operators of LTL can be expressed by the above operators similar as for SCTPL.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, let $\lambda : AP_{\mathcal{D}} \rightarrow 2^{\Gamma^*}$ be a labeling function that assigns a regular set of words over Γ to each predicate. Let $c = \langle p, \omega \rangle$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SLTPL formula ψ in c (denoted by $c \models_\lambda \psi$) iff there exists an environment $B \in \mathcal{B}$ s.t. c satisfies ψ under B (denoted by $c \models_\lambda^B \psi$). $c \models_\lambda^B \psi$ holds iff $\mathcal{P}$ has a path π starting from c s.t. π satisfies ψ under B (denoted by $\pi \models_\lambda^B \psi$), where $\pi \models_\lambda^B \psi$ is defined by induction as follows: where $\diamond \in \{==, >\}$

- $\pi \models_\lambda^B a(x_1, \ldots, x_m)$ iff $\omega\lambda\big(a(B(x_1), \ldots, B(x_m))\big)$ and $c = \langle p, \omega \rangle$;
- $\pi \models_\lambda^B e$ iff $(\pi(0), B) \in L(e)$;
- $\pi \models_\lambda^B x \diamond n$ iff for every Boolean constraint b such that $\omega \in \lambda(b), B(x) \diamond n \wedge b$ is satisfiable, where $\pi(0) = \langle p, \omega \rangle$;
- $\pi \models_\lambda^B \neg\psi$ iff $\pi \not\models_\lambda^B \psi$;
- $\pi \models_\lambda^B \psi_1 \wedge \psi_2$ iff $\pi \models_\lambda^B \psi_1$ and $\pi \models_\lambda^B \psi_2$;
- $\pi \models_\lambda^B \forall x\ \psi$ iff for every $v \in \mathcal{D}$, $\pi \models_\lambda^{B[x \leftarrow v]} \psi$;
- $\pi \models_\lambda^B \mathbf{X}\psi$ iff $\pi^1 \models_\lambda^B \psi$;
- $\pi \models_\lambda^B \psi_1 \mathbf{U} \psi_2$ iff there exists $i \geq 0$ s.t. $\pi^i \models_\lambda^B \psi_2$ and $\forall j,\ 0 \leq j < i : \pi^j \models_\lambda^B \psi_1$.

Theorem 3 [54] *SLTPL model-checking for PDSs is decidable.*

The idea is similar to the SCTPL model-checking problem for PDSs. We reduce the SLTPL model-checking for PDSs to the emptiness problem of PDSs with Büchi acceptance (BPDS), where each computation of BPDS mimics the checking procedure of the SLTPL formula on a path of the PDS. More details can be found in [54].

3.4 Extracting predicates for API specifications

API usage rules often state properties concerning the order of API function calls and return value tests. Indeed, usually, after making a call to an API function, one has to check whether the call was successful. For example, when *fopen* is called to open a file "t1", one has to make sure that the call was successful or not, i.e., that the file "t1" exists (as done in Fig. 1, Line n_4). Thus, to check API usage rules, we need to extract predicates about API function calls and return value tests. To do this, for every API function call $y = f(p_1, \ldots, p_m)$ at a control point n where y denotes the return value[2] and for every $1 \leq i \leq m$, p_i denotes the ith parameter of the function f, we add the predicate $f(p_1, \ldots, p_m, y)$ to $AP_{\mathcal{D}}$ and associate this predicate to the control point n (i.e., we let $\lambda(f(p_1, \ldots, p_m, y)) = \{n\omega \mid \omega \in \Gamma^*\}$) which denotes that the proposition $y = f(p_1, \ldots, p_m)$ holds at the control point n. By abuse of notation, such predicates $f(p_1, \ldots, p_m, y)$ will also be denoted by $y = f(p_1, \ldots, p_m)$.

For every Boolean expression b in a conditional statement (i.e., if-then-else) at a control point n such that a return value y of some function call is used in b, we add the proposition *Test(y)* in $AP_{\mathcal{D}}$ and associate this predicate to n (i.e., we let $\lambda(Test(y)) = \{n\omega \mid \omega \in \Gamma^*\}$) which denotes that y is checked ($Test(y)holds$) at the control location n. Moreover, if n_t is next control point of n when b is *true*, we associate the Boolean expression b to the control point n_t, i.e., $\lambda(b) = \{n_t\omega \mid \omega \in \Gamma^*\}$. Similar, if n_f is next control point of n when b is *false*, we associate the Boolean expression $\neg b$ to the control point n_f, i.e., $\lambda(\neg b) = \{n_f\omega \mid \omega \in \Gamma^*\}$. W.l.o.g., we suppose that the return value of some API function is immediately checked in the same procedure where the API function is called. This assumption will not restrict the usefulness of the libraries, and it is recommended to check the return value immediately after the function call. Without this assumption, the techniques also work. But with this assumption, we only need to associate, respectively, the Boolean expressions b and $\neg b$ to the next control points n_t and n_f of n.

As mentioned previously, Boolean expressions are used to differentiate the paths selected by the existential path quantifier **E**. A configuration $\langle p_0, n_t\omega \rangle$ satisfies a Boolean expression b' in a SLTPL/SLTPL formula iff $b' \wedge b$ is satisfiable for the Boolean expression b associated with n_t. Then, $\langle p_0, n_t\omega \rangle$ satisfying b' means that the path from n to n_t is selected. Conversely, the path from n to n_f is selected. The formalisms of $Test$ and Boolean expressions are motivated by the fact that in practice, a programmer has several ways to check whether a return value of an API function is good or not. Thus, it is unpredictable. For example, checking whether the return value x of calling $fopen$ is $null$ or not can be done by checking $x == null$, $x \neq null$, $x > 0$, $x == 0$, etc.

Example Let us consider the fragment of Fig. 1 and the SCTPL formula ψ_3 described in Sect. 1. Then, we have:

- $\Gamma = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, fo_0, fc_0\}$ is the stack alphabet, where n_7 is the next control point of n_6, and fo_0 (resp. fc_0) is the entry point of $fopen$ (resp. $fclose$);
- $\mathcal{R} = \emptyset$, since ψ_3 does not have any regular predicate;
- $AP = \{fopen, fclose, Test\}$ is the set of atomic propositions corresponding to the API functions in the program;
- $AP_{\mathcal{D}} = \{f_1 \neq 0, f_1 == 0, Test(f_1), fclose(f_1), fclose(f_3), f_i = fopen(\text{“}t_i\text{”}, \text{“}w\text{”}) \mid 1 \leq i \leq 3\}$ is the set of atomic predicates appearing in the program;[3]

[2] W.l.o.g., we assume that each function call has a return value assigned to some variable.

[3] We reformulate $if(f_1)$ to $if(f_1 \neq 0)$ and $if(\neg f_1)$ to $if(f_1 == 0)$.

(a)

$\lambda(f_1 = fopen(\text{“}t_1\text{”}, \text{“}w\text{”})) = \{n_1\omega \mid \omega \in \Gamma^*\}$
$\lambda(f_2 = fopen(\text{“}t_2\text{”}, \text{“}w\text{”})) = \{n_2\omega \mid \omega \in \Gamma^*\}$
$\lambda(f_3 = fopen(\text{“}t_3\text{”}, \text{“}w\text{”})) = \{n_3\omega \mid \omega \in \Gamma^*\}$
$\lambda(Test(f_1)) = \{n_4\omega \mid \omega \in \Gamma^*\}$
$\lambda(fclose(f_1)) = \{n_5\omega \mid \omega \in \Gamma^*\}$
$\lambda(fclose(f_3)) = \{n_6\omega \mid \omega \in \Gamma^*\}$
$\lambda(f_1 \neq 0) = \{n_5\omega \mid \omega \in \Gamma^*\}$

(b)

$\langle p_0, n_1\rangle \hookrightarrow \langle p_0, fo_0n_2\rangle$
$\langle p_0, n_2\rangle \hookrightarrow \langle p_0, fo_0n_3\rangle$
$\langle p_0, n_3\rangle \hookrightarrow \langle p_0, fo_0n_4\rangle$
$\langle p_0, n_4\rangle \hookrightarrow \langle p_0, n_5\rangle$
$\langle p_0, n_4\rangle \hookrightarrow \langle p_0, n_6\rangle$
$\langle p_0, n_5\rangle \hookrightarrow \langle p_0, fc_0n_6\rangle$
$\langle p_0, n_6\rangle \hookrightarrow \langle p_0, fc_0n_7\rangle$

Fig. 4 **a** The labeling function λ and **b** transition rules Δ

```
int s, c, ns;
if((s=socket(AF_INET,SOCK_STREAM,0))==-1)
  return;
if(bind(s,&s_addr,len)==-1)
  {close(s); return;}
if(listen(s,5)==-1){close(s); return;}
while(1){
  ns=accept(s,&c_addr, &size);
  do{
    recv(ns,data,256,0);
    ...
    send(ns,data,256,0);
    if(cond1){close(ns); return;}
  }while(cond2)
}
close(s);
```

Fig. 5 TCP server side

- $\mathcal{D} = \{\text{“}w\text{”}, \text{“}t_i\text{”}, f_i \mid 1 \leq i \leq 3\}$ is the finite domain
- $AP_{\mathcal{X}} = \{x = fopen(y, z), fclose(x)\}$ is the set of atomic predicates appearing in ψ_3;
- The labeling function λ is shown in Fig. 4a;
- The set of transition rules Δ of the PDS modeling this fragment is shown in Fig. 4b.

3.5 An illustrating example

To illustrate our approach, we show how to specify the API usage rules for the GNU socket library.

3.5.1 Description of the socket library

The socket library implements a generalized interprocess communication channel. It provides TCP and UDP Protocols. As shown in Fig. 5, a server-side program using the TCP protocol should first create a socket *s* by calling *socket* with SOCK_STREAM as second parameter, then bind *s* to some address by calling *bind* and listen to the address by calling *listen*. When the server receives a connection request, it will create a new socket *ns* by calling *accept*. Then, the server can communicate with the client by calling *send* and *recv* via the socket *ns*. Finally, *s* and *ns* should be destroyed by calling *close*.

```
int s;
if((s=socket(AF_INET,SOCK_STREAM,0))==-1)
  return;
...
connect(s,&s_addr,len)
do{
  send(s,data,256,0);
  ...
  recv(s,data,256,0);
}while(cond3)
close(s);
```

Fig. 6 TCP client side

```
int s;
if((s=socket(AF_INET, SOCK_DGRAM,0))==-1)
  return;
if(bind(s,&s_addr,sizeof(s_addr))==-1)
  { close(s); return; }
do{
  recvfrom(s,data,256,0,&c_addr, len);
  sendto(s,data,256,0,&c_addr,len);
}while(cond4)
close(s);
```

Fig. 7 UDP server side

```
int s;
if((s=socket(AF_INET,SOCK_DGRAM,0))==-1)
  return;
do(1){
  sendto(s,data,256,0,&addr, len);
  ...
  recvfrom(s,data,256,0,&addr, len);
 }while(cond5)
close(s);
```

Fig. 8 UDP client side

Figure 6 shows a typical application of the TCP protocol at the client side. It connects to a server by calling *connect* after creating the socket *s*. Then, it can communicate with the server by calling *send* and *recv* via the socket *s*. Finally, *s* should be destroyed by calling *close*.

The server-side program using the UDP protocol should create a socket *s* by calling *socket* with SOCK_DGRAM as second parameter as shown in Fig. 7. After that, it should bind *s* to some address by calling *bind*. Then, it can communicate with a client by calling *recvfrom* and *sendto* via *s*. Finally, the socket *s* should be closed by calling *close*. The client-side program using the UDP protocol can communicate with a server by calling *recvfrom* and *sendto* via a socket *s* after its creation. Figure 8 is a typical implementation of the UDP protocol at the client side.

Table 1 A set of API usage rules of the Socket Library in SCTPL

No.	Rule
r_1	$\forall y\, \forall l\ \mathbf{AG}\left(\left(y = socket(-,-,-) \wedge \Gamma l \Gamma^*\right) \Longrightarrow\ \mathbf{AF}\left(Test(y) \wedge \Gamma l \Gamma^* \wedge\ \mathbf{EX}\,\mathbf{AF}\, close(y)\right)\right)$
r_1'	$\forall y\, \forall l\ \mathbf{AG}\left(\left(y = socket(-,-,-) \wedge \Gamma l \Gamma^*\right) \Longrightarrow\ \mathbf{AF}\left(Test(y) \wedge \Gamma l \Gamma^* \wedge \mathbf{EX}(y \neq -1 \Longrightarrow\ \mathbf{AF}\, close(y))\right)\right)$
r_2	$\forall y\ \mathbf{A}[y = socket(-,-,-)\ \mathbf{R}\ \neg bind(y,-,-)]$
r_3	$\forall y\ \mathbf{A}[listen(y,-)\ \mathbf{R}\ \neg accept(y,-,-)]$
r_4	$\forall y\ \mathbf{A}[y = socket(-, SOCK\_STREAM,-)\ \mathbf{R}\ \neg connect(y,-,-)]$
r_5	$\forall y\ \mathbf{A}[(y = socket(-, SOCK\_STREAM,-) \wedge \mathbf{A}[bind(y,-,-)\ \mathbf{R}\ \neg listen(y,-)])\ \mathbf{R}\ \neg listen(y,-)]$
r_6	$\forall y\ \mathbf{A}[connect(y,-,-) \vee y = accept(-,-,-)\ \mathbf{R}\ \neg send(y,-,-,-)]$
r_7	$\forall y\ \mathbf{A}[connect(y,-,-) \vee y = accept(-,-,-)\ \mathbf{R}\ \neg recv(y,-,-,-)]$
r_8	$\forall y\ \mathbf{A}[y = socket(-, SOCK\_DGRAM,-)\ \mathbf{R}\ \neg(sendto(y,-,-,-,-,-) \vee recvfrom(y,-,-,-,-,-))]$
r_9	$\forall y\ \mathbf{A}[sendto(y,-,-,-,-,-) \vee bind(y,-,-)\ \mathbf{R}\ \neg recvfrom(y,-,-,-,-,-)]$

3.5.2 Specifying the socket library API usage rules in SCTPL

Table 1 shows some SCTPL formulas describing some API usage rules of the socket library. Let us consider the API usage rule "The return value of *socket* should be checked immediately after the call to *socket* is made, and after a socket is created, this socket should be destroyed in all the future paths". We specify this rule by the SCTPL formula r_1 as shown in Table 1 in our previous work [55]. r_1 states that whenever the call to *socket* is made in a procedure $proc$ whose return address is l (the regular predicate $\Gamma l \Gamma^*$ ensures that the return address of the procedure $proc$ is l), the return value stored in the variable y should be eventually checked in all the future paths (i.e., *Test(y)*) inside this procedure (this is ensured by the fact that the stack is still of the form $\Gamma l \Gamma^*$ when the test of y is made). After this test, the socket y should be eventually closed in all the future paths (this is ensured by $\mathbf{EXAF}\ close(y)$). With Boolean constraints introduced in this work, we now can improve the precision of the specification as r_1' shown in Table 1. r_1' states that whenever the call to *socket* is made in a procedure $proc$ whose return address is l, the return value stored in the variable y should be eventually checked in all the future paths (i.e., *Test(y)*) inside this procedure. After this test, the socket y should be eventually closed in all the future paths if the creation of the socket is successful, i.e., $y \neq -1$. Notice that this formula cannot be expressed in SLTPL. There are many such formulas that cannot be expressed in SLTPL. The other rules in Table 1 are explained as follows.

The formula r_2 specifies that a socket y should be created ($y = socket(-,-,-)$) prior to binding the socket y to some address ($bind(y,-,-)$), where $-$ matches any constant (i.e., a variable quantified by $\forall$). r_3 is similar to r_2. r_4 states that any occurrence of $connect(y,-)$ should be preceded by an occurrence of $y = socket(-, SOCK\_STREAM,-)$ using the TCP protocol. The formula r_5 specifies that any occurrence of listening to a socket y ($listen(y,-)$) should be preceded by an occurrence of creating the socket y using the TCP protocol ($y = socket(-, SOCK\_STREAM,-)$), and the socket y should be bound to some address ($bind(y,-,-)$) before listening. The formula r_6 states that before sending a data ($send(y,-,-,-)$) via a socket y, the socket y should be connected either to the target server at the client side ($connect(y,-,-)$) or the socket created by $y = accept(-,-,-)$ at the server side. r_7 is similar. r_8 states that the socket should be created using the UDP protocol ($y = socket(-, SOCK\_DGRAM,-)$) prior to sending ($sendto(y,-,-,-,-)$) or receiving ($recvfrom(y,-,-,-,-)$) some data using the UDP protocol. The formula r_9 specifies that before receiving ($recvfrom(y,-,-,-,-)$) some data using the UDP protocol, one has to send some data ($sendto(y,-,-,-,-)$) to the server at the client side or bind ($bind(y,-,-)$) the socket to some address at the server side. Since using the UDP protocol, no connection is created and the client sends data by specifying the target address in the third parameter of the function *sendto*. After this, the client can receive data from the server. The server can send data only after receiving the client address from some client.

3.5.3 Specifying the socket library API usage rules in SLTPL

In this section, we illustrate how to express API usage rules of the GNU socket library in SLTPL. Table 2 gives 4 SLTPL formulas of API usage rules.

The formula τ_1 states that whenever *bind* is called to bind the socket to some address in a procedure whose return address is l, the user has to check whether the binding is correct before this procedure returns. τ_2 and τ_3 are similar to τ_1. The formula τ_4 specifies that the new socket cre-

Table 2 A set of API usage rules of the Socket Library in SLTPL

No.	Rule
τ_1	$\forall y\, \forall l\, \mathbf{G}\left(y = bind(-,-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{F}\left(Test(y) \wedge \Gamma l \Gamma^*\right)\right)$
τ_2	$\forall y\, \forall l\, \mathbf{G}\left(y = listen(-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{F}\left(Test(y) \wedge \Gamma l \Gamma^*\right)\right)$
τ_3	$\forall y\, \forall l\, \mathbf{G}\left(y = connect(-,-,-) \wedge \Gamma l \Gamma^* \Longrightarrow \mathbf{F}\left(Test(y) \wedge \Gamma l \Gamma^*\right)\right)$
τ_4	$\mathbf{G}\, \forall y\, \left(y = accept(-,-,-) \Longrightarrow \mathbf{F}\, close(y)\right)$

ated by $y = accept(-,-,-)$ should be eventually closed ($close(y)$) in all the future paths.

3.5.4 Checking the API usage rules

Consider the program in Fig. 5. If *cond1* is true (Fig. 5: line 13), the socket s will never be closed. r_1 or r_1' can detect this bug by model-checking the program against r_1 or r_1'. Consider the program in Fig. 6, if the client managed to connect to a server which only supports the UDP protocol as in Fig. 7, the connection at line 5 of Fig. 6 will fail, then sending (Fig. 6: line 7) or receiving (Fig. 6: line 9) some data via the socket s will induce an error. This error can be detected by checking the SLTPL formula τ_3.

3.6 Expressiveness of SCTPL and SLTPL

In this section, we compare the expressiveness of SCTPL and SLTPL with some other well-known formalisms for API usage rule specification.

Comparison with QBEC: QBEC is a quantified binary temporal logic with equality constraints which is used to specify API usage rules [37]. We compare SLTPL/SCTPL to QBEC, as QBEC includes several classes of API specifications (cf. [37]).

In QBEC, an *event* predicate combines a procedure name with potential constraints on the parameter values or return value. Formally, an event is a tuple $t = [f, 0, \ldots, n]$ with equality constraints, f denotes the function name, the constraint $t[0] = c$ denotes the return value of f is c, $t[i] = c_i$ for $i : 1 \leq i \leq n$ denotes that the (i)th parameter of f is c_i. For example, the event $[fopen, 0, 1, 2]$ with equality constraint predicates $t[0] = f_1 \wedge t[1] = \text{“}e\text{”} \wedge t[2] = \text{“}r\text{”}$ is true at a control point iff the function $f_1 = fopen(\text{“}e\text{”}, \text{“}r\text{”})$ is called at this control point.

QBEC consists of two temporal operators: the *eventual* operator and the *alternation* operator. But the subformulas of $\longrightarrow$ and $\longleftarrow$ should be events, i.e., QBEC does not allow temporal operators nesting. Formally, $t_1 \xrightarrow{*} t_2$ (forward eventual operator) represents the rule that any occurrence of the event t_1 must eventually be followed by an occurrence of the event t_2. Similarly, $t_2 \xleftarrow{*} t_1$ (backward eventual operator) represents the rule that any occurrence of t_1 must be preceded by an occurrence of t_2. $t_1 \xrightarrow{a} t_2$ (forward alternating operator) represents the rule that any occurrence of t_1 must eventually be followed by an occurrence of t_2 and an occurrence of t_1 cannot be followed by another occurrence of t_1 before occurrence of t_2. $t_2 \xleftarrow{a} t_1$ (backward alternating operator) is defined similar. QBEC allows quantifier $\forall$ to quantify over variables that occur in an event. For example, $\forall x[x, fopen, \text{“}e\text{”}, \text{“}r\text{”}] \longrightarrow [-, fclose, x]$ specifies that for any x, $x = fopen(\text{“}e\text{”}, \text{“}r\text{”})$ should be followed by $fclose(x)$. The semantics of QBEC formulas are interpreted over a trace of a program.

The events of QBEC can be presented as predicates in SLTPL, and the eventual operator and the alternating operator can also be represented by temporal operators in SLTPL, as well as quantifier $\forall$. Thus, we can show that:

Theorem 4 *SLTPL is more expressive than QBEC.*

Proof First, we show that every QBEC formula can be expressed by an equivalent SLTPL formula. An event $[f, 0, \ldots, n]$ with a set of equality constraint predicates C can be represented by predicate $f(y_1, \ldots, y_n, y_0)$ such that for every $i : 0 \leq i \leq n$, $y_i = c$ if $t[i] = c$; otherwise, $y_i = -$.

$t_1 \xrightarrow{*} t_2$ is represented by $\mathbf{G}(q_1 \Longrightarrow \mathbf{F}q_2)$, where q_1 and q_2 are predicates corresponding to t_1 and t_2, respectively. $t_2 \xleftarrow{*} t_1$ is represented by $q_2 \mathbf{R} \neg q_1$, where q_1 and q_2 are predicates corresponding to t_1 and t_2, respectively. $t_1 \xrightarrow{a} t_2$ is represented by $\mathbf{G}(q_1 \Longrightarrow \mathbf{X}(\neg q_1 \mathbf{U} q_2))$, where q_1 and q_2 are predicates corresponding to t_1 and t_2, respectively. $t_2 \xleftarrow{a} t_1$ is represented by $(q_2 \wedge \mathbf{X}(\neg q_2 \mathbf{U} q_1))\mathbf{R} \neg q_1$, where q_1 and q_2 are predicates corresponding to t_1 and t_2, respectively. $\forall x_1 \ldots \forall x_n \psi$ such that ψ is a temporal formula of QBEC is represented by $\forall x_1 \ldots \forall x_n \psi'$, where ψ' is a SLTPL formula corresponding to the QBEC formula ψ.

Strict inclusion follows from that fact that QBEC disallows the nesting of temporal operators. For example, the formula SLTPL $\mathbf{FG}(q_1 \Longrightarrow \mathbf{X}(\neg q_1 \mathbf{U} q_2))$ cannot be expressed by any QBEC formula. □

Comparison with finite-state automata: Besides QBEC, finite-state automata and its extensions are well-adapted formalisms to specify API usage rules. The expressiveness of LTL and finite-state automata is well known that LTL is as expressive as regular star-free language (regular language is as expressive as finite-state automata). On the other hand, CTL/LTL can express liveness properties and ω-languages

that cannot be expressed in finite-state automata. To address the parameters problem of API functions, many researchers introduce variables into finite state automata, such as [7,16]. In these automata, variables are implicitly quantified by ∃ quantifier. The relation between SCTPL/SCTPL and finite-state automata with variables is similar to the one between CTL/LTL and finite state automata. Finite-state automata with variables is called *security automata* in [47], and it is concretely realized for the C language in SLAM [9] for specifying and verifying API usage rules for Windows driver programs.

4 Integrating may-alias analysis into model-checking

In our previous study [55] which checks API usage rules by SCTPL model-checking technique, we found several false alarms that occurred due to variable aliasing. In order to avoid these false alarms, we integrate may-alias analysis for single-level pointers into API usage rule checking. We use the context-sensitive may-alias analysis of [44] which computes the context-sensitive may-alias pairs via solving the GPS problem of WPDSs. In this section, we first briefly recall the approach of [44]. Then, we show how to integrate the may-alias pairs into API usage rule checking.

4.1 Context-sensitive may-alias analysis

Given a program M with its model $\mathcal{P}$, two variables x and y are *alias* (denoted by $[x, y]$) at a configuration $\langle p, n\omega\rangle$ if in some program execution they refer to the same memory location when the execution reaches the program control point n with ω as the calling history (i.e., the return addresses of the calling functions that have not yet returned).

Let V denote the set of all the variables and pointer dereferences in the given program M. W.l.o.g., we assume that all variables have different names (local variables can be prefixed by the name of the procedure that contains them) so that there are no name conflicts. Let $V^2 = V \times V$ denote the set of all the possible may-alias pairs. Let $V^2_\perp$ denote the set $V^2 \cup \{\perp\}$, where $\perp$ represents the absence of an alias pair. The set of may-alias pairs for all the configurations are computed via solving the GPS problem of WPDSs. Intuitively, the side effect of each statement on may-alias pairs in a program, called transfer function, is encoded as an element of the weight S for an idempotent semiring $(S, \oplus, \otimes, \overline{0}, \overline{1})$. Consider a control point n_1 with a set of may-alias pairs v at n_1, the statement $stmt$ with the corresponding transfer function s at n_1 leading to the next control point n_2, then $s(v)$ gives the set of may-alias pairs at the location n_2. For a path $\pi = n_1 \cdots n_i$ with the transfer functions $s_1, \ldots, s_i$ at the locations n_1 up to n_i, $s_1 \otimes \cdots \otimes s_{i-1}$ is the transfer function of the path and $(s_1 \otimes \cdots \otimes s_{i-1})(v)$ is the set of may-alias pairs at n_i after the execution of the path. Consider the set of all the paths $\pi_1, \ldots, \pi_m$ from n_1 to n_i, then $(f_1 \oplus \cdots \oplus f_m)(v)$ such that f_i is the transfer function of the path π_i gives the set of all the may-alias pairs at n_i. When s_1 is the entry point of the program with no may-alias pair denoted by $\perp$, $(f_1 \oplus \cdots \oplus f_m)(\perp)$ is the set of all the possible may-alias pairs when the program reaches the control point n_i. The transfer functions of all the possible paths are computed by solving the GPS problem of WPDSs. Then, the context-sensitive may-alias pairs can be queried from the solution of the GPS problem.

Formally, we construct a weight domain $S = (V^2_\perp \to 2^{V^2}) \cup \overline{0}$. The semiring operators are defined as follows: for every $s_1, s_2 \in S \backslash \{\overline{0}\}$, $x \in V^2_\perp$,

$$(s_1 \oplus s_2)(x) = s_1(x) \cup s_2(x)$$

$$(s_1 \otimes s_2)(x) = s_2(\perp) \cup \left(\bigcup_{y \in s_1(x)} s_2(y) \right)$$

$$\overline{1}(x) = \begin{cases} \emptyset & \text{if } x = \perp \\ \{x\} & \text{otherwise,} \end{cases}$$

where $\overline{0}$ is a special element satisfying all properties of the bounded idempotent semiring.

Using the modeling approach shown in Sect. 2, we construct a PDS model $\mathcal{P}$ of the program M, a bounded semiring $\mathcal{W} = (S, \oplus, \otimes, \overline{0}, \overline{1})$ as shown above. The transfer function of may-alias pairs for each program statement is expressed by an element s of the weight S. We associate the element s to the PDS transition rule $\langle p_0, n\rangle \hookrightarrow \langle p_0, \omega\rangle$ such that s is the transfer function of the statement at n. We do not present the transfer functions of the statements, as they are beyond the scope of this paper and can be found in [44]. To this end, we obtain a WPDS $\mathcal{WP} = (\mathcal{P}, \mathcal{W}, l)$. By Theorem 1, we get that:

Theorem 5 *Let $C = \{\langle p_0, main_0\rangle\}$ be the regular set of configurations, we can construct a $\mathcal{P}$-automaton $\mathcal{WA}$ such that for every reachable configuration c from $\langle p_0, main_0\rangle$, the set of all the may-alias pairs at the configuration c is $\mathcal{B}_{\mathcal{WA}}(c)(\perp)$.*

4.2 Updating predicate via may-alias analysis

By applying Theorem 5, we can get a $\mathcal{P}$-automaton $\mathcal{WA}$ such that for every reachable configuration c from $\langle p_0, main_0\rangle$, $\mathcal{B}_{\mathcal{WA}}(c)(\perp)$ is the set of all the may-alias pairs at the configuration c. We update the labeling function λ as follows:

> For every predicate $a(\alpha_1, \ldots, \alpha_m) \in AP_{\mathcal{D}}$ such that $n\omega \in \lambda(a(\alpha_1, \ldots, \alpha_m))$, we remove $a(\alpha_1, \ldots, \alpha_m)$ from λ, and let $n\omega' \in \lambda(a(\alpha'_1, \ldots, \alpha'_m))$, for every tuple

of program variables $(\alpha'_1, \ldots, \alpha'_m)$ such that $\forall i : 1 \leq i \leq m, [\alpha_i, \alpha'_i] \in \mathcal{B}_{\mathcal{WA}}(\langle p_0, n\omega' \rangle)$.

Intuitively, consider an API function call $f(x)$ made at a control point n, then we have $\lambda(f(x) = \{n\omega \mid \omega \in \Gamma^*\})$ before updating (cf. Sect. 3.4). Suppose that we have only the may-alias pairs $[x, x]$ and $[x, y]$ at the configuration $\langle p_0, n\omega_1 \rangle$, then the runs of the program reaching $\langle p_0, n\omega_1 \rangle$ call $f(x)$ or $f(y)$. While $f(x)$ may not be called by the runs reaching another configuration $\langle p_0, n\omega'_1 \rangle$. Therefore, we let $n\omega_1 \in \lambda(f(x))$ and $n\omega_1 \in \lambda(f(y))$.

Example Consider the example shown in Fig. 3 which does not satisfy the formula ψ_6. After extracting predicates as argued in Sect. 3.4, we have:

- $\lambda(f_1 = fopen(\text{“}t_1\text{”}, \text{“}r\text{”})) = \{l_3\omega \mid \omega \in \Gamma^*\}$,
- $\lambda(fclose(f_1)) = \{l_6\omega \mid \omega \in \Gamma^*\}$,
- $\lambda(f_2 = fopen(\text{“}t_2\text{”}, \text{“}r\text{”})) = \{l_9\omega \mid \omega \in \Gamma^*\}$,
- $\lambda(fclose(f_2)) = \{l_{12}\omega \mid \omega \in \Gamma^*\}$,
- $\lambda(fread(buf, 1, 10, f)) = \{l_{14}\omega \mid \omega \in \Gamma^*\}$.

By applying the above updating, we have:

- $\lambda(f_1 = fopen(\text{“}t_1\text{”}, \text{“}r\text{”})) = \{l_3\omega \mid \omega \in \Gamma^*\}$,
- $\lambda(fclose(f_1)) = \{l_6\omega \mid \omega \in \Gamma^*\}$,
- $\lambda(f_2 = fopen(\text{“}t_2\text{”}, \text{“}r\text{”})) = \{l_9\omega \mid \omega \in \Gamma^*\}$,
- $\lambda(fclose(f_2)) = \{l_{12}\omega \mid \omega \in \Gamma^*\}$,
- $\lambda(fread(buf_1, 1, 10, f_1)) = \{l_{14}l_6l_{17}\}$,
- $\lambda(fread(buf_2, 1, 10, f_2)) = \{l_{14}l_{12}l_{18}\}$.

Using this updated labeling function, we can see that the program in Fig. 3 satisfies the formula ψ_6. In our experiments, integration of API usage rule checking with context-sensitive may-alias analysis avoids most of false alarms.

Remark 2 In this work, we do not consider indirect function calls. To take indirect function calls into account, we could perform value-set analysis. However, this will affect the scalability of API usage rule verification.

5 rSCTPL, rSLTPL and the procedure-cutting abstraction

To make API usage rules verification more efficient, it is important to model programs by PDSs having *small size*. We propose in this section the *procedure-cutting abstraction* to drastically reduce the size of the program model. The procedure-cutting abstraction eliminates all the procedures whose runs do not call any API function specified in the given SCTPL/SLTPL formula. We characterize sublogics rSCTPL and rSLTPL of SCTPL and SLTPL that are sufficient to specify all the API usage rules that we met, and we show that the procedure-cutting abstraction preserves all rSCTPL/rSLTPL formulas.

5.1 Procedure-cutting abstraction

Given a program $\mathcal{M}$, let $Proc = \{proc_i \mid 1 \leq i \leq m\}$ be the set of procedures of $\mathcal{M}$. Each procedure $proc_i$ will generate transition rules in the PDS model. Imagine there exists some procedure $proc_j$ whose runs do not call any API function specified in the given SCTPL/SLTPL formula ψ, then removing $proc_j$ will not change the satisfiability of ψ. This means that the procedure $proc_j$ can be cut. Cutting such procedure $proc_j$ will drastically reduce the size of the PDS model. We call this *procedure-cutting abstraction*. From the PDS's point of view, a function call statement $y = proc_j(\ldots)$ at a control point n (suppose n' is the next control point of n) is represented by the transition rule $\rho = \langle p_0, n \rangle \hookrightarrow \langle p_0, e_{proc_j} n' \rangle$ where e_{proc_j} denotes the entry control point of the procedure $proc_j$. Whenever the procedure $proc_j$ can be cut, we will add the transition rule $\rho' = \langle p_0, n \rangle \hookrightarrow \langle p_0, n' \rangle$ instead of ρ. The transition rule ρ' expresses that the run from n will immediately move to n' without entering the procedure $proc_j$. By doing the procedure-cutting abstraction, the size of the stack alphabet and transition rules will be drastically reduced.

Formally, to compute the abstracted program, we proceed as follows. Let $\mathcal{M}$ be a program, a *call graph* of $\mathcal{M}$ is a tuple $G = (Proc, E, proc_0)$, where $Proc$ is a finite set of nodes denoting the procedure names of $\mathcal{M}$; $E \subseteq Proc \times Proc$ is a finite set of edges such that $(proc_i, proc_j) \in E$, denoted by $proc_i \longrightarrow proc_j$, iff $proc_j$ is called in the procedure $proc_i$; $proc_0 \in Proc$ is the initial node corresponding to the entry procedure (usually, the *main* function) of $\mathcal{M}$. A node $proc_i$ can reach the node $proc_j$ iff there exists a set of edges $proc_{k_1} \longrightarrow proc_{k_2}, \ldots, proc_{k_m} \longrightarrow proc_{k_{m+1}}$ in E such that $k_1 = i$ and $k_{m+1} = j$. Let $Op(\psi)$ denote the set of atomic propositions (i.e., API function names) used in the SCTPL or SLTPL formula ψ except the additional atomic proposition *Test*. The procedure-cutting abstraction computes the abstracted program $\mathcal{M}'$ by (1) removing all the procedures $proc \in Proc$ such that the node $proc$ cannot reach any node of $Op(\psi)$ in G (i.e., the run of $proc$ will not call any function in $Op(\psi)$) and (2) replacing each function call $y = proc(p_1, \ldots, p_m)$ by a *skip* statement, i.e., no operation statement.

Proposition 1 *Given a program $\mathcal{M}$ and a SCTPL/SLTPL formula ψ, we can compute the abstracted program $\mathcal{M}'$ in linear time.*

Proof To compute the abstracted program $\mathcal{M}'$, we first compute the set $Proc'$ of procedures that cannot be removed as follows:

1. We can first construct the call graph $G = (Proc, E, proc_0)$ by traversing the program one time. Indeed, $Proc$ is the set of all the function names in $\mathcal{M}$. For every function call statement $y = f(b_1, \ldots, b_n)$ in a procedure $proc$, we add the edge $proc \longrightarrow f$ into E.
2. Let $worklist = Proc' = Op(\psi)$.
3. Remove a node $proc$ from the $worklist$.
4. For each edge $e = proc' \longrightarrow proc \in E$: if $proc' \notin Proc'$, then we add the node $proc'$ into $worklist$ and $Proc'$, add the edge e into E'; otherwise, we jump to item 2 and continue computing until $worklist$ is empty.

During items 2–4, we only add each node $proc$ into $worklist$ at most once and the number of the nodes is at most $|Proc|$. We can also represent the edges E using a hash table where the key is the right endpoint of the edge. Thus, item 4 can be done in a constant time, and items 2–4 can be done in time $|Proc|$.

Now, we can compute $\mathcal{M}'$ as follows. We (1) remove each procedure from $\mathcal{M}$ whose name is not in $Proc'$ and (2) replace each function call $y = f(p_1, \ldots, p_m)$ in $\mathcal{M}$ such that $f \notin Proc'$ by the *skip* statement. Then, the result program is $\mathcal{M}'$. □

The procedure-cutting abstraction can drastically reduce the size of the program model. However, it cannot preserve all SCTPL formulas. Indeed, formulas using the **X** operator without any restriction are not preserved, since the procedure-cutting abstraction removes procedures in the programs and replaces some function calls by *skip*. However, formulas of the form $a(x_1, \ldots, x_m) \wedge \mathbf{EX}\phi$ and $a(x_1, \ldots, x_m) \wedge \mathbf{AX}\phi$ are preserved when ϕ is a regular predicate e or its negation $\neg e$ or a SCTPL formula using the **X** operator as in the above form. Indeed, if the predicate $a(x_1, \ldots, x_m)$ occurring in a SCTPL formula (a function call or a return value test) is made in some procedure $proc$, then all the procedures including $proc$ whose runs can reach $proc$ will not be removed by the procedure-cutting abstraction. This implies that the next control point of $a(x_1, \ldots, x_m)$ will not be removed and the stack content at the next control point in the abstracted program $\mathcal{M}'$ is the same as in $\mathcal{M}$.

Moreover, formulas using regular variable expressions (i.e., e or $\neg e$) without any restriction are not preserved. Indeed, control points in $\mathcal{M}$ satisfying e or $\neg e$ may be removed by the procedure-cutting abstraction. Thus, the runs of $\mathcal{M}'$ cannot reach these control points. However, formulas of the form $a(x_1, \ldots, x_m) \wedge e$ or $a(x_1, \ldots, x_m) \wedge \neg e$ are preserved. Since all the procedures which can reach the procedure $proc$ where $a(x_1, \ldots, x_m)$ is made are not removed, each control point in $\mathcal{M}$ satisfying $a(x_1, \ldots, x_m)$ has the same calling procedures (i.e., stack content) as in $\mathcal{M}'$. Then, a configuration of $\mathcal{M}$ satisfies $a(x_1, \ldots, x_m) \wedge e$ iff this configuration of $\mathcal{M}'$ satisfies $a(x_1, \ldots, x_m) \wedge e$.

Based on the above observations, we define rSCTPL as follows (where $a(x_1, \ldots, x_m) \in AP_{\mathcal{X}}$, $b \in AP_{\mathcal{X}}$, $x \in \mathcal{X}$, and $e \in \mathcal{R}$):

$$
\begin{aligned}
\varphi ::= {} & a(x_1, \ldots, x_m) \mid \neg a(x_1, \ldots, x_m) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
& \mid \forall x\ \varphi \mid \exists x\ \varphi \mid \mathbf{A}[\varphi \mathbf{U} \varphi] \mid \mathbf{E}[\varphi \mathbf{U} \varphi] \mid \mathbf{A}[\varphi \mathbf{R} \varphi] \\
& \mid \mathbf{E}[\varphi \mathbf{R} \varphi] \mid a(x_1, \ldots, x_m) \wedge \psi \\
\psi ::= {} & e \mid \neg e \mid \mathbf{EX}\ e \mid \mathbf{AX}\ e \mid \mathbf{EX}\neg e \mid \mathbf{AX}\neg e \mid \mathbf{EX}\varphi \mid \mathbf{AX}\varphi \\
& \mid \mathbf{EX}(b \wedge \varphi) \mid \mathbf{EX}((\neg b) \wedge \varphi) \mid \mathbf{AX}(b \wedge \varphi) \\
& \mid \mathbf{AX}(b \vee \varphi) \mid \mathbf{EX}(b \vee \varphi) \mid \mathbf{EX}((\neg b) \vee \varphi) \\
& \mid \mathbf{AX}((\neg b) \wedge \varphi) \mid \mathbf{AX}((\neg b) \vee \varphi).
\end{aligned}
$$

Intuitively, rSCTPL is a sublogic of SCTPL, where (1) the next-time operator **X** is used only to specify that a rSCTPL formula ψ or a regular predicate e or its negation $\neg e$ or a Boolean constraint b or its negation $\neg b$, holds immediately after an atomic predicate holds (i.e., an API function call is made or a return value is tested), and (2) regular predicates and their negations are conjuncted with atomic predicates; Boolean constraints and their negations are conjuncted with atomic predicates.

However, the procedure-cutting abstraction does not preserve rSCTPL formulas when a cut procedure has an infinite execution (i.e., nonterminating). For instance, let $n_1 \xrightarrow{stmt} n_2$ be an edge s.t. $stmt$ is a function call $y = f(p_1, \ldots, p_m)$ and the procedure f has an infinite execution. Suppose we replace this function call by *skip*. If n_1 and all the control locations of f do not satisfy the atomic predicate a (i.e., API function calls or return value test), while n_2 satisfies a, then the configuration $\langle p_0, n_1\omega\rangle$ of $\mathcal{M}$ satisfies $\mathbf{EG}\neg a$, but $\langle p_0, n_1\omega\rangle$ does not satisfy $\mathbf{EG}\neg a$ in $\mathcal{M}'$ due to the removal of the infinite execution. On the other hand, if n_1 and all the control locations of f do not satisfy the atomic predicate a, while n_2 satisfies the atomic predicate b, then the configuration $\langle p_0, n_1\omega\rangle$ of $\mathcal{M}'$ satisfies $\mathbf{A}[\neg a \mathbf{U} b]$ due to the removal of the infinite execution, while $\langle p_0, n_1\omega\rangle$ does not satisfy $\mathbf{A}[\neg a \mathbf{U} b]$ in $\mathcal{M}$ (since b is never true in the infinite execution). We can show the following theorem.

Theorem 6 *Let ψ be a rSCTPL formula. Let $\mathcal{M}$ be a program and $\mathcal{M}'$ be the program obtained from $\mathcal{M}$ by applying the procedure-cutting abstraction. Let $\mathcal{P}$ (resp. $\mathcal{P}'$) be the PDS modeling the program $\mathcal{M}$ (resp. $\mathcal{M}'$). If all the removed procedures are infinite execution free, then $\mathcal{P}$ satisfies ψ iff $\mathcal{P}'$ satisfies ψ.*

The proof is given in the "Appendix".

5.2 The rSLTPL logic

Similarly, we define rSLTPL as follows: (where $a(x_1, \ldots, x_m) \in AP_{\mathcal{X}}$, $b \in AP_{\mathcal{X}}$, $x \in \mathcal{X}$, and $e \in \mathcal{R}$):

$$\varphi ::= a(x_1, \ldots, x_m) \mid \neg a(x_1, \ldots, x_m) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$
$$\mid \forall x\ \varphi \mid \exists x\ \varphi \mid \varphi \mathbf{U} \varphi$$
$$\mid \varphi \mathbf{R} \varphi \mid a(x_1, \ldots, x_m) \wedge \psi$$
$$\psi ::= e \mid \neg e \mid \mathbf{X} e \mid \mathbf{X} \neg e \mid \mathbf{X} \varphi$$
$$\mid \mathbf{X}(b \wedge \varphi) \mid \mathbf{X}((\neg b) \wedge \varphi) \mid \mathbf{X}(b \vee \varphi) \mid \mathbf{X}((\neg b) \vee \varphi).$$

The intuition is similar to the one underlying the definition of rSCTPL. We can show that the procedure-cutting abstraction preserves all the rSLTPL formulas when all the removed procedures are infinite execution free, i.e., terminating.

Theorem 7 *Let ψ be a rSLTPL formula, $\mathcal{M}$ a program and $\mathcal{M}'$ the program obtained from $\mathcal{M}$ by applying the procedure-cutting abstraction. Let $\mathcal{P}$ (resp. $\mathcal{P}'$) be the PDS modeling the program $\mathcal{M}$ (resp. $\mathcal{M}'$). If all the removed procedures are infinite execution free, then $\mathcal{P}$ satisfies ψ iff $\mathcal{P}'$ satisfies ψ.*

The proof is similar to the proof of Theorem 6.

6 Experiments

We implemented our techniques in a tool for automated verifying API usage rules. Given a program $\mathcal{M}$ using some libraries which are equipped with the API usage rules specified in SCTPL and/or SLTPL, our tool automatically answers Yes or No, depending on whether the program violates the API usage rules or not.

In our implementation, we use goto-cc [34] as front end to parse ANSI-C programs into goto-cc binary programs. We implemented a translator that transforms goto-cc binary programs into pushdown systems and outputs the required predicates as discussed in Sect. 3.4. May-alias analysis is implemented based on the weighted automaton library WALi [20]. We use the SCTPL model checker of [53] and the SLTPL model checker of [54] as engines.

In our experiments, we consider several API usage rules: the socket library API usage rules and the file operation usage rules. We checked several open-source C programs against these API usage rules. All the experiments were run on a Linux platform (Fedora 13) with a 2.4 GHz CPU and 2 GB of memory. The time limit is fixed to 30 min. Our tool detected several previously unknown errors in some well-known open-source programs. The runtime consists of the time spent for parsing goto-cc binary programs, may-alias analysis and model-checking. It excludes the time for translating ANSI-C programs into goto-cc binary programs. We also run our tool without considering the procedure-cutting abstraction. We observed that the procedure-cutting abstraction significantly speeds up the analysis. Finally, we apply our tool integrated with may-alias analysis to verify the API usage rules that are reported as negative. Essentially, one can directly check API usage rules with may-alias analysis. However, not only may-alias analysis, but also SCTPL and SLTPL model-checking with regular predicates cost times and affects the scalability. Thus, we first check API usage rules without may-alias analysis. If a potential bug is found, we apply API usage rules checking with may-alias analysis confirm the reported result.

6.1 Checking the socket library API usage rules

To check the socket library API usage rules as shown in Tables 1 and 2, we checked seven open-source programs from SourceForge [46] which are written in C and use the socket library, and four generic tutorial socket programs written by Seshadri [49].

The benchmark contains the following programs. **Comserial** is a program that helps turn console application into a web-based service, by reading from TCP connections and providing commands from each connection to applications through a socket. **MrChaTTY** is a chat program that allows users to chat via UNIX terminals through sockets. **Mrhttpd** is a web server. **Nerv** is a common socket server. **Nssl** is a netcat-like program with SSL support. **Pop3client** is a mail client which reads mail in a console and connects to servers using POP3 protocol. **Ser2nets** is a program allowing network connections to remote serial ports. **TCPC**, **TCPS**, **UDPC** and **UDPS** are a TCP client, a TCP server, a UDP client and a UDP server tutorial programs, respectively.

Tables 3, 4, 5 and 6 show the results of checking the socket library API usage rules with and without using the procedure-cutting abstraction, respectively, but without may-alias analysis. The row *#LOC* gives the number of lines of the program. **Time(s)** and **Mem(MB)** give the time consumption in seconds and memory consumption in MB, respectively. The result *Proved* denotes that the program satisfies the corresponding API usage rule, *FA* denotes false alarm, and *Bug* denotes a real bug. *o.o.m.* (resp. *o.o.t.*) means run out of memory (resp. time). False alarms are confirmed by manually reviewing the source codes.

As we can see from Tables 3 and 5, there are 22 alarms including *Bug* and *FA*. We found that 12 of these alarms are real bugs and the others are false alarms. These false alarms arose from the fact that we abstract away the data. We found 12 real errors in these programs. For instance, the program **Comserial** does not call *listen* before calling *accept* in the file *passwdserver.c* when *argc* is 1. Moreover, most of these programs will not close the socket by calling *close* nor check the return values of *socket* in some paths. For example, **Comserial** does not check the return value (i.e., socket) in the files *comserver.c* and *comclient.c* before it is used. In the file *main.c*, when it fails in binding a socket to some address, **Mrhttpd** will not close a socket before the program terminates.

Table 3 Results of checking the socket library API usage rules in SCTPL with the procedure-cutting abstraction

Program #LOC	Comserial 1.0 k	MrChaTTY 1.2 k	Mrhttpd 1.4 k	Nerv 1.1 k	Nssl 1.1 k	Pop3client 1.6 k	Ser2nets 7.3 k	TCPC 70	TCPS 90	UDPC 50	UDPS 60
r_1											
Time (s)	0.08	0.26	0.29	7.94	1.24	0.41	70.53	0.01	0.01	0.01	0.01
Mem (MB)	0.24	0.44	0.66	5.94	1.44	0.58	11.63	0.09	0.13	0.06	0.06
Result	Bug	FA	Bug	FA	Bug	Bug	Bug	Bug	Bug	Bug	Bug
r_1'											
Time (s)	1.38	1.32	1.51	10.21	2.67	1.67	81.42	0.02	0.02	0.02	0.02
Mem (MB)	0.84	1.22	1.25	7.87	2.18	0.98	18.77	0.13	0.16	0.08	0.08
Result	Bug	FA	Bug	FA	Bug	Bug	Bug	Bug	Bug	Bug	Bug
r_2											
Time (s)	0.01	0.01	0.01	0.01	0.01	0.01	0.18	0.01	0.01	0.01	0.01
Mem (MB)	0.04	0.18	0.05	0.22	0.19	0.14	1.07	0.04	0.06	0.04	0.04
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_3											
Time (s)	0.06	0.01	0.01	0.01	0.01	0.01	0.20	0.01	0.03	0.01	0.01
Mem (MB)	0.15	0.18	0.05	0.19	0.01	0.01	1.12	0.01	0.10	0.01	0.01
Result	Bug	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved	Proved
r_4											
Time (s)	0.01	0.01	0.01	0.02	0.02	0.02	0.21	0.01	0.01	0.01	0.01
Mem (MB)	0.04	0.15	0.05	0.22	0.19	0.18	0.92	0.05	0.05	0.04	0.04
Result	Proved	Proved	Proved	Proved	Bug	FA	Proved	Proved	Proved	Proved	Proved
r_5											
Time (s)	0.01	0.07	0.01	0.09	0.07	0.03	1.03	0.01	0.01	0.01	0.01
Mem (MB)	0.07	0.47	0.08	0.54	0.44	0.30	2.86	0.07	0.12	0.05	0.05
Result	Proved	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved	Proved
r_6											
Time (s)	0.01	0.01	0.01	0.02	0.01	0.01	0.07	0.02	0.01	0.01	0.01
Mem (MB)	0.11	0.34	0.30	0.50	0.29	0.30	1.46	0.08	0.10	0.01	0.01
Result	Proved	FA	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved
r_7											
Time (s)	0.01	0.01	0.01	0.05	0.01	0.01	0.07	0.01	0.01	0.01	0.01
Mem (MB)	0.11	0.33	0.33	0.75	0.29	0.35	1.46	0.08	0.09	0.01	0.01
Result	Proved	FA	Proved	FA	Proved	FA	Proved	Proved	Proved	Proved	Proved
r_8											
Time (s)	0.01	0.01	0.01	0.01	0.01	0.01	0.04	0.01	0.01	0.01	0.01
Mem (MB)	0.04	0.15	0.05	0.18	0.15	0.14	0.71	0.04	0.05	0.05	0.04
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_9											
Time (s)	0.01	0.01	0.01	0.01	0.01	0.01	0.07	0.01	0.03	0.01	0.01
Mem (MB)	0.05	0.31	0.07	0.17	0.30	0.01	1.46	0.01	0.10	0.05	0.05
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved

Tables 3, 4, 5 and 6 show that model-checking using the procedure-cutting abstraction performs better. Without using the procedure-cutting abstraction, the analysis of several examples run out of time.

6.2 Checking file operation usage rules

File reading and writing are frequently used in programs. To read or write a file, a user has to correctly open the file by

Table 4 Results of checking the socket library API usage rules in SCTPL without the procedure-cutting abstraction

Program #LOC	Comserial 1.0 k	MrChaTTY 1.2 k	Mrhttpd 1.4 k	Nerv 1.1 k	Nssl 1.1 k	Pop3client 1.6 k	Ser2nets 7.3 k	TCPC 70	TCPS 90	UDPC 50	UDPS 60
r_1											
Time (s)	1.69	1.58	17.98	o.o.t.	27.37	16.33	o.o.t.	0.02	0.04	0.01	0.01
Mem (MB)	1.15	1.63	2.73	–	5.49	3.63	–	0.17	0.25	0.11	0.11
Result	Bug	FA	Bug	–	Bug	Bug	–	Bug	Bug	Bug	Bug
r_1'											
Time (s)	4.56	3.26	24.31	o.o.t.	33.45	22.53	o.o.t.	0.04	0.08	0.02	0.02
Mem (MB)	3.66	2.44	4.42	–	7.92	5.21	–	0.22	0.41	0.19	0.19
Result	Bug	FA	Bug	–	Bug	Bug	–	Bug	Bug	Bug	Bug
r_2											
Time (s)	0.05	0.03	0.16	0.48	0.10	0.04	4.38	0.01	0.01	0.01	0.01
Mem (MB)	0.04	0.52	0.91	4.91	0.82	0.95	7.05	0.07	0.11	0.05	0.06
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_3											
Time (s)	0.07	0.03	0.16	0.47	0.11	0.04	4.21	0.01	0.01	0.01	0.01
Mem (MB)	0.40	0.52	0.91	4.82	0.82	0.92	7.08	0.07	0.10	0.05	0.50
Result	Bug	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved	Proved
r_5											
Time (s)	0.02	0.01	0.06	0.48	0.11	0.39	2.34	0.01	0.01	0.01	0.01
Mem (MB)	0.31	0.40	0.68	4.92	0.83	1.31	5.99	0.08	0.09	0.05	0.50
Result	Proved	Proved	Proved	Proved	Bug	FA	Proved	Proved	Proved	Proved	Proved
r_5											
Time (s)	0.19	0.14	0.67	0.86	0.48	0.13	15.95	0.01	0.01	0.01	0.01
Mem (MB)	1.02	1.39	2.35	11.46	2.06	2.25	19.50	0.13	0.23	0.09	0.09
Result	Proved	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved	Proved
r_6											
Time (s)	0.05	0.05	0.21	0.71	0.06	0.48	0.66	0.01	0.01	0.01	0.01
Mem (MB)	0.65	1.02	1.68	10.46	1.28	2.37	9.91	0.14	0.19	0.09	0.09
Result	Proved	FA	Proved	Proved	Proved	FA	Proved	Proved	Proved	Proved	Proved
r_7											
Time (s)	0.05	0.05	0.22	0.72	0.06	0.39	0.64	0.01	0.01	0.01	0.01
Mem (MB)	0.65	1.02	1.68	10.46	1.28	2.37	9.91	0.14	0.19	0.09	0.09
Result	Proved	FA	Proved	FA	Proved	FA	Proved	Proved	Proved	Proved	Proved
r_8											
Time (s)	0.04	0.02	0.09	0.64	0.04	0.07	0.55	0.01	0.01	0.01	0.01
Mem (MB)	0.46	0.61	1.02	7.36	0.90	1.42	6.92	0.09	0.12	0.08	0.07
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
r_9											
Time (s)	0.05	0.03	0.13	0.76	0.05	0.09	0.68	0.01	0.01	0.01	0.01
Mem (MB)	0.67	0.89	1.47	10.80	1.32	2.05	10.14	0.13	0.17	0.09	0.09
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved

calling *fopen* which returns a file pointer to the file. Then the user can read from or write to that file. Finally, the file pointer should be closed by calling *fclose*.

For file operation API usage rules, we consider two rules from *stdio.h*:

$$F_1 = \forall x\ \mathbf{AG}\Big(x = fopen(-,-) \Longrightarrow \mathbf{AF}(Test(x) \wedge \big(\mathbf{EX}(x \neq 0 \Longrightarrow \mathbf{AF}\, fclose(x))\big)\Big)$$

$$F_2 = \forall\, y\ \mathbf{A}[y = fopen(-,-)\mathbf{R}\neg(fread(-,-,-,y) \vee fwrite(-,-,-,y))]$$

Table 5 Results of checking the socket library API usage rules in SLTPL with the procedure-cutting abstraction

Program #LOC	Comserial 1.0 k	MrChaTTY 1.2 k	Mrhttpd 1.4 k	Nerv 1.1 k	Nssl 1.1 k	Pop3client 1.6 k	Ser2nets 7.3 k	TCPC 70	TCPS 90	UDPC 50	UDPS 60
τ_1											
Time (s)	0.01	0.02	0.01	0.04	0.08	0.01	0.43	0.01	0.01	0.01	0.01
Mem (MB)	0.08	0.32	0.11	0.29	0.52	0.02	2.44	0.10	0.14	0.12	0.07
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
τ_2											
Time (s)	0.01	0.06	0.01	0.04	0.05	0.02	0.32	0.01	0.01	0.01	0.01
Mem (MB)	0.08	0.32	0.11	0.30	0.41	0.03	2.39	0.08	0.14	0.08	0.05
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
τ_3											
Time (s)	0.01	0.01	0.01	0.05	0.05	0.06	0.27	0.01	0.01	0.01	0.01
Mem (MB)	0.02	0.38	0.05	0.41	0.48	0.41	2.30	0.10	0.03	0.10	0.04
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
τ_4											
Time (s)	0.03	0.02	0.06	0.41	0.10	0.05	0.65	0.01	0.03	0.01	0.01
Mem (MB)	0.28	0.38	0.45	2.60	0.65	0.48	3.47	0.10	0.15	0.07	0.03
Result	Bug	Proved	Proved	Proved	Bug	Proved	Proved	Proved	Proved	Proved	Proved

Table 6 Results of checking the socket library API usage rules in SLTPL without the procedure-cutting abstraction

Program #LOC	Comserial 1.0 k	MrChaTTY 1.2 k	Mrhttpd 1.4 k	Nerv 1.1 k	Nssl 1.1 k	Pop3client 1.6 k	Ser2nets 7.3 k	TCPC 70	TCPS 90	UDPC 50	UDPS 60
τ_1											
Time (s)	2.12	2.11	10.16	0.56	4.89	1.23	452.21	0.02	0.08	0.04	0.03
Mem (MB)	2.14	1.51	2.08	3.29	2.76	1.02	24.76	0.16	0.25	0.15	0.13
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
τ_2											
Time (s)	1.32	2.67	8.24	378.18	4.21	8.67	352.14	0.02	0.03	0.02	0.02
Mem (MB)	1.24	1.23	2.01	8.56	2.02	1.72	15.11	0.12	0.21	0.08	0.09
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
τ_3											
Time (s)	1.02	0.67	4.02	358.17	3.78	19.52	223.89	0.08	0.07	0.06	0.08
Mem (MB)	0.65	0.72	1.16	9.62	1.92	2.67	13.02	0.21	0.19	0.13	0.14
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved	Proved
τ_4											
Time (s)	4.13	7.61	45.37	42.33	31.17	10.89	52.32	0.12	0.11	0.09	0.11
Mem (MB)	3.42	9.52	17.32	18.44	18.21	5.12	22.14	0.51	1.01	0.12	0.13
Result	Bug	Proved	Proved	Proved	Bug	Proved	Proved	Proved	Proved	Proved	Proved

where F_1 is as explained as the formula ψ_5 in Sect. 1. F_2 states that the user cannot read from or write to a file pointer y unless the file pointer y points to some file (i.e., has already been opened).

To evaluate these two rules, we checked the following open-source programs which use file API functions from *stdio.h*. **Verbs** is a bounded model checker [60]. **Getafix** is a symbolic model checker for recursive Boolean programs [26]. **Moped** is a model checker for pushdown systems [42]. **Acacia+** is a tool for LTL realizability and synthesis [1]. **Mist** is a solver of the coverability problem for monotonic extensions of Petri nets [41]. **Elastic** is a translator from Elastic specifications to hytech or UPPAAL language [21]. **Mckit** is a model-checking kit [39]. **TSPASS** is a fair automated theorem prover for monodic first-order temporal logic with expanding domain semantics and propositional linear-time temporal logic [57]. **Walksat**, **MiniSat** and **Ubcsat** are three SAT solvers [40,59,63].

Table 7 Results of checking the API usage rules F_1 and F_2 with the procedure-cutting abstraction

Program #LOC	Verbs 4.0 k	Getafix 11.5 k	Moped 30.3 k	Acacia+ 8.0 k	Mist 16.0 k	Elastic 15.4 k	Mckit 26.7 k	TSPASS 62.3 k	MiniSat 1.4 k	Walksat 1.4 k	Ubcsat 16.9 k
F_1											
Time (s)	1.36	0.76	13.23	0.05	0.02	3.81	–	0.74	0.02	0.24	282.45
Mem (MB)	1.72	0.58	15.61	0.20	0.22	4.25	o.o.m.	0.98	0.23	0.46	21.02
Result	Bug	Bug	Proved	Bug	Proved	Proved	–	Proved	Proved	Bug	FA
F_2											
Time (s)	0.08	0.29	9.67	0.01	0.26	0.89	23.60	0.01	0.01	0.01	0.06
Mem (MB)	0.50	0.84	10.26	0.09	0.90	2.94	15.00	0.27	0.27	0.13	0.89
Result	FA	Proved	FA	Proved	FA	FA	Proved	Proved	FA	Proved	Proved

Table 8 Results of checking the API usage rules F_1 and F_2 without the procedure-cutting abstraction

Program #LOC	Verbs 4.0 k	Getafix 11.5 k	Moped 30.3 k	Acacia+ 8.0 k	Mist 16.0 k	Elastic 15.4 k	Mckit 26.7 k	TSPASS 62.3 k	MiniSat 1.4 k	Walksat 1.4 k	Ubcsat 16.9 k
F_1											
Time (s)	1654.76	404.63	0.06	198.52	51.23	o.o.t.	–	o.o.t.	38.24	6.06	o.o.t.
Mem (MB)	45.81	32.56	18.76	27.44	31.45	–	o.o.m.	–	7.87	5.18	–
Result	Bug	Bug	Proved	Bug	Proved	Proved	–	Proved	Proved	Bug	FA
F_2											
Time (s)	0.81	6.19	3.49	0.45	924.14	0.76	375.85	o.o.t.	0.42	0.08	5.80
Mem (MB)	8.66	8.46	21.02	5.74	33.95	8.11	76.82	–	3.10	1.60	27.70
Result	FA	Proved	FA	Proved	FA	FA	Proved	Proved	FA	Proved	Proved

Tables 7 and 8 show the results of checking these programs against F_1 and F_2 with and without the procedure-cutting abstraction. The results confirm the significance of the procedure-cutting abstraction. As shown in Table 7, we found that **Verbs**, **Getafix**, **Acacia+** and **MiniSat** have real errors. For example, in the file *main.c*, **Verbs** does not close an opened file by calling *fclose* before the program terminates. Moreover, in the files *issat.c*, *main.c* and *util.c*, a file pointer is used without checking whether it is *null* or not (i.e., whether the file exists or not). **Acacia+**, **Walksat** and **Getafix** do not close opened files which are opened in *main.c*, *walksat.c*, *bpsuspend.y* and *bp.y*, respectively.

6.3 Checking API usage rules with may-alias analysis

To assess the improvement of accuracy and reduction of scalability of may-alias analysis in API usage rules checking, we apply our tool with may-alias analysis to check the false alarms reported in Sects. 6.1 and 6.2. The results are shown in Table 9. Checking API usage rules with may-alias analysis allows us to avoid 14 false alarms. In the other cases, one runs out of time. However, the unique false alarm cannot be avoided even using our may-alias analysis. After a detailed manual analysis, we found that this false alarm is due to the pointers to pointers which cannot be handled by our may-alias analysis for single-level pointers.

7 Related work

There has been a lot of works on API usage rules specification and checking such as [2,3,7,10,25,27,28,31,33,35–38,43,50,64–66,68]. However, all these works cannot specify context-sensitive specifications, whereas our approach can.

Some tools dedicated to software model-checking were used to check API usage rules for device drivers, such as SDV [9] and DDVerify [66], and used to verify security-critical applications in which security vulnerabilities are expressed as safety properties over API functions [15,16]. But these tools can only check safety properties. One could apply some techniques [12,18,48] that reduce the liveness properties into safety properties and then apply existing tools. Other works on software model-checking, such as [11,14,29,61,62], also could be applied to check API usage rules. All these works are aimed at either performing a comprehensive verification of programs which limit the scalability or do not consider any data dependencies which may introduce many false alarms, while our work addresses to the API usage rule specifications from the library developers's point of view and to a scal-

Table 9 Results of checking the API usage rules with the may-alias analysis

Program&Rule	MrChaTTY&r_1	MrChaTTY&r_6	MrChaTTY&r_7	Nerv& r_1	Nerv& r_7	Nssl& r_3	Nssl& r_5	Pop3client& r_4
Time (s)	10.28	4.26	4.27	319.22	21.01	4.32	3.11	8.43
Memory (MB)	17.55	13.23	13.19	250.23	30.20	4.40	17.20	7.98
Result	Proved	Proved	Proved	Proved	FA	Proved	Proved	Proved
Program&Rule	Pop3client& r_6	Pop3client& r_7	Verbs&F_2	Moped& F_2	Mist& r_2	Elastic& F_2	MiniSat& r_2	16.9 k& F_1
Time (s)	4.22	4.23	3.42	392.08	9.21	37.43	2.18	o.o.t.
Memory (MB)	13.07	13.12	22.08	421.32	37.21	119.87	10.43	–
Result	Proved	Proved	Proved	Proved	Proved	Proved	Proved	–

able but yet precise enough approach for automated checking these API usage rules from a practical point of view.

Code contracts introduced in [25] can specify pre/post-conditions and invariants for each API function. Programmers have to make sure that a precondition (resp. post-condition) holds at the entry (resp. exit) of each API function and that invariants always hold inside the API function. These code contracts can be verified via either runtime checking or static checking at compile time. However, they cannot specify relations between API functions which are often used in API usage rules.

Mining-based methods are proposed [2,3,7,27,28,33,35,36,38,50,65,68] to discover API usage rules from executing traces or source codes, where API usage rules are represented by some patterns or finite automata. One can apply model-checking techniques to check whether programs violate or not API usage rules represented by patterns or finite automata. All these works cannot specify data dependencies between API functions' parameters and return values of API functions, resulting in imprecise API usage rule specifications. Variables are introduced into finite automata to specify data dependencies between API functions in [7,31]. However, these works cannot express CTL-like properties and do not show how to check whether programs violate or not API usage rules represented by finite automata equipped with variables.

A class of temporal properties, called QBEC, is used to specify API usage rules using at most one temporal operator [37]. We show that SCTPL is more expressive than QBEC. Indeed, all the temporal operators in QBEC can be expressed by SCTPL formulas. Ramanathan et al. propose a formalism in [43] to specify data dependence between API functions. However, they only consider mining preconditions of API functions rather than verification. CTL extended with variables is proposed to specify API usage rules in [64]. This work cannot specify context-sensitive specifications which is important for API usage rules.

Alur et al. introduce some nested words/trees related formalisms to specify program properties, e.g., [4–6]. These formalisms allow to express pre-/post-conditions on procedures, stack inspection properties, call-return matchings, etc. that cannot be equivalently expressed in CTL, LTL or ω-regular languages. Regular predicates and Boolean constraints in SCTPL and SLTPL can specify stack inspection properties and post conditions on procedures, but pre conditions and call-return matchings on procedures cannot be expressed in SCTPL and SLTPL. Their works have not yet considered how to express API usage rules in nested words/trees related formalisms. But it is non-trivial to do this, as API usage rules usually involve the data dependencies between API function calls.

SCTPL and SLTPL are introduced in our previous work [53,54], in which SCTPL and SLTPL are used to express malicious behaviors and model-checking is applied to detect malwares. SCTPL/SLTPL is as expressive as CTL/LTL with regular valuations [24,51]. In [53,54], we have shown that SCTPL/SLTPL model-checking for PDSs is more efficient than CTL/LTL model-checking with regular valuations. This work is not a trivial adaptation of the results of [53,54]. Concerning on API usage rules specifications and verification, we introduce Boolean constraints into SCTPL and SLTPL which can specify API usage rules in a more precise way. To avoid false alarms, we combine SCTPL/SLTPL model-checking and context-sensitive may-alias analysis. These efforts avoid almost all of the false alarms we found in our previous work [55]. To make the tool more scalable, we propose the procedure-cutting abstraction that reduces drastically the size of the program model and makes the verification more efficient.

The sublogics rSCTPL and rSLTPL we proposed are similar as stutter-closed fragments of temporal logics (e.g., CTL, LTL) [8]. Stutter-closed fragments of temporal logics are preserved by partial order reductions. rSCTPL/rSLTPL and stutter-closed fragments of temporal logics differ in the next-time temporal operators. In general, every stutter-invariant propositional linear temporal property is expressible without the next-time operators. However, as shown in this work, we need the next-time operators in order to precisely express API usage rules, especially for specification of the error handle [2] of the API function call. Therefore, rSCTPL and rSLTPL are not preserved under stuttering due to next-time operators. However, stutter-closed fragments of temporal logics CTL and LTL will be preserved under the procedure-cutting abstraction.

The procedure-cutting abstraction is similar to program slicing [19,30,58,67]. Program slicing is a technique to extract relevant program fragments from a given program with respect to some criterion (typically consisting of a program point and a subset of program variables) and is widely used in program debugging, software testing and so on. The program fragments computed by program slicing have a direct or indirect effect on the values of the slicing criterion. Our procedure-cutting abstraction extracts fragments from a program with respect to the given rSLTPL/rSCTPL specification. Our abstraction preserves rSLTPL/rSCTPL formulas, while program slicing does not due to the next-time operator **X** [19,30].

8 Conclusion and future work

We showed how to use SCTPL and SLTPL to formally and precisely specify API usage rules in libraries without knowing how the libraries will be used by programmers. We proposed an approach to automatically verify programs against API usage rules. It involves the procedure-cutting

abstraction that reduces drastically the size of the program model and makes the verification more efficient. Moreover, we characterize two sublogics rSCTPL and rSLTPL of SCTPL and SLTPL respectively. rSCTPL and rSLTPL are preserved by the abstraction. rSCTPL and rSLTPL are sufficient to precisely specify all the API usage rules we met. We implemented our techniques in a tool that allowed us to detect several unknown errors in well-known open-source programs, such as Nssl, Verbs, Acacia+, Walksat and Getafix.

However, as said previously, most of API usage rules are not well documented or explicitly stated. Formalizing API usages for third-party libraries is a challenge. There are many works studying how to discover API usages from existing source codes (e.g., [2,7]). We plan to integrate API mining techniques into our tool and construct an API usage database.

Acknowledgments We gratefully acknowledge the editor and anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work was partially supported by Shanghai Pujiang Program (No. 14PJ1403200), NSFC Projects (Nos. 61402179, 91418203), Shanghai ChenGuang Program (No. 13CG21), Shanghai Knowledge Service Platform for Trustworthy Internet of Things (No. ZF1213) and ANR Grant (No. ANR-08-SEGI-006).

9 Appendix

In this appendix, we give the proof of Theorem 6.

Theorem 6 *Let ϕ be a rSCTPL formula. Let $\mathcal{M}$ be a program and $\mathcal{M}'$ be the program obtained from $\mathcal{M}$ by applying the procedure-cutting abstraction. Let $\mathcal{P}$ (resp. $\mathcal{P}'$) be the PDS modeling the program $\mathcal{M}$ (resp. $\mathcal{M}'$). If all the removed procedures are infinite execution free, then $\mathcal{P}$ satisfies ϕ iff $\mathcal{P}'$ satisfies ϕ.*

Proof Let $\mathcal{P} = (P, \Gamma, \Delta)$ be the PDS model of the program $\mathcal{M}$, $\mathcal{P}' = (P', \Gamma', \Delta')$ the PDS model of the program $\mathcal{M}'$. Let *main* be the entry procedure of the program $\mathcal{M}$, $main_0$ the entry point of the procedure *main*. We can get that $P = P' = \{s_0\}$, $\Gamma' \subseteq \Gamma$, and *main* is also the entry procedure of the program $\mathcal{M}'$. Thus, the initial configuration of $\mathcal{P}$ and $\mathcal{P}'$ is $\langle s_0, main_0\rangle$. Let λ and λ' be the labeling functions of $\mathcal{P}$ and $\mathcal{P}'$, respectively. Then, for every control point $n \in \Gamma'$, environment function $B \in \mathcal{B}$ and atomic predicate $a(x_1, \ldots, x_m)$ used in ϕ, $n \in \lambda(a(B(x_1), \ldots, B(x_m)))$ iff $n \in \lambda'(a(B(x_1), \ldots, B(x_m)))$.

To prove that $\mathcal{P}$ satisfies ϕ iff $\mathcal{P}'$ satisfies ψ, it is sufficient to show that $\langle s_0, main_0\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, main_0\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$. We prove that for every $\langle s_0, n\omega\rangle \in P \times \Gamma'^*$ s.t. $n \in \Gamma'$ and $\omega \in \Gamma'^*$, $B \in \mathcal{B}$, $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$. The proof proceeds by induction on the structure of ϕ.

Let $\Longrightarrow_{\mathcal{P}}$ be the transitive closure of the relation $\rightsquigarrow_{\mathcal{P}}$, $\Longrightarrow_{\mathcal{P}'}$ the transitive closure of the relation $\rightsquigarrow_{\mathcal{P}'}$.

– **Case** $\phi \equiv a(x_1, \ldots, x_m)$: Since $n \in \lambda(a(B(x_1), \ldots, B(x_m)))$ iff $n \in \lambda'(a(B(x_1), \ldots, B(x_m)))$, we obtain that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv \neg a(x_1, \ldots, x_m)$: Since $n \notin \lambda(a(B(x_1), \ldots, B(x_m)))$ iff $n \notin \lambda'(a(B(x_1), \ldots, B(x_m)))$, we obtain that $\langle s_0, n\omega\rangle \models^B_\lambda \neg\phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \neg\phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv a(x_1, \ldots, x_m) \wedge e$: Whether $(\langle s_0, n\omega\rangle, B)$ is in $L(e)$ or not is independent of λ and λ', we obtain that $\langle s_0, n\omega\rangle \models^B_\lambda e$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} e$ in $\mathcal{P}'$.

$\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_\lambda a(x_1, \ldots, x_m)$ in $\mathcal{P}$ and $\langle s_0, n\omega\rangle \models^B_\lambda e$ in $\mathcal{P}$. $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} a(x_1, \ldots, x_m)$ in $\mathcal{P}'$ and $\langle s_0, n\omega\rangle \models^B_{\lambda'} e$ in $\mathcal{P}'$. By applying the induction hypothesis: $\langle s_0, n\omega\rangle \models^B_\lambda a(x_1, \ldots, x_m)$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} a(x_1, \ldots, x_m)$ in $\mathcal{P}'$. We obtain that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv a(x_1, \ldots, x_m) \wedge \neg e$: Whether $(\langle s_0, n\omega\rangle, B)$ is in $L(e)$ or not is independent of λ and λ', we obtain that $\langle s_0, n\omega\rangle \models^B_\lambda \neg e$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \neg e$ in $\mathcal{P}'$.

$\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_\lambda a(x_1, \ldots, x_m)$ in $\mathcal{P}$ and $\langle s_0, n\omega\rangle \models^B_\lambda \neg e$ in $\mathcal{P}$. $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} a(x_1, \ldots, x_m)$ in $\mathcal{P}'$ and $\langle s_0, n\omega\rangle \models^B_{\lambda'} \neg e$ in $\mathcal{P}'$. By applying the induction hypothesis: $\langle s_0, n\omega\rangle \models^B_\lambda a(x_1, \ldots, x_m)$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} a(x_1, \ldots, x_m)$ in $\mathcal{P}'$. We obtain that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv \phi_1 \wedge \phi_2$: $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$ and $\langle s_0, n\omega\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$. $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$ and $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$.

By applying the induction hypothesis to ϕ_1 and ϕ_2, we obtain that $\langle s_0, n\omega\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$, and $\langle s_0, n\omega\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$. Thus, $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv \phi_1 \vee \phi_2$: $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$ or $\langle s_0, n\omega\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$. $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$ or $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$. By applying the induction hypothesis to ϕ_1 and ϕ_2, we obtain that $\langle s_0, n\omega\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$, and $\langle s_0, n\omega\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$. Thus, $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv \exists x\phi'$: $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff there exists $v \in \mathcal{D}$ such that $\langle s_0, n\omega\rangle \models^{B[x\leftarrow v]}_\lambda \phi'$ in $\mathcal{P}$. $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$ iff there exists $v \in \mathcal{D}$ such that $\langle s_0, n\omega\rangle \models^{B[x\leftarrow v]}_{\lambda'} \phi'$ in $\mathcal{P}'$. By applying the induction hypothesis: $\langle s_0, n\omega\rangle \models^{B[x\leftarrow v]}_\lambda \phi'$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^{B[x\leftarrow v]}_{\lambda'} \phi'$ in $\mathcal{P}'$. Thus, $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv \forall x \phi'$: $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff for every $v \in \mathcal{D}$, $\langle s_0, n\omega\rangle \models^{B[x\leftarrow v]}_\lambda \phi'$ in $\mathcal{P}$. $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$ iff for every $v \in \mathcal{D}$, $\langle s_0, n\omega\rangle \models^{B[x\leftarrow v]}_{\lambda'} \phi'$ in $\mathcal{P}'$. By applying the induction hypothesis: for every $v \in \mathcal{D}$, $\langle s_0, n\omega\rangle \models^{B[x\leftarrow v]}_\lambda \phi'$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^{B[x\leftarrow v]}_{\lambda'} \phi'$ in $\mathcal{P}'$. Thus, $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$: First we show that (**Case** i) if $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$, then $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$. Next, we show that (**Case** ii) if $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$, then $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$.

  – **Case** i: Suppose $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$, we show that $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.
  $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff there exist a run $\langle s_0, n_0\omega_0\rangle\langle s_0, n_1\omega_1\rangle \ldots$ and $k \geq 0$ such that $n_0\omega_0 = n\omega$, for every $0 \leq i < k$: $\langle s_0, n_i\omega_i\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$ and $\langle s_0, n_k\omega_k\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$. Suppose k is the minimum index such that the above holds.

  We construct a subsequence $n_{j_0}, \ldots, n_{j_m}$ of $n_0, \ldots, n_k$ as follows: n_{j_0} is the first control point in the sequence of $n_0, \ldots, n_k$ such that the statement at the control point n_{j_0} is a function call $y_{j_0} = f_{j_0}(\ldots)$ and is replaced by a *skip* statement. Since the function f_{j_0} is infinite execution free, then, necessarily, there exists a control point $n_{j_0'}$ in $n_{j_0+1}, \ldots, n_k$ such that $n_{j_0} \xrightarrow{y_{j_0}=f_{j_0}(\ldots)} n_{j_0'}$ is an edge of the control flow graph of $\mathcal{M}$ and $\omega_{j_0} = \omega_{j_0'}$ (note that if f_{j_0} is not infinite execution free, then such j_0' may not exist, as the run of $\mathcal{P}$ may never reach the return address of the caller site.). This means that $n_{j_0'}$ is the corresponding return address of the function call $f_{j_0}(\ldots)$ made at the control point n_{j_0}. For every $0 < t \leq m$, n_{j_t} is the first control point of the sequence $n_{j'_{t-1}}, \ldots, n_k$ such that the statement at the control point n_{j_t} is a function call $y_{j_t} = f_{j_t}(\ldots)$ and is replaced by a *skip* statement. $n_{j_t'}$ is the control point of $n_{j_t}, \ldots, n_k$ such that $n_{j_t} \xrightarrow{y_{j_t}=f_{j_t}(\ldots)} n_{j_t'}$ is an edge of the control flow graph of $\mathcal{M}$ and $\omega_{j_t} = \omega_{j_t'}$, i.e., $n_{j_t'}$ is the corresponding return address of the function call $f_{j_t}(\ldots)$ made at the control point n_{j_t}.

  Since for every $0 \leq t \leq m$, each control point crossed in the run of $\langle s_0, n_{j_t}\omega_{j_t}\rangle \Longrightarrow_{\mathcal{P}} \langle s_0, n_{j_t'}\omega_{j_t'}\rangle$ is a control point of the procedure f_{j_t}, we obtain that $\langle s_0, n_{j_t}\omega_{j_t}\rangle \rightsquigarrow_{\mathcal{P}'} \langle s_0, n_{j_t'}\omega_{j_t'}\rangle$.

  Since k is the minimum index, we obtain that $\langle s_0, n_k\omega_k\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$ and $\langle s_0, n_{k-1}\omega_{k-1}\rangle \not\models^B_\lambda \phi_2$ in $\mathcal{P}$. Thus, the control point n_k is not in any removed procedure. This implies that $j_t' \leq k$.

  According to the above construction, we get that the sequence $\langle s_0, n_0\omega_0\rangle, \ldots, \langle s_0, n_{j_0}\omega_{j_0}\rangle, \ldots, \langle s_0, n_{j_m}\omega_{j_m}\rangle, \ldots, \langle s_0, n_k\omega_k\rangle$ is a run of $\mathcal{P}'$.

  By applying the induction hypothesis, we obtain that $\langle s_0, n_k\omega_k\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$, $\langle s_0, n_0\omega_0\rangle \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$, and for every configuration c crossed in the run of $\langle s_0, n_0\omega_0\rangle \Longrightarrow_{\mathcal{P}'} \langle s_0, n_k\omega_k\rangle$, $c \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$. Thus, $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

  – **Case** ii: Suppose $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$, we show that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$.

  Since $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$, there exist a run $\langle s_0, n_0\omega_0\rangle, \langle s_0, n_1\omega_1\rangle \ldots$ and $k \geq 0$ such that $n_0\omega_0 = n\omega$, for every $0 \leq i < k$: $\langle s_0, n_i\omega_i\rangle \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$ and $\langle s_0, n_k\omega_k\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$. Suppose k is the minimum index such that the above holds.

  By applying the induction hypothesis, we obtain that $\langle s_0, n_k\omega_k\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$ and for every $1 \leq i < k$, $\langle s_0, n_i\omega_i\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$.

  Since the removed procedures are infinite execution free, for every $0 \leq i < k$: $\langle s_0, n_i\omega_i\rangle \rightsquigarrow_{\mathcal{P}'} \langle s_0, n_{i+1}\omega_{i+1}\rangle$, we have that $\langle s_0, n_i\omega_i\rangle \Longrightarrow_{\mathcal{P}} \langle s_0, n_{i+1}\omega_{i+1}\rangle$. Thus, $\mathcal{P}$ has a run $\langle s_0, n_0\omega_0\rangle \Longrightarrow_{\mathcal{P}} \langle s_0, n_1\omega_1\rangle \Longrightarrow_{\mathcal{P}} \cdots \langle s_0, n_{k-1}\omega_{k-1}\rangle \Longrightarrow_{\mathcal{P}} \langle s_0, n_k\omega_k\rangle$.

  To show that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$, it is sufficient to prove that for every $0 \leq i < k$, every configuration $\langle s_0, n_i'\omega_i'\rangle$ crossed in the run of $\langle s_0, n_i\omega_i\rangle \Longrightarrow_{\mathcal{P}} \langle s_0, n_{i+1}\omega_{i+1}\rangle$, $\langle s_0, n_i'\omega_i'\rangle \models^B_\lambda \phi_1$ holds in $\mathcal{P}$. If $\langle s_0, n_i'\omega_i'\rangle$ is a configuration crossed in the run of $\langle s_0, n_i\omega_i\rangle \Longrightarrow_{\mathcal{P}} \langle s_0, n_{i+1}\omega_{i+1}\rangle$, then the statement at the control point n_i is a function call $y = f(\ldots)$. Moreover, the run of the procedure f does not reach any control point in any procedure of $Op(\phi)$ and $f \notin Op(\phi)$. Thus, for any atomic predicate $a(x_1, \ldots, x_m)$, environment function $B' \in \mathcal{B}$, $n_i \notin \lambda(a(B'(x_1), \ldots, B'(x_m)))$ and $n_i' \notin \lambda(a(B'(x_1), \ldots, B'(x_m)))$.

  Now, to prove that $\langle s_0, n_i'\omega_i'\rangle \models^B_\lambda \phi_1$ holds in $\mathcal{P}$, it is sufficient to prove that for every regular variable expression e in $cl(\phi_1)$, $\langle s_0, n_i'\omega_i'\rangle \models^B_\lambda \phi_1$ always holds in $\mathcal{P}$ whether $(\langle s_0, n_i'\omega_i'\rangle, B)$ is in $L(e)$ or not.

  Since every regular variable expression e can appear in ϕ_1 only in the form of $a(x_1, \ldots, x_m) \wedge e$, $n_i \notin \lambda(a(B'(x_1), \ldots, B'(x_m)))$ and $n_i' \notin \lambda(a(B'(x_1), \ldots, B'(x_m)))$, we obtain that $\langle s_0, n_i'\omega_i'\rangle \not\models^{B'}_\lambda a(x_1, \ldots, x_m) \wedge e$ in $\mathcal{P}$ and $\langle s_0, n_i\omega_i\rangle \not\models^{B'}_\lambda a(x_1, \ldots, x_m) \wedge e$ in $\mathcal{P}$. These imply that $\langle s_0, n_i'\omega_i'\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$. Thus, we obtain that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$.

– **Case** $\phi \equiv \mathbf{A}[\phi_1 \mathbf{U} \phi_2]$ is similar to $\phi \equiv \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$.

– **Case** $\phi \equiv \mathbf{E}[\phi_1 \mathbf{R} \phi_2]$: First we show that (**Case** i) if $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$, then $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$. Next, we show that (**Case** ii) if $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$, then $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$.

– **Case** i: Suppose $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$, we show that $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.
Since $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$, there exists a run $\langle s_0, n_0\omega_0\rangle, \langle s_0, n_1\omega_1\rangle, \ldots$ in $\mathcal{P}$ such that $n_0\omega_0 = n\omega$ and, either (**Case** 1) for every $i \geq 0$, $\langle s_0, n_i\omega_i\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$, or (**Case** 2) there exists $m \geq 0$ s.t. for every $0 \leq i \leq m$, $\langle s_0, n_i\omega_i\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$ and $\langle s_0, n_m\omega_m\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$. The proof depends on whether **Case** 1 or **Case** 2 holds.

  - **Case** 1: For every $i \geq 0$, $\langle s_0, n_i\omega_i\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$. As discussed in **Case** $\phi \equiv \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$, there exists a subsequence $\langle s_0, n_0\omega_0\rangle, \langle s_0, n_{j_1}\omega_{j_1}\rangle, \ldots$ of $\langle s_0, n_0\omega_0\rangle, \langle s_0, n_1\omega_1\rangle, \ldots$ such that the subsequence $\langle s_0, n'_0\omega'_0\rangle, \langle s_0, n'_1\omega'_1\rangle, \ldots$ is a run of $\mathcal{P}'$ and $n'_0\omega'_0 = n_0\omega_0$.
    By applying the induction hypothesis, we obtain that for every $i \geq 0$, $\langle s_0, n'_i\omega'_i\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$. Thus, $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

  - **Case** 2: There exists $m \geq 0$ s.t. for every $0 \leq i \leq m$, $\langle s_0, n_i\omega_i\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$ and $\langle s_0, n_m\omega_m\rangle \models^B_\lambda \phi_1$ in $\mathcal{P}$. We can show that $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$ as done for the case $\phi \equiv \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$.

– **Case** ii: Suppose $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$, we show that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$.
Since $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$, there exists a run $\langle s_0, n_0\omega_0\rangle, \langle s_0, n_1\omega_1\rangle, \ldots$ in $\mathcal{P}'$ such that $n_0\omega_0 = n\omega$ and, either (**Case** 1) for every $i \geq 0$, $\langle s_0, n_i\omega_i\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$, or (**Case** 2) there exists $m \geq 0$ s.t. for every $0 \leq i \leq m$, $\langle s_0, n_i\omega_i\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$ and $\langle s_0, n_m\omega_m\rangle \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$. The proof depends on whether **Case** 1 or **Case** 2 holds.

  - **Case** 1: For every $i \geq 0$, $\langle s_0, n_i\omega_i\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$. As discussed in **Case** $\phi \equiv \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$, there exists a run $\langle s_0, n_0\omega_0\rangle \Longrightarrow_{\mathcal{P}} \langle s_0, n_1\omega_1\rangle \Longrightarrow_{\mathcal{P}} \ldots$ in $\mathcal{P}$.
    By applying the induction hypothesis, we obtain that $\langle s_0, n_i\omega_i\rangle \models^B_\lambda \phi_2$ in $\mathcal{P}$. As discussed in **Case** $\phi \equiv \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$, we can show that for every $i \geq 0$, for every configuration c crossed in the run of $\langle s_0, n_i\omega_i\rangle \Longrightarrow_{\mathcal{P}} \langle s_0, n_{i+1}\omega_{i+1}\rangle$, $c \models^B_\lambda \phi_2$ in $\mathcal{P}$. Thus, $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$.

  - **Case** 2: There exists $m \geq 0$ s.t. for every $0 \leq i \leq m$, $\langle s_0, n_i\omega_i\rangle \models^B_{\lambda'} \phi_2$ in $\mathcal{P}'$ and $\langle s_0, n_m\omega_m\rangle \models^B_{\lambda'} \phi_1$ in $\mathcal{P}'$. We can show that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ as done for the case $\phi \equiv \mathbf{E}[\phi_1 \mathbf{U} \phi_2]$.

– **Case** $\phi \equiv \mathbf{A}[\phi_1 \mathbf{R} \phi_2]$ is similar to $\phi \equiv \mathbf{E}[\phi_1 \mathbf{R} \phi_2]$.

– **Case** $\phi \equiv a(x_1, \ldots, x_m) \wedge \mathbf{EX}\varphi'$ $\mathbf{EX}\varphi'$ is in the syntax of ψ in the definition of $rSCTPL$ in Sect. 5.1):
By applying the induction hypothesis to $a(x_1, \ldots, x_m)$: we obtain that $\langle s_0, n\omega\rangle \models^B_\lambda a(x_1, \ldots, x_m)$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} a(x_1, \ldots, x_m)$ in $\mathcal{P}'$.
Since the procedure containing the control point n is preserved due to $a(x_1, \ldots, x_m)$, we obtain that n' is a next control point of n in $\mathcal{M}$ iff n' is a next control point of n in $\mathcal{M}'$. Thus, $\langle s_0, n\omega\rangle \rightsquigarrow_{\mathcal{P}} \langle s_0, n'\omega'\rangle$ iff $\langle s_0, n\omega\rangle \rightsquigarrow_{\mathcal{P}'} \langle s_0, n'\omega'\rangle$.
By applying the induction hypothesis to φ': we obtain that $\langle s_0, n'\omega'\rangle \models^B_\lambda \varphi'$ in $\mathcal{P}$ iff $\langle s_0, n'\omega'\rangle \models^B_{\lambda'} \varphi'$ in $\mathcal{P}'$. Thus, $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$.

– **Case** $\phi \equiv a(x_1, \ldots, x_m) \wedge \mathbf{AX}\varphi'$ ($\mathbf{EX}\varphi'$ is in the syntax of ψ in the definition of $rSCTPL$ in Sect. 5.1):: We can show that $\langle s_0, n\omega\rangle \models^B_\lambda \phi$ in $\mathcal{P}$ iff $\langle s_0, n\omega\rangle \models^B_{\lambda'} \phi$ in $\mathcal{P}'$ as done for the case $\phi \equiv a(x_1, \ldots, x_m) \wedge \mathbf{EX}\varphi'$. □

References

1. Acacia+: http://lit2.ulb.ac.be/acaciaplus/
2. Acharya, M., Xie, T.: Mining API error-handling specifications from source code. In: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS), pp. 370–384, York, UK (2009). doi:10.1007/978-3-642-00593-0_25
3. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE), pp. 25–34, Dubrovnik, Croatia (2007). doi:10.1145/1287624.1287630
4. Alur, R., Arenas, M., Barceló, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. Log. Methods Comput. Sci. **4**(4) (2008). doi:10.2168/LMCS-4(4:11)2008
5. Alur, R., Chaudhuri, S., Madhusudan, P.: Software model checking using languages of nested trees. ACM Trans. Program. Lang. Syst. **33**(5), 15 (2011). doi:10.1145/2039346.2039347
6. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS), pp. 467–481, Barcelona, Spain (2004). doi:10.1007/978-3-540-24730-2_35
7. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 4–16, Portland, OR, USA (2002). doi:10.1145/503272.503275
8. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
9. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011). doi:10.1145/1965724.1965743
10. Besson, F., Jensen, T.P., Métayer, D.L.: Model checking security properties of control flow graphs. J. Comput. Secur. **9**(3), 217–250 (2001)
11. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Proceedings of the software model checker BLAST. Int. J. Softw. Tools Technol. Transf. **9**(5–6), 505–525 (2007). doi:10.1007/s10009-007-0044-z

12. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electron. Notes Theor. Comput. Sci. **66**(2), 160–177 (2002). doi:10.1016/S1571-0661(04)80410-9
13. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Proceedings of the 8th International Conference on Concurrency Theory (CONCUR), pp. 135–150, Warsaw, Poland (1997). doi:10.1007/3-540-63141-0_10
14. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Softw. Eng. **30**(6), 388–402 (2004). doi:10.1109/TSE.2004.22
15. Chen, H., Dean, D., Wagner, D.: Model checking one million lines of C code. In: Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA (2004)
16. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: ACM Conference on Computer and Communications Security, pp. 235–244 (2002). doi:10.1145/586110.586142. http://doi.acm.org/10.1145/586110.586142
17. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 399–410, Austin, TX, USA (2011). doi:10.1145/1926385.1926431
18. Cook, B., Koskinen, E., Vardi, M.Y.: Temporal property verification as a program analysis task—extended version. Form. Methods Syst. Des. **41**(1), 66–82 (2012). doi:10.1007/s10703-012-0153-5
19. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE), pp. 439–448, Limerick, Ireland (2000). doi:10.1145/337180.337234
20. Driscoll, E., Thakur, A.V., Reps, T.W.: OpenNWA: A nested-word automaton library. In: Proceedings of the 24th International Conference on Computer Aided Verification (CAV), pp. 665–671, Berkeley, CA, USA (2012). doi:10.1007/978-3-642-31424-7_47
21. elastic: http://www.ulb.ac.be/di/ssd/madewulf/aasap/
22. Elgammal, A., Türetken, O., van den Heuvel, W.J., Papazoglou, M.P.: On the formal specification of regulatory compliance: a comparative analysis. In: Proceedings of International Workshops on Service-Oriented Computing (ICSOC), PAASC, WESOA, SEE, and SOC-LOG, pp. 27–38, San Francisco, CA, USA, Revised Selected Papers (2010). doi:10.1007/978-3-642-19394-1_4
23. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Proceedings of the 12th International Conference on Computer Aided Verification (CAV), pp. 232–247, Chicago, IL, USA (2000). doi:10.1007/10722167_20
24. Esparza, J., Kucera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Inf. Comput. **186**(2), 355–376 (2003). doi:10.1016/S0890-5401(03)00139-1
25. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Proceedings of International Conference on Formal Verification of Object-Oriented Software (FoVeOOS), pp. 10–30, Paris, France, Revised Selected Papers (2010). doi:10.1007/978-3-642-18070-5_2
26. Getafix: http://www.cs.uiuc.edu/madhu/getafix/
27. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE), pp. 339–349, Atlanta, GA, USA (2008). doi:10.1145/1453101.1453150
28. Gabel, M., Su, Z.: Symbolic mining of temporal specifications. In: Proceedings of the 30th International Conference on Software Engineering (ICSE), pp. 51–60, Leipzig, Germany (2008). doi:10.1145/1368088.1368096
29. Godefroid, P.: Software model checking: the Verisoft approach. Form. Methods Syst. Des. **26**(2), 77–101 (2005). doi:10.1007/s10703-005-1489-x
30. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher-Order Symb. Comput. **13**(4), 315–353 (2000). doi:10.1023/A:1026599015809
31. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE), pp. 31–40, Lisbon, Portugal (2005). doi:10.1145/1081706.1081713
32. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Proceedings of the 2nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 174–187, Vienna, Austria (2005). doi:10.1007/11506881_11
33. Kremenek, T., Twohey, P., Back, G., Ng, A.Y., Engler, D.R.: From uncertainty to belief: Inferring the specification within. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), pp. 161–176, Seattle, WA, USA (2006)
34. Kroening, D.: CBMC http://www.cprover.org/cbmc (2012)
35. Liu, C., Ye, E., Richardson, D.J.: Software library usage pattern extraction using a software model checker. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 301–304, Tokyo, Japan (2006). doi:10.1109/ASE.2006.63
36. Lo, D., Khoo, S.C.: SMArTIC: towards building an accurate, robust and scalable specification miner. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 265–275, Portland, OR, USA (2006). doi:10.1145/1181775.1181808
37. Lo, D., Ramalingam, G., Ranganath, V.P., Vaswani, K.: Mining quantified temporal rules: formalism, algorithms, and evaluation. Sci. Comput. Program. **77**(6), 743–759 (2012). doi:10.1016/j.scico.2010.10.003
38. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering (ICSE), pp. 501–510, Leipzig, Germany (2008). doi:10.1145/1368088.1368157
39. Mckit: http://www.fmi.uni-stuttgart.de/szs/tools/mckit/
40. Minisat: C Language Version. http://minisat.se/MiniSat.html
41. Mist2: http://software.imdea.org/pierreganty/software.html
42. Moped: http://www.fmi.uni-stuttgart.de/szs/tools/moped/
43. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 123–134, San Diego, California, USA (2007). doi:10.1145/1250734.1250749
44. Reps, T.W., Lal, A., Kidd, N.: Program analysis using weighted pushdown systems. In: Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), pp. 23–51, New Delhi, India (2007). doi:10.1007/978-3-540-77050-3_4
45. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. **58**(1–2), 206–263 (2005). doi:10.1016/j.scico.2005.02.009
46. SourceForge: http://sourceforge.net (2012)
47. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000). doi:10.1145/353323.353382
48. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electron. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006). doi:10.1016/j.entcs.2005.11.018

49. Seshadri, P.: Generic Socket Programming Tutorial (2008). http://www.prasannatech.net/2008/07/socket-programming-tutorial.html
50. Shoham, S., Yahav, E., Fink, S.J., Pistoia, M.: Static specification mining using automata-based abstractions. IEEE Trans. Softw. Eng. **34**(5), 651–666 (2008). doi:10.1109/TSE.2008.63
51. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. In: Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR), pp. 434–449 , Aachen, Germany (2011). doi:10.1007/978-3-642-23217-6_29
52. Song, F., Touili, T.: Efficient malware detection using model-checking. In: Proceedings of the 18th International Symposium on Formal Methods (FM), pp. 418–433, Paris, France (2012). doi:10.1007/978-3-642-32759-9_34
53. Song, F., Touili, T.: Pushdown model checking for malware detection. In: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS), pp. 110–125, Tallinn, Estonia (2012). doi:10.1007/978-3-642-28756-5_9
54. Song, F., Touili, T.: LTL model-checking for malware detection. In: Proceedings of the 19th International Conferene on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS), pp. 416–431, Rome, Italy (2013). doi:10.1007/978-3-642-36742-7_29
55. Song, F., Touili, T.: Model-checking software library API usage rules. In: Proceedings of the 10th International Conference on Integrated Formal Methods (iFM), pp. 192–207, Turku, Finland (2013). doi:10.1007/978-3-642-38613-8_14
56. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In: Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 141–153, Beijing, China (2006). doi:10.1007/11901914_13
57. Tspass: http://www.csc.liv.ac.uk/michel/software/tspass/
58. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**(3) (1995)
59. Ubcsat: http://ubcsat.dtompkins.com/
60. Verbs: http://lcs.ios.ac.cn/zwh/verbs/index.html
61. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003). doi:10.1023/A:1022920129859
62. Visser, W., Mehlitz, P.C.: Model checking programs with Java PathFinder. In: Proceedings of the 12th International SPIN Workshop on Model Checking Software (SPIN), p. 27, San Francisco, CA, USA (2005). doi:10.1007/11537328_5
63. Walksat: version 35. http://www.cs.rochester.edu/kautz/walksat/
64. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. Autom. Softw. Eng. **18**(3–4), 263–292 (2011). doi:10.1007/s10515-011-0084-1
65. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE), pp. 35–44, Dubrovnik, Croatia (2007). doi:10.1145/1287624.1287632
66. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 501–504, Atlanta, GA, USA (2007). doi:10.1145/1321631.1321719
67. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005). doi:10.1145/1050849.1050865
68. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proceedings of the 28th International Conference on Software Engineering (ICSE), pp. 282–291, Shanghai, China (2006). doi:10.1145/1134325

Fu Song is a lecturer at Software Engineering Institute of East China Normal University, People's Republic of China. He received his Ph.D. in Computer Science from University Paris 7 in 2013 and M.Sc. from Software Engineering Institute of East China Normal University in 2009. His major research interests include software verification (e.g., infinite-state system modeling, temporal logics and model-checking), computer security (e.g., malware detection and binary code disassembly). Fu has published more than 15 refereed papers in international journals and conferences.

Tayssir Touili is a senior CNRS researcher (DR) at the laboratory LIPN, France. During 2004 and 2014, she was a CNRS researcher at the laboratory LIAFA, France. She received her Habilitation and PhD in Computer Science from University of Paris 7 in 2009 and 2003, respectively. Her major research interests include software verification (e.g., infinite-state system modeling and model-checking), computer security (e.g., malware detection). Touili has been participating in numerous research projects in the verification area and contributing about 50 scientific papers on verification-related topics. She has been Co-Chair of the CAV'10, VECOS'11 and PC of the CAV'15, PLDI'15, VMCAI'14, POPL'14, etc. conferences.