Using Unified Model Checking to Verify Heaps

Xu Lu, Zhenhua Duan(✉), and Cong Tian(✉)

ICTT and ISN Lab, Xidian University, Xi'an 710071, P.R. China
{zhhduan,ctian}@mail.xidian.edu.cn

Abstract. This paper addresses the problem of verifying heap evolution properties of pointer programs. To this end, a new unified model checking approach with MSVL (Modeling, Simulation and Verification Language) and PPTL^{SL} is presented. The former is an executable subset of PTL (Projection Temporal Logic) while the latter is an extension of PPTL (Propositional Projection Temporal Logic) with separation logic. MSVL is used to model pointer programs, and PPTL^{SL} to specify heap evolution properties. In addition, we implement a prototype in order to demonstrate our approach.

Keywords: Heap verification · Model checking · MSVL · PPTL · Separation logic

1 Introduction

Pointers are indispensable in real-world programs or applications. Reasoning about pointer programs is quite challenging since pointer usage is often complex and flexible. Potential bugs encountered in pointer programs such as null pointer dereference, memory leaks, or shape destruction are due to the nature of pointers. The problem is more serious for concurrent programs since we need to consider all possible execution sequences of processes. Alias analysis, as the name implies, is a point-to analysis which naively checks whether pointers can be aliased. Shape analysis is another form of pointer analysis that attempts to discover the possible shapes of heap structures. It aims to prove that these structures are not misused or corrupted.

Reynolds [1] proposes a famous Hoare-style logic known as separation logic which has received much attention. For the last decade, many works extend separation logic to do automated assertion checking [2] and shape analysis [3] in real-world applications. PTL (Pointer Assertion Logic) is a notation for expressing assertions about the heap structures of imperative languages. PALE (PAL Engine) is a complete implementation of PAL that encodes both programs and partial assertions as formulas in monadic second-order logic. However, loop invariants have to be manually provided so that it is not fully automatic.

This research is supported by the National Natural Science Foundation of China Grant Nos. 61133001, 61322202, 61420106004, and 91418201.

© Springer International Publishing AG 2016
T-H.H. Chan et al. (Eds.): COCOA 2016, LNCS 10043, pp. 735–743, 2016.
DOI: 10.1007/978-3-319-48749-6_55

In this paper we intend to apply the model checking framework to verify heap evolution properties. As the property-specification language we use a variant of temporal logic namely PPTL$^{\text{SL}}$ [4]. PPTL$^{\text{SL}}$ is a two-dimensional (spatial and temporal) logic by extending PPTL (Propositional Projection Temporal Logic) [5] with a decidable fragment of separation logic. For the program part, our method makes use of a temporal logic programming language, which is an executable subset of PTL (Projection Temporal Logic), called MSVL (Modeling, Simulation and Verification Language) [6], to model heap programs. PPTL$^{\text{SL}}$ can be translated (preserving satisfiability) into a strict subset of PTL. Specifications and models lie in the same logic framework, hence the name unified model checking. The previous unified model checking approach [7] cannot verify heap evolution properties. We extend the approach in [7] by replacing PPTL with the more expressive specification language PPTL$^{\text{SL}}$ such that heap evolution properties can be verified, and also the corresponding model checking approach is developed in this paper.

The work in [8] studies the problem of establishing temporal properties, including liveness properties of Java programs with evolving heaps. A specification language Evolution Temporal Logic (ETL) is defined which is a first-order linear temporal logic with transitive closure. ETL mainly focuses on describing behaviors of large granularity heap objects and high-level threads. Navigation Temporal Logic (NTL) [9] extends LTL with pointer assertions on single-reference structures including primitives for the birth and death of entities. The abstracted model checking algorithm for NTL is a non-trivial extension of the tableau based algorithm for LTL, which can be applied for both sequential and concurrent pointer programs. The major disadvantage of the approach of [9] is that it can only verify programs manipulating singly-linked lists. In [10], Rieger presents an abstraction and verification framework for pointer programs operating on unbounded heaps. In his work, two abstraction techniques are introduced, one is for singly-linked structures and the other employs context-free hyperedge replacement graph grammars to model more general heap structures. A two-dimensional (time and space) logic named maLTL is developed in [11] by combining temporal logic LTL and CTL, which is suitable to deal with pointers and heap management in the context of C programs. Since both dimensions of maLTL are realized by temporal logics that makes the difference between the two dimensions unclear.

2 Projection Temporal Logic

Let Var be a countable set of typed variables consisting of static and dynamic variables, and $Prop$ a countable set of propositions. $\mathbb{B}$ represents the boolean domain $\{true, false\}$, and $\mathbb{D}$ denotes the data domain. The terms e and formulas Q of PTL are given by the following grammar:

$$e ::= x \mid \bigcirc e \mid \circleddash e \mid fun(e_1, \ldots, e_n)$$
$$Q ::= q \mid e_1 = e_2 \mid Pred(e_1, \ldots, e_n) \mid \neg Q \mid Q_1 \vee Q_2 \mid \exists y : Q \mid \bigcirc Q \mid Q^* \mid (Q_1, \ldots, Q_m)\, prj\, Q$$

where $q \in Prop$ is a proposition, $x \in Var$ a variable, and fun a function of arity n and $Pred$ is a predicate of arity n.

A state s is defined to be a pair (I_v, I_p) of state interpretations I_v and I_p, $I_v : Var \to \mathbb{D} \cup \{nil\}, I_p : Prop \to \mathbb{B}$. An interval $\sigma = \langle s_0, s_1, \ldots \rangle$ is a non-empty sequence of states, finite or infinite. The length of σ, denoted by $|\sigma|$, is ω if σ is infinite, otherwise it is the number of states minus one. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider the set N_0 of non-negative integers, and define $N_\omega = N_0 \cup \{\omega\}$, and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0$, $i < \omega$. Moreover, we define $\preceq$ as $\leq -\{(\omega, \omega)\}$. With such a notation, $\sigma_{(i..j)}(0 \leq i \preceq j \leq |\sigma|)$ denotes the sub-interval $\langle s_i, \ldots, s_j \rangle$ and $\sigma^{(k)}(0 \leq k \preceq |\sigma|)$ denotes the suffix interval $\langle s_k, \ldots, s_{|\sigma|} \rangle$ of σ. The concatenation of σ with another interval σ' is denoted by $\sigma \cdot \sigma'$. Further, let $\sigma = \langle s_k, \ldots, s_{|\sigma|} \rangle$ be an interval and $r_1, \ldots, r_h$ be integers $(h \geq 1)$ such that $0 \leq r_1 \leq r_2 \leq \cdots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the interval, $\sigma \downarrow (r_1, \ldots, r_h) = \langle s_{t_1}, \ldots, s_{t_l} \rangle$, where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates.

An interpretation for a PTL formula is a triple $\mathcal{I} = (\sigma, k, j)$ where $\sigma = \langle s_0, s_1, \ldots \rangle$ is an interval, k a non-negative integer and j an integer or ω such that $k \preceq j \leq |\sigma|$. We write $(\sigma, k, j) \models Q$ to mean that a formula Q is interpreted over a sub-interval $\sigma_{(k..j)}$ of σ with the current state being s_k. The notation $s_k = (I_v^k, I_p^k)$ indexed by k represents the k-th state of an interval σ. The semantics of terms and PTL formulas are defined by:

$$\mathcal{I}[x] = \begin{cases} I_v^k[x] = I_v^i[x] & x \text{ is static,} \\ I_v^k[x] & \text{otherwise.} \end{cases} \qquad \mathcal{I}[fun(e_1, \ldots, e_n)] = \begin{cases} \mathcal{I}[fun](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) & \text{if } i < k, \\ nil & \text{otherwise.} \end{cases}$$

$$\mathcal{I}[\bigcirc e] = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j, \\ nil & \text{otherwise.} \end{cases} \qquad \mathcal{I}[\circleddash e] = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k, \\ nil & \text{otherwise.} \end{cases}$$

$\boldsymbol{\mathcal{I} \models q}$ iff $I_p^k(q) = true$. $\quad \boldsymbol{\mathcal{I} \models e_1 = e_2}$ iff $\mathcal{I}(e_1) = \mathcal{I}(e_2)$.

$\boldsymbol{\mathcal{I} \models Pred(e_1, \ldots, e_n)}$ iff $Pred(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) = true$ and $\mathcal{I}[e_i] \neq nil$, for all i.

$\boldsymbol{\mathcal{I} \models \neg Q}$ iff $\mathcal{I} \not\models Q$. $\quad \boldsymbol{\mathcal{I} \models \exists y : Q}$ iff $\exists \sigma'$ such that $\sigma'_{(k..j)} \overset{x}{=} \sigma_{(k..j)}$ and $(\sigma', k, j) \models Q$.

$\boldsymbol{\mathcal{I} \models Q_1 \vee Q_2}$ iff $\mathcal{I} \models Q_1$ or $\mathcal{I} \models Q_2$. $\quad \mathcal{I} \models \bigcirc Q$ iff $k < j$ and $(\sigma, k+1, j) \models Q$.

$\boldsymbol{\mathcal{I} \models (Q_1, \ldots, Q_m)\, prj\, Q}$ iff $\exists k = r_0 \leq r_1 \leq \cdots \leq r_m \preceq j$ such that $(\sigma, r_0, r_1) \models Q_1$, $(\sigma, r_{l-1}, r_l) \models Q_l (1 < l \leq m), (\sigma', 0, 0, |\sigma'|) \models Q$ for one of the σ' : (a) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma_{(r_m+1..j)}$(b) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_h)$ for $0 \leq h \leq m$.

$\boldsymbol{\mathcal{I} \models Q^*}$ iff $\exists r_0, \ldots, r_n \in N_\omega$ such that $k = r_0 \leq r_1 \leq \cdots \leq r_{n-1} \preceq r_n = j (n \geq 0)$ and $(\sigma, r_0, r_1) \models Q$ and for all $1 < l \leq n, (\sigma, r_{l-1}, r_l) \models Q$; or $\exists k = r_0 \leq r_1 \leq r_2 \leq \cdots$ such that $\lim_{i \to \infty} r_i = \omega$ and $(\sigma, r_0, r_1) \models Q$ and for $l > 1, (\sigma, r_{l-1}, r_l) \models Q$.

A formula Q is satisfied over an interval σ, written $\sigma \models Q$, if $(\sigma, 0, |\sigma|) \models Q$ holds. Also we have the following derived formulas:

$$\varepsilon \stackrel{\text{def}}{=} \neg \bigcirc true \quad more \stackrel{\text{def}}{=} \neg\varepsilon \quad Q_1; Q_2 \stackrel{\text{def}}{=} (Q_1, Q_2)\, prj\, \varepsilon \quad Q^+ \stackrel{\text{def}}{=} Q; Q^*$$

$$\Diamond Q \stackrel{\text{def}}{=} true; Q \quad \Box Q \stackrel{\text{def}}{=} \neg\Diamond\neg Q \quad len(n) \stackrel{\text{def}}{=} \bigcirc len(n-1) \quad skip \stackrel{\text{def}}{=} len(1)$$

where ε (or $len(0)$) denotes an interval with zero length, ";" and "$^+(^*)$" are used to describe sequential and loop properties respectively.

3 Modeling, Simulation and Verification Language

MSVL is an executable temporal logic with framing technique which is recently extended with function calls [12] and groups of types such as basic data types, pointer types and struct types [13]. Thus it is capable of modeling pointer programs. The arithmetic and boolean expressions of MSVL can be defined as:

$$e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 / e_2 \quad b ::= \neg b \mid b_1 \vee b_2 \mid e_1 = e_2 \mid e_1 < e_2$$

Some useful elementary statements of MSVL can be inductively defined as follows. A convenient way to execute MSVL programs is to transform them into their equivalent normal forms (Definition 1).

$$\boldsymbol{empty} \stackrel{\text{def}}{=} \varepsilon \quad \boldsymbol{x <== e} \stackrel{\text{def}}{=} x = e \wedge p_x \quad \boldsymbol{Q_1\, and\, Q_2} \stackrel{\text{def}}{=} Q_1 \wedge Q_2 \quad \boldsymbol{Q_1\, or\, Q_2} \stackrel{\text{def}}{=} Q_1 \vee Q_2$$

$$\boldsymbol{x := e} \stackrel{\text{def}}{=} \bigcirc x = e \wedge \bigcirc p_x \wedge skip \quad \boldsymbol{if\, b\, then\, Q_1\, else\, Q_2} \stackrel{\text{def}}{=} (b \rightarrow Q_1) \wedge (\neg b \rightarrow Q_2)$$

$$\boldsymbol{while\, b\, do\, Q} \stackrel{\text{def}}{=} (Q \wedge b)^* \wedge \Box(empty \rightarrow \neg b) \quad \boldsymbol{lbf(x)} \stackrel{\text{def}}{=} \neg af(x) \rightarrow \exists b : (\ominus x = b \wedge x = b)$$

$$\boldsymbol{frame(x)} \stackrel{\text{def}}{=} \Box(more \rightarrow \bigcirc lbf(x)) \quad \boldsymbol{Q_1 \| Q_2} \stackrel{\text{def}}{=} (Q_1 \wedge (Q_2; true)) \vee (Q_2 \wedge (Q_1; true))$$

$$\boldsymbol{await(b)} \stackrel{\text{def}}{=} frame(x_1, \ldots, x_n) \wedge \Box(empty \leftrightarrow b) \text{ where } x_i \in \{x \mid x \text{ appears in } b\}$$

Definition 1 (Normal Form of MSVL). An MSVL program Q is in normal form if $Q \stackrel{\text{def}}{=} \bigvee_{i=1}^{n'} Q_{e_i} \wedge \varepsilon \vee \bigvee_{j=1}^{n} Q_{c_j} \wedge \bigcirc Q_{f_j}$, where Q_{f_j} is a general MSVL program, whereas Q_{e_i} and Q_{c_j} are true or all are state formulas of the form: $(x_1 = e_1) \wedge \cdots \wedge (x_m = e_m) \wedge \dot{q}_{x_1} \wedge \cdots \wedge \dot{q}_{x_m}$. $\dot{q}$ denotes either q or $\neg q$.

The normal form divides the formula into two parts: the present part and the future part. A key conclusion is that any MSVL program can be reduced to its normal form [5,6]. Therefore, we can use an incremental way to execute MSVL programs based on normal form.

Theorem 1. Any MSVL program Q can be reduced to its normal form.

Using normal form of MSVL as a basis, a graph can be constructed, namely Normal Form Graph (NFG) [5,6], by recursively progressing the future part of a normal form, which explicitly illustrates the state space of an MSVL program.

3.1 Examples of MSVL Programs Manipulating Pointers

Producer-consumer program. Consider the producer-consumer problem encoded in MSVL below. The producer process and the consumer process share a buffer which is realized as a global singly-linked list. The producer repeatedly generates new items by allocating new memory heap cells, and adds them to the tail x of the buffer, whereas the consumer removes items from the head y of the buffer and disposes them.

The parallel operator "||" in MSVL considers the true concurrency semantics of programs. In order to simulate the interleaving semantics of the two concurrent processes, the *await* statement is employed to force a process to sleep when the waiting condition is false, otherwise the process will continue to execute.

```
struct Node { Node *nxt };
frame(PC, x, y, t, r) and (
  int PC<==0 and Node *x<==NULL, *y<==NULL, *t<==NULL, *r<==NULL and empty;
  y:=(Node*)malloc(sizeof(Node)) and x := (Node*)malloc(sizeof(Node));
  y->nxt:=x and (PC:=0 or PC:=1);
  //Producer                          //Consumer
  while(true) {                       while(true) {
    await(PC=0);                        if(x!=y) then {
    t:=(Node*)malloc(sizeof(Node));       await(PC=1);
    x->nxt:=t and (PC:=0 or PC:=1); ||    r:=y and (PC:=0 or PC:=1);
    await(PC=0);                          await(PC=1);
    x:=x->nxt and (PC:=0 or PC:=1)        y:=y->nxt and (PC:=0 or PC:=1);
  }                                       await(PC=1);
                                          next(free(r)) and (PC:=0 or PC:=1)
                                        } else {
                                          await(PC=1);(PC:=0 or PC:=1)
                                        }
                                      }
)
```

4 The Two-Dimensional Logic PPTL$^{\text{SL}}$

Previously, we integrate a decidable fragment of separation logic (referred to as SL) with PPTL to obtain a two-dimensional logic. The logic, referred to as PPTL$^{\text{SL}}$ [4], allows us to express heap evolution properties. We assume a countable set $PVar$ of variables with pointer type, and a finite set Loc of memory locations. $PVal = Loc \cup \{null\}$ denotes the set of pointer values which are either locations or $null$. The syntax of PPTL$^{\text{SL}}$ formulas P is defined by the grammar:

$$e ::= null \mid l \mid x \qquad \phi ::= e_1 = e_2 \mid e_0 \mapsto \{e_1, \ldots, e_n\} \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \# \phi_2 \mid \exists x : \phi$$
$$P ::= \phi \mid \neg P \mid P_1 \vee P_2 \mid \bigcirc P \mid (P_1, \ldots, P_m)\, prj\, P \mid P^*$$

$l \in Loc$, $x \in PVar$, ϕ represents SL formulas, and P PPTL$^{\text{SL}}$ formulas. Formula $e_0 \mapsto \{e_1, \ldots, e_n\}$ denotes that e_0 points to $e_1, \ldots, e_n$, where e_0 represents an address in the heap and $e_1, \ldots, e_n$ the consecutive values held in that address.

The formula $\phi_1\#\phi_2$ specifies properties holding respectively for disjoint portions of the current heap, one makes ϕ_1 true and the other makes ϕ_2 true. The temporal operators as well as their semantics are taken from PTL.

We refer to a pair (I_s, I_h) as a memory state, $I_s : PVar \rightharpoonup PVal, I_h : Loc \rightharpoonup \bigcup_{i=1}^{n} PVal^i$, where I_s represents a stack and I_h a heap. I_s serves as valuations of pointer variables and I_h as valuations of heap cells. We write $dom(f)$ to denote the domain of mapping f. Given two mappings f_1 and f_2, the notation $f_1 \perp f_2$ means that f_1 and f_2 have disjoint domains. Moreover, we use $f_1 \cdot f_2$ to denote the union of f_1 and f_2. The semantics of SL formulas is given by:

$(I_s, I_h)[null] = null \qquad (I_s, I_h)[l] = l \qquad (I_s, I_h)[x] = I_h(x)$

$\boldsymbol{I_s, I_h \models_{SL} e_1 = e_2}$ iff $(I_s, I_h)[e_1] = (I_s, I_h)[e_2]$. $\boldsymbol{I_s, I_h \models_{SL} \neg\phi}$ iff $I_s, I_h \not\models_{SL} \phi$.

$\boldsymbol{I_s, I_h \models_{SL} e_0 \mapsto \{e_1, \ldots, e_n\}}$ iff $dom(I_h) = \{(I_s, I_h)[e_0]\}$ and $I_h((I_s, I_h)[e_0]) = ((I_s, I_h)[e_1], \ldots, (I_s, I_h)[e_n])$. $\boldsymbol{I_s, I_h \models_{SL} \phi_1 \vee \phi_2}$ iff $I_s, I_h \models_{SL} \phi_1$ or $I_s, I_h \models_{SL} \phi_2$.

$\boldsymbol{I_s, I_h \models_{SL} \phi_1\#\phi_2}$ iff $\exists I_{h_1}, I_{h_2} : I_{h_1} \perp I_{h_2}$ and $I_h = I_{h_1} \cdot I_{h_2}$ and $I_s, I_{h_1} \models_{SL} \phi_1$ and $I_s, I_{h_2} \models_{SL} \phi_2$. $\boldsymbol{I_s, I_h \models_{SL} \exists x : \phi}$ iff $\exists v \in PVal$ such that $I_s[x \to v], I_h \models_{SL} \phi$.

The semantics of P is similar to that of Q since the only difference is their state formulas (formulas without temporal operators). Therefore, we only give the interpretation of state formulas, i.e., $\boldsymbol{\mathcal{I} \models \phi}$ iff $I_s^k, I_h^k \models_{SL} \phi$. We abusively use the notation $\mathcal{I} \models P$, and in this case P is interpreted over an interval of memory states.

SL can describe various heap structures, we present the following derived formulas expressed in SL which are related to singly-linked lists.

$$e \mapsto \{-_1, \ldots, -_n\} \stackrel{\text{def}}{=} \exists x_1, \ldots, x_n : e \mapsto \{x_1, \ldots, x_n\}\#true$$

$$e \hookrightarrow \{-, \ldots, -\} \stackrel{\text{def}}{=} e \mapsto \{-, \ldots, -\}\#true \quad alloc(e, n) \stackrel{\text{def}}{=} e \hookrightarrow \{-_1, \ldots, -_n\}$$

$$alloc(e) \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} alloc(e, i) \quad emp \stackrel{\text{def}}{=} \neg\exists x : alloc(x) \quad \sharp e \geq n \stackrel{\text{def}}{=} \#_{i=1}^{n}(\exists x_i : x_i \hookrightarrow \{e\})$$

$$e_1 \xrightarrow{\circlearrowleft} e_2 \stackrel{\text{def}}{=} alloc(e_1, 1) \wedge (e_2 \neq e_1 \rightarrow \neg alloc(e_2, 1) \wedge \sharp e_1 = 0) \wedge (\forall x : x \neq e_2 \rightarrow (\sharp x = 1 \rightarrow alloc(x, 1))) \wedge (\forall x : x \neq null \rightarrow \sharp x \leq 1) \wedge (\forall x : alloc(x) \rightarrow alloc(x, 1))$$

$$ls(e_1, e_2) \stackrel{\text{def}}{=} e_1 \xrightarrow{\circlearrowleft} e_2 \wedge \neg(e_1 \xrightarrow{\circlearrowleft} e_2\#\neg emp)$$

$$e_1 \rightarrow^+ e_2 \stackrel{\text{def}}{=} ls(e_1, e_2)\#true \quad e_1 \rightarrow^* e_2 \stackrel{\text{def}}{=} e_1 = e_2 \vee e_1 \rightarrow^+ e_2$$

Here, $ls(e_1, e_2)$ describes a list segment starting from the location e_1 to e_2, $e_1 \rightarrow^+ e_2$ and $e_1 \rightarrow^* e_2$ mean that e_2 is reachable from e_1 via certain pointer links. emp denotes an empty heap, and $alloc(e)$ indicates the address e is allocated in the current heap. Formulas describing other heap structures can also be derived.

4.1 Specify Heap Evolution Properties

For the producer-consumer program with a shared list, some interesting heap evolution properties can be specified by PPTL^{SL}:

(1) Absence of memory leaks, i.e., any item can be reached by certain variable during the execution of the program: $\Box(\forall z : alloc(z) \rightarrow (x \rightarrow^* z \vee y \rightarrow^* z \vee t \rightarrow^* z \vee r \rightarrow^* z))$.
(2) The tail of the list is never deleted nor disconnected from the head: $\bigcirc^2\Box(alloc(x) \wedge y \rightarrow^* x)$.
(3) Shape integrity of the buffer, i.e., the shape of the buffer is repeatedly formed as a linked list within every five time units: $\bigcirc^2((len(5) \wedge \Diamond ls(y, null))^+)$.

In [4], we have proved that PPTL$^{\text{SL}}$ is decidable by an equisatisfiable translation. The key idea is that all the formulas expressing heaps in SL are reduced into the first-order theory. Since the model of first-order set has no heap ingredient, we use extra variables to simulate it. Let H denote a vector of n tuples of pointer variables, i.e., $H = ((H_{1,1}, \ldots, H_{1,m_1}), \ldots, (H_{n,1}, \ldots, H_{n,m_n}))$. The translations $f(\phi, H)$ and $F(P, H)$ generate an equisatisfiable state formula ϕ' to ϕ and an equisatisfiable temporal formula P' to P respectively. Both ϕ' and P' belong to PTL. The detailed definitions of f and F can be found in [4]. Analogous to MSVL, by using a very similar approach, we can construct a graph structure (also called NFG) that explicitly characterizes the model for any reduced PPTL$^{\text{SL}}$ formula. The detailed proofs are given in [4], here we only give a brief summary.

Theorem 2. PPTL$^{\text{SL}}$ is decidable, and its complexity is the same as PPTL, i.e., non-elementary.

5 Model Checking with MSVL and PPTL$^{\text{SL}}$

Our model checking approach is similar to the traditional automata-based model checking except that ours is based on NFG. The starting point is a pointer program modeled by an MSVL program M, and a PPTL$^{\text{SL}}$ formula P that formalizes the desired heap evolution property on M.

In general, the model checking procedure in this framework first creates the NFG G_M of an MSVL program and the NFG $G_{\neg P}$ of the negation of the input PPTL$^{\text{SL}}$ formula, then constructs the product G of the two NFGs. The nodes (edges) of G are conjunctions of nodes (edges) in G_M and $G_{\neg P}$. If there exist a valid path in G, a counterexample is found, otherwise M satisfies P.

We have developed a unified model checking tool (prototype) based on the approach presented in this paper. As shown Fig. 1, the tool structure consists of three essential modules: MSV, PPTL$^{\text{SL}}$ solver and unified model checker. An MSVL program M is feeded into MSV, and a property P is given to the PPTL$^{\text{SL}}$ solver. MSV constructs the NFG of M and PPTL$^{\text{SL}}$ solver builds the NFG of $\neg P$. The Model checker does not try to build the complete production of the two NFGs in practice. Instead, it works in "on the fly" manner and tries to find one valid path as early as possible. The SMT solver Z3 is called when checking whether an edge in the product NFG is satisfied or not. We have successfully verified the producer-consumer program with respect to the corresponding heap evolution properties mentioned in Sect. 4.

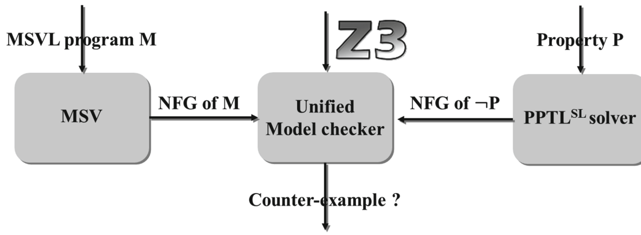

Fig. 1. Tool architecture.

6 Conclusion

In this paper, we propose a unified model checking approach with MSVL and $\mathrm{PPTL}^{\mathrm{SL}}$. We can apply this approach to verify heap evolution properties of pointer programs, including both safety and liveness properties on heap structures. Since $\mathrm{PPTL}^{\mathrm{SL}}$ is able to be reduced to a subset of PTL so that programs and properties both belong to the same logic framework which makes the verification more convenient. We have developed a model checking tool that exploits the SMT solver Z3 as the verification engine. In the future, more case studies on various heap structures in addition to singly-linked structures will be carried out.

References

1. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science, Computer Society, pp. 55–74. IEEE, Washington (2002)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
3. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM (JACM) **58**(6), 26 (2011)
4. Lu, X., Duan, Z., Tian, C.: Extending PPTL for verifying heap evolution properties. arXiv preprint arXiv:1507.08426 (2015)
5. Duan, Z.: An extended interval temporal logic and a framing technique for temporal logic programming. Ph.D. thesis, University of Newcastle upon Tyne (1996)
6. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Sci. Comput. Program. **70**(1), 31–61 (2008)
7. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008). doi:10.1007/978-3-540-88194-0_12
8. Yahav, E., Reps, T.W., Sagiv, S., Wilhelm, R.: Verifying temporal heap properties specified via evolution logic. Logic J. IGPL **14**(5), 755–783 (2006)
9. Distefano, D., Katoen, J.-P., Rensink, A.: Safety and liveness in concurrent pointer programs. In: Boer, F.S., Bonsangue, M.M., Graf, S., Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 280–312. Springer, Heidelberg (2006). doi:10.1007/11804192_14

10. Rieger, S.: Verification of pointer programs. Ph.D. thesis, RWTH Aachen University (2009)
11. del Mar Gallardo, M., Merino, P., Sanán, D.: Model checking dynamic memory allocation in operating systems. J. Autom. Reasoning **42**(2–4), 229–264 (2009)
12. Zhang, N., Duan, Z., Tian, C.: Extending MSVL with function calls. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 446–458. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11737-9_29
13. Wang, X., Duan, Z., Zhao, L.: Formalizing and implementing types in MSVL. In: Liu, S., Duan, Z. (eds.) SOFL+MSVL 2013. LNCS, vol. 8332, pp. 60–73. Springer, Heidelberg (2014)