Verification of Snapshot Isolation in Transactional Memory Java Programs

Ricardo J. Dias[1], Dino Distefano[2], João Costa Seco[1], and João M. Lourenço[1⋆]

[1] CITI, Universidade Nova de Lisboa, Portugal
{rjfd,joao.seco,joao.lourenco}@di.fct.unl.pt
[2] Queen Mary University of London, UK
ddino@eecs.qmul.ac.uk

Abstract. This paper presents an automatic verification technique for transactional memory Java programs executing under snapshot isolation level. We certify which transactions in a program are safe to execute under snapshot isolation without triggering the *write-skew* anomaly, opening the way to run-time optimizations that may lead to considerable performance enhancements.
Our work builds on a novel deep-heap analysis technique based on separation logic to statically approximate the read- and write-sets of a transactional memory Java program.
We implement our technique and apply our tool to a set of micro benchmarks and also to one benchmark of the STAMP package. We corroborate known results, certifying some of the examples for safe execution under snapshot isolation by proving the absence of *write-skew* anomalies. In other cases our analysis has identified transactions that potentially trigger previously unknown *write-skew* anomalies.

1 Introduction

Full-fledged Software Transactional Memory (STM) [18,11] usually provides strict isolation between transactions and full serializability semantics. Alternative relaxed semantics approaches, based on weaker isolation levels that allow transactions to interfere and to generate non-serializable execution schedules, are known to perform considerably better in some cases. The interference among non-serializable transactions are commonly known as *serializability anomalies* [2].

Snapshot Isolation (SI) [2] is a well known relaxed isolation level widely used in databases, where each transaction executes with relation to a private copy of the system state — a snapshot — taken at the beginning of the transaction and stored in a local buffer. All write operations are kept pending in the local buffer until they are committed in the global state. Reading modified items always refer

⋆ This work was partially supported by the Euro-TM EU COST Action IC1001, the Portuguese national research projects RepComp (PTDC/EIA-EIA/108963/2008), Synergy-VM (PTDC/EIA-EIA/113613/2009) and StreamLine (PTDC/EIA-CCO/104583/2008), and the research grant SFRH/BD/41765/2007. Distefano was supported by the Royal Academy of Engineering.

to the pending values in the local buffer. In all cases, committing transactions obey the general First-Commiter-Wins rule. This rule states that a transaction A can only commit if no other concurrent transaction B has committed modifications to data items pending to be committed by transaction A. Hence, for any two concurrent transactions modifying the same data item, only the first one to commit will succeed.

Tracking memory operations introduces some overhead, and TM systems running under serializable isolation level must track both memory read and write accesses, incurring in considerable performance penalties. Validating transactions in SI only requires to check if any two concurrent transaction wrote at a common data item. Hence the runtime system only needs to track the memory write accesses per transaction, ignoring the read accesses, possibly boosting the overall performance of the transactional runtime, as shown in [7].

Although appealing for performance reasons, the application of SI may lead to non-serializable executions, resulting in a serializability anomaly called *write-skew*. For instance, a *write-skew*, arises in the following example of two statements running in concurrent transactions

$$x := x + y \quad || \quad y := y + x$$

In this case, it is possible to find a trace of execution that is not serializable and yields unexpected results. In general, this anomaly occurs when two transactions are writing on disjoint memory locations (x and y) but are also reading data that is being modified by the other.

In this paper we present a verification technique for STM Java programs that statically detects if any two transactions may cause a *write-skew* anomaly. The application of the proposed technique may be used to optimize program execution, by letting memory transactions run in snapshot isolation whenever possible, and by explicitly requiring the full serializability semantics otherwise. Our technique performs deep-heap analysis (also called shape analysis) based on separation logic [16,8] to compute memory locations in the read- and write-sets for each distinguished transaction in a Java program. The analysis only requires the specification of the state of the heap for each transaction and is able to automatically compute *loop invariants* during the analysis. Our analysis computes read and write-sets of transactions using *heap paths*, which capture dereferences through field labels, choice and repetition.

For instance, a *heap path* of the form $x.(\mathit{left} \mid \mathit{right})^*.\mathit{right}$ describes the access to a field labeled *right*, on a memory location reachable from variable x after a number of dereferences through *left* or *right* fields.

We implemented a tool with the proposed techniques, called StarTM, which allows to analyze Java Bytecode programs extended with STM annotations. To validate our approach, we tested implementations of a transactional Linked List and of a transactional Binary Search Tree, and also of a Java implementation of the STAMP Intruder benchmark [5]. Our results confirm that i) it is possible to safely execute concurrent transactions of a Linked List under snapshot isolation with noticeable performance improvements, supporting the arguments of

[17]; ii) it is possible to build a transactional insert method in a Binary Search Tree that is safe to execute under SI; and iii) our automatic analysis of the STAMP Intruder benchmark found a new *write-skew* anomaly in the existing implementation.

We impose some limitations on the programs for which our approach is able to guarantee the absence of *write-skew* anomalies. We only support acyclic data structures, such as tree-like data structures, and only detect *write-skews* between pairs of transactions.

The main contributions of this paper are:

- The first program verification technique to statically detect the *write-skew* anomaly in transactional memory programs;
- The first technique able to verify transactional memory programs even in presence of deep-heap manipulation thanks to the use of shape analysis techniques;
- A model that captures fine-grained manipulation of memory locations based on *heap paths*;
- An implementation of our technique and the application of the tool to a set of intricate examples.

The remainder of the paper describes the theory of our analysis technique and the validation experiments. We start by describing a step-by-step example of applying StarTM to a simple example in section 2. We then present the core language, in section 3, and the abstract domain for the analysis procedure in section 4. In section 5, we present the symbolic execution of programs against the abstract state representation. We finalize the paper by presenting some experimental results in Section 6 and comparing our approach with others in Section 7.

2 StarTM by Example

StarTM analyzes Java multithreaded programs that make use of memory transactions. The scope of a memory transaction is defined by the scope of a Java method annotated with **@Atomic**, which in our case requires a mandatory argument with an abstract description of the initial state of the heap. Other methods called inside a transactional method do not require this initial description, as it is automatically computed by the symbolic execution.

To describe the abstract state of the heap, we use a subset of separation logic formulae composed of a set of predicates — among which a *points-to* ($\mapsto$) predicate — separated by the special separation conjunction ($*$) typical of separation logic. The user can define new predicates in a proper scripting language and also define abstraction functions which, in case of infinite state spaces, allows the analysis to converge. The abstraction function is defined by a set of abstraction rules as in the jStar tool [9]. The user defined predicates and abstraction rules are described in separate files and are associated with the transactions' code by the class annotations **@Predicates** and **@Abstractions**, which receive as argument the corresponding file names.

```
1  @Predicates(file="list_pred.sl")
2  @Abstractions(file="list_abs.sl")
3  public class List { public class Node{ ... } ... }
```

```
23 @Atomic(state= "| this -> [head:h']
24     * List(h', nil)")
25 public void add(int value) {
26   boolean result;
27   Node prev = head;
28   Node next = prev.getNext();
29   while (next.getValue() < value) {
30     prev = next;
31     next = prev.getNext();
32   }
33   if (next.getValue() != value) {
34     Node n = new Node(value, next);
35     prev.setNext(n);
36   }
37 }
```

```
39 @Atomic(state= "| this -> [head:h']
40     * List(h', nil)")
41 public void remove(int value) {
42   boolean result;
43   Node prev = head;
44   Node next = prev.getNext();
45   while (next.getValue() < value) {
46     prev = next;
47     next = prev.getNext();
48   }
49   if (next.getValue() == value) {
50     prev.setNext(next.getNext());
51   }
52 }
```

Fig. 1. Order Linked List code

```
// list_pred.sl file

/*** Predicate definition ***/
Node(+x,-n) <=> x -> [next:n] ;;

List(+x,-y) <=> x != y /\
  ( Node(x,y) \/ E z'. Node(x,z') *
     List(z',y) );;
```

```
// list_abs.sl file
/*** Abstractions definition ***/
Node(x, y') * Node(y',z)~~>List(x, z):
  y' nin context;
  y' nin x;
  y' nin z
;;
...
List(x,y') * Node(y',z)~~>List(x, z):
  y' nin context;
  y' nin x;
  y' nin z
;;
```

Fig. 2. Predicates and Abstraction rules of Linked List

We use as running example the implementation of an ordered singly linked list, adapted from the DeuceSTM [13] samples, shown in Fig. 1. The corresponding predicates and abstractions rules are defined in Fig. 2. The predicate `Node(+x,-y)` defined in Fig. 2 by

$$Node(x, y) \Leftrightarrow x \mapsto [next : y]$$

is valid if variable x points to a memory location where the corresponding *next* field points to the same location as variable y, or both the *next* field and y point to nil. Predicate `List(+x,-y)` defined by

$$List(x, y) \Leftrightarrow x \neq y \wedge (Node(x, y) \vee \exists z'. Node(x, z') * List(z', y))$$

is valid if variables x and y point to distinct memory locations and there is a chain of nodes leading from the memory location pointed by x to the memory location pointed by y. The predicate is also valid when both y and the last node in the chain point to nil.

```
# Method boolean add(int value)
Result 1:
ReadSet:   { this.head.(next)[*A].next.value }
WriteSet>: { }
WriteSet<: { }

Result 2:
ReadSet:   { this.head.(next)[*B].next.value }
WriteSet>: { this.head.(next)[*B].next }
WriteSet<: { this.head.(next)[*B].next }


# Method boolean remove(int value)
Result 1:
ReadSet:   { this.head.(next)[*C].next.value }
WriteSet>: { }
WriteSet<: { }

Result 2:
ReadSet:   { this.head.(next)[*D].next.value, this.head.(next)[*D].next.next}
WriteSet>: { this.head.(next)[*D].next }
WriteSet<: { this.head.(next)[*D].next }
```

Fig. 3. Sample of StarTM result output for the Linked List example

The modifiers `+` and `-` of the predicate parameters indicate that the corresponding parameter points to a memory location respectively inside or outside of the memory region defined by the predicate. A more precise definition of these modifiers is presented in Section 4.2.

In Fig. 1, we annotate the add(**int**) and remove(**int**) methods as transactions with the initial state described by the following formula:

```
| this->[head:h'] * List(h',nil)
```

This formula states that variable **this** points to a memory location that contains an object of class List, and whose field *head* points to the same memory location pointed by the existential variable[3] h', which is the entry point of a list with at least one element.

StarTM performs an inter-procedural symbolic execution of the program. The abstract domain used by the symbolic execution is composed by a separation logic formula describing the abstract heap structure, and the abstract read- and write-sets. The abstract write-set is defined by two sets: a *may* write-set and a *must* write-set. As the naming implies one over-approximates, and the other under-approximates the possible real write-set. The abstract read-set is an over-approximation of the possible real read-set. The read- and write-sets are defined as sets of *heap paths*. A memory location is represented by its path, in terms of field accesses, beginning from some shared variable. We assume that the parameters of a transactional method and the instance variable `this` are shared in the context of that transaction.

[3] Throughout this paper we consider primed variables as implicitly existentially quantified.

The sample of the results of our analysis, depicted in Fig. 3, includes two possible pairs of read- and write-sets for method `add(int)`. The *may* write-set is denoted by label `WriteSet>` and the *must* write-set is denoted by label `WriteSet<`. The first result has an empty write-set[4], and thus corresponds to a read-only execution of the method `add(int)`, where the *heap path* in the read-set can be interpreted as follows. The *heap path* `this.head.(next)[*A].next.value` asserts that method `add(int)` reads the *head* field from the memory location pointed by variable *this* and following the memory location pointed by *head* it reads the *next* field, then for each memory location it reads the *next* and *value* fields and hops to the next memory location through the *next* field. In the last memory location accessed it only reads the *value* field. In general, we can interpret the meaning of an abstract read-set as all the memory locations represented by the *heap paths* present in the read-set and also by their prefixes.

The star ($*$) operator has always a label attached, in case of `[*A]`, the label is `A`. This label is used to identify the subpath guarded by the star and can be interpreted, in this case, as $A = (next)^*$. This label is existentially quantified in a pair of read- and write-sets.

The second pair of read- and write-sets of method `add(int)` in Fig. 3 contains the same read-set and a different write-set. In this case the *may* and *must* write-sets are equal. The *heap path* `this.head.(next)[*B].next` asserts that the *next* field, of the memory location represented by the path $this.head.(next)^*_B$, was written.

It is important to notice that the interpretations of the read- and write-set are different. In the read-set we consider that all the path prefixes of all *heap path* expressions were read, while in the write-set we consider that there was a single write operation in the last field of each *heap path* expression.

The *may* write-set may contain *heap paths* of the form $this.head.(next)^{\bar{*}}_B$. In this case, the interpretation of this expression is that the field *next* is written in every memory location represented by the path $this.head.(next)^*_B$. More details on *heap path* expressions are given in Section 4.2.

The analysis also originates two possible results for method `remove(int)`. The first result for this method is similar to the first result for method `add(int)`. In the second result for method `remove(int)`, the field *next* is read for all memory locations including the last memory location where field *value* was accessed, since the star label is the same in the two *heap path* expressions in the read-set. The write-set is the same as in the `add(int)` method.

We can now check for the possible occurrence of a *write-skew* anomaly. We define a *write-skew* condition as:

Definition 1 (Abstract Write-Skew). *Let* T_1 *and* T_2 *be two transactions, and let* $\mathcal{R}_i$, $\mathcal{W}_i^{>}$ *and* $\mathcal{W}_i^{<}$ $(i = 1, 2)$ *be their corresponding abstract read-,* may *write- and* must *write-sets. There is a write-skew anomaly if*

$$\mathcal{R}_1 \cap \mathcal{W}_2^{>} \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{>} \cap \mathcal{R}_2 \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{<} \cap \mathcal{W}_2^{<} = \emptyset$$

[4] If the context is not ambiguous we will always refer to both the *may* and *must* write-sets.

```
# Method boolean remove(int value)
Result 2:
ReadSet:   { this.head.(next)[*D].next.value, this.head.(next)[*D].next.next }
WriteSet>: { this.head.(next)[*D].next, this.head.(next)[*D].next.next }
WriteSet<: { this.head.(next)[*D].next, this.head.(next)[*D].next.next }
```

Fig. 4. Sample of StarTM result output for corrected remove(**int**) method

We will consider that each result (a pair of a read- and a write-set) corresponds to a single transaction instance. From the above condition we may trivially ignore the results with an empty write-set. Hence, only result pairs with non-empty write-sets need to be checked.

We denote the second result of the add(**int**) method as T_{add}, and the second result of the remove(**int**) method as T_{rem}. To detect the possible existence of a *write-skew* we need to check the following pairs:

$$(\mathrm{T}_{add}, \mathrm{T}_{add}),\ (\mathrm{T}_{rem}, \mathrm{T}_{rem}),\ (\mathrm{T}_{add}, \mathrm{T}_{rem})$$

Let's examine in detail the pair $(\mathrm{T}_{add}, \mathrm{T}_{rem})$. We simplify the description of the read-set of each transaction by ignoring the field `value`, since neither transactions writes to that field and thus we will focus only on interactions with the field `next`. We assume that the shared variable `this` points to the same object in both transactions, otherwise no conflicts would ever arise. The read- and write-set for transactions T_{add}, and T_{rem} (relative to field *next*) are

$$\begin{aligned}
\mathcal{R}_{add} &= \{\mathit{this.head},\ \mathit{this.head.B},\ \mathit{this.head.B.next}\} \\
\mathcal{W}^{>}_{add} &= \mathcal{W}^{<}_{add} = \{\mathit{this.head.B.next}\} \\
\mathcal{R}_{rem} &= \{\mathit{this.head},\ \mathit{this.head.D},\ \mathit{this.head.D.next},\ \mathit{this.head.D.next.next}\} \\
\mathcal{W}^{>}_{rem} &= \mathcal{W}^{<}_{rem} = \{\mathit{this.head.D.next}\}
\end{aligned}$$

Given these read- and write-sets, if an instantiation of B and D exist that satisfies the *write-skew* condition then the concurrent execution of these two transactions could possibly cause a *write-skew* anomaly. In this particular case, the assertion $B = D.next$, which means that the memory locations represented by B are the same as the ones represented by $D.next$, satisfies the *write-skew* condition.

To correct the list implementation from triggering a *write-skew* anomaly one can add the additional write operation next.setNext(**null**) between lines 50 and 51 of the code shown in Fig. 1. This write operation, although unnecessary in terms of the list semantics, is essential to make the list implementation safe under snapshot isolation as we shall see. Given this new implementation, the result of the analysis by StarTM is depicted in Fig. 4. Notice that the write-set has two *heap paths* describing that the transaction writes the *next* field of the penultimate and last memory locations. Now, the new read- and write-set for

$$
\begin{array}{lll}
e & ::= & (\textit{expression}) \\
 & x & (\text{variables}) \\
 & \mid \ \mathsf{null} & (\text{null value}) \\
A & ::= & (\textit{assignments}) \\
 & x := e & (\text{local}) \\
 & \mid \ x := y.f & (\text{heap read}) \\
 & \mid \ x := \mathit{fun}(\vec{y}) & (\text{function call}) \\
 & \mid \ x.f := e & (\text{heap write}) \\
 & \mid \ x := \mathsf{new} & (\text{allocation})
\end{array}
\qquad
\begin{array}{lll}
b & ::= & (\textit{boolean exp}) \\
 & e \oplus_b e & (\text{boolean op}) \\
 & \mid \ \mathsf{true} \mid \mathsf{false} & (\text{bool values}) \\
S & ::= & (\textit{statements}) \\
 & S \ ; S & (\text{sequence}) \\
 & \mid \ A & (\text{assignment}) \\
 & \mid \ \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S & (\text{conditional}) \\
 & \mid \ \mathsf{while}\ b\ \mathsf{do}\ S & (\text{loop}) \\
 & \mid \ \mathsf{return}\ e & (\text{return}) \\
 & \mid \ \mathsf{skip} & (\text{Skip})
\end{array}
$$

$$P ::= \mathit{fun}(\vec{x}) = S \ \mid \ P \ (\textit{program})$$

Fig. 5. Core language syntax for programs

transactions T_{add}, and T_{rem} (relative to field *next*) are

$$
\begin{aligned}
\mathcal{R}_{add} &= \{\mathit{this.head},\ \mathit{this.head.B},\ \mathit{this.head.B.next}\} \\
\mathcal{W}^{>}_{add} &= \mathcal{W}^{<}_{add} = \{\mathit{this.head.B.next}\} \\
\mathcal{R}_{rem} &= \{\mathit{this.head},\ \mathit{this.head.D},\ \mathit{this.head.D.next}, \mathit{this.head.D.next.next}\} \\
\mathcal{W}^{>}_{rem} &= \mathcal{W}^{<}_{rem} = \{\mathit{this.head.D.next},\ \mathit{this.head.D.next.next}\}
\end{aligned}
$$

In this case, it is not possible to find an instantiation for B and D, such that the *write-skew* condition is true. Hence, these transactions can execute concurrently under SI without ever triggering the *write-skew* anomaly.

3 Core Language

In this section we define a core language to support our static analysis. We include the subset of Java that captures essential features such as object creation (new), field dereferencing ($x.f$), assignment ($x := e$), and function invocation ($\mathit{fun}(\vec{x})$). The syntax of the language is defined by the grammar in Fig. 5. A program in this language is a set of function definitions. We do not explicitly represent transactions nor an entry point in the syntax, and we assume that all functions are transactions that can be called concurrently.

We assume a countable set of program variables Vars (ranged over by $x, y, \ldots$), a set of shared variables SVars $\subseteq$ Vars, a countable disjoint set of primed variables Vars$'$ (ranged over by $x', y', \ldots$), a countable set of locations Locations, and a finite set of field names Fields. The operational semantics for the language is defined over configurations of the form $\langle s, h, S\rangle$, where $s \in$ Stacks is a stack (a mapping from variables to values), $h \in$ Heaps is a (concrete) heap (a mapping

$$
\begin{array}{lll}
e & ::= & (\textit{expressions}) \\
 & x, y, \ldots \in \mathsf{Vars} & (\text{program variables}) \\
 & \mid\ x', y', \ldots \in \mathsf{Vars}' & (\text{existential variables}) \\
 & \mid\ \mathsf{nil} & (\text{null value}) \\
\rho & ::= f_1 : e,\ \ldots,\ f_n : e & (\textit{record}) \\
 & & \\
S & ::= e \mapsto [\rho] \ \mid\ p(\vec{e}) & (\textit{spatial predicates}) \\
P & ::= e = e & (\textit{pure predicates}) \\
\Pi & ::= \mathsf{true} \ \mid\ P \wedge \Pi & (\textit{pure part}) \\
\Sigma & ::= \mathsf{emp} \ \mid\ S * \Sigma & (\textit{spatial part}) \\
 & & \\
\mathcal{H} & ::= \Pi | \Sigma & (\textit{symbolic heap})
\end{array}
$$

Fig. 6. Separation logic syntax

from locations to values through field labels).

$$
\begin{aligned}
\mathsf{Values} &= \mathsf{Locations} \cup \{\mathsf{nil}\} \\
\mathsf{Stacks} &= (\mathsf{Vars} \cup \mathsf{Vars}') \to \mathsf{Values} \\
\mathsf{Heaps} &= \mathsf{Locations} \rightharpoonup_{\mathit{fin}} (\mathsf{Fields} \to \mathsf{Values})
\end{aligned}
$$

The small step structural operational semantics is the standard for this kind of imperative language defined by the reduction relation $\langle s, h, S \rangle \Longrightarrow \langle s', h', S' \rangle$.

4 Symbolic States

In the symbolic execution a *symbolic state* is of the form $(\mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W})$: where $\mathcal{H}$ is a *symbolic heap*, defined using a fragment of separation logic formulae, $\mathcal{M}$ is a map between variables and *heap path* expressions, and $\mathcal{R}$ and $\mathcal{W}$ are read- and write-sets. The write-set $\mathcal{W}$ in our analysis is actually composed by two sets: a *may* write-set, denoted by $\mathcal{W}^{>}$, which over-approximates the concrete write-set, and a *must* write-set, denoted by $\mathcal{W}^{<}$, which under-approximates the concrete write-set.

The fragment of separation logic formulae that we use to describe symbolic heaps is defined by the grammar in Fig. 6. Satisfaction of a formula $\mathcal{H}$ by a stack s and heap h is denoted $s, h \models \mathcal{H}$ and defined by structural induction on $\mathcal{H}$ in Fig. 7. There, $[\![p]\!]$ is as usual a component of the least fixed point of a monotone operator constructed from a inductive definition set; see [3] for details. In this heap model a location maps to a record of values. The formula $e \mapsto [\rho]$ can mention any number of fields in ρ, and the values of the remaining fields are implicitly existentially quantified.

4.1 Symbolic Heaps

Symbolic heaps are abstract models of the heap of the form $\mathcal{H} = \Pi | \Sigma$ where Π is called the *pure part* and Σ is called the *spatial part*. We use prime variables

$$
\begin{array}{rl}
s, h \models \mathsf{emp} & \text{iff } \mathsf{dom}(h) = \emptyset \\
s, h \models x \mapsto [f_1 : e_1, \ , f_n : e_n] & \text{iff } h = [s(x) \mapsto r] \text{ where } r(f_i) = s(e_i) \text{ for } i \in [1, n] \\
s, h \models p(\vec{e}) & \text{iff } (s(\vec{e}), h) \in [\![p]\!] \\
s, h \models \Sigma_0 * \Sigma_1 & \text{iff } \exists h_0, h_1.\ h = h_0 * h_1 \text{ and } s, h_0 \models \Sigma_0 \text{ and } s, h_1 \models \Sigma_1 \\
s, h \models e_1 = e_2 & \text{iff } s(e_1) = s(e_2) \\
s, h \models \Pi_1 \wedge \Pi_2 & \text{iff } s, h \models \Pi_1 \text{ and } s, h \models \Pi_2 \\
\\
s, h \models \Pi | \Sigma & \text{iff } \exists \vec{v'}.\ \left(s(\vec{x'} \mapsto \vec{v'}), h \models \Pi\right) \text{ and } \left(s(\vec{x'} \mapsto \vec{v'}), h \models \Sigma\right) \\
 & \quad \text{where } \vec{x'} \text{ is the collection of existential variables} \\
 & \quad \text{in } \Pi | \Sigma
\end{array}
$$

Fig. 7. Separation Logic semantics

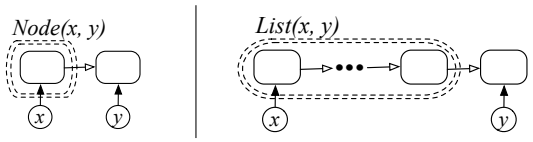

Fig. 8. Graph representation of the $Node(x, y)$ and $List(x, y)$ predicates

$(x'_1, \ldots, x'_n)$ to implicitly denote existentially quantified variables that occur in $\Pi|\Sigma$. The pure part Π is a conjunction of pure predicates which states facts about the stack variables and existential variables (e.g., $x = \mathsf{nil}$). The spatial part Σ is the $*$ conjunction of spatial predicates, i.e., related to heap facts. In separation logic, the formula $S_1 * S_2$ holds in a heap that can be split into two disjoint parts, one of them described exclusively by S_1 and the other described exclusively by S_2.

In symbolic heaps, memory locations are either pointed directly by program variables (e.g., v) or existential variables (e.g., v'), or they are abstracted by predicates. Predicates are abstractions for the graph-like structure of a set of memory locations. For example, the predicate $Node(x, y)$, in Fig. 8, abstracts a single memory location pointed by variable x, while the predicate $List(x, y)$ abstracts a set of an unbound number of memory locations, where each location is linked to another location of the set by the *next* field.

A predicate $p(\vec{e})$ has at least one parameter, from its parameter set, that is the entry point for reaching every memory location that the predicate abstracts. We denote this kind of parameter as *entry* parameters. Also, there is a subset of parameters that correspond to the exit points of the memory region abstracted by the predicate. These parameters denote variables pointing to memory locations that are outside the predicate but the predicate has memory locations with links to these *outsider* locations. In Fig. 8 we can observe that the predicate $List(x, y)$ has one *entry* parameter x and one *exit* parameter y. Users of

$$
\begin{array}{lll}
H ::= v \mid v.P & & (\textit{heap path}) \\
P ::= f \mid f.P \mid C_A^*.P & & (\textit{subpath}) \\
C ::= f \mid f \text{ “|” } C & & (\text{choice})
\end{array}
$$

$$
\begin{array}{rlr}
\mathcal{S}[\![v]\!]_{s,h,l} = \{l'\} \quad \mathcal{S}[\![v.P]\!]_{s,h,l} = \mathcal{S}[\![P]\!]_{s,h,l'} & & \text{where } l' = s(v) \\
\mathcal{S}[\![f]\!]_{s,h,l} = \{l'\} \quad \mathcal{S}[\![f.P]\!]_{s,h,l} = \mathcal{S}[\![P]\!]_{s,h,l'} & & \text{where } l' = h(l,f) \\
\mathcal{S}[\![C^*.P]\!]_{s,h,l} = \mathcal{S}[\![f_1.C^*.P]\!]_{s,h,l} \cup ... \cup \mathcal{S}[\![f_n.C^*.P]\!]_{s,h,l} \cup \mathcal{S}[\![P]\!]_{s,h,l} & & \text{where } C = f_1|...|f_n
\end{array}
$$

Fig. 9. *Heap Path* syntax and semantics

StarTM are required to indicate which parameters of a predicate are *entry* or *exit* by prefixing them with the unary operators `+` and `-`, denoting *entry* and *exit* respectively. In the definition of our analysis we can query if a parameter a of a predicate p is of *entry* or *exit* type with the $\delta_p^+(a)$ or $\delta_p^-(a)$ operators respectively. For the special case of predicate ($\mapsto$), we always consider that the variable on its left side is an *entry* parameter and the variables on its right side are *exit* parameters.

4.2 Heap Paths

We are going to represent a memory location as a sequence of fields, starting from a program variable. If we successively dereference the field labels that appear in the sequence, we reach the memory location denoted by the sequence. We call these sequences of field labels, prefixed by a variable name, a *heap path*. For instance, the path $x.left.right$, denotes the memory location that is reachable by dereferencing the field *left* of the location pointed by variable x, and by dereferencing the field *right* of the location represented by $x.left$.

We can also represent sequences of field dereferences in a *heap path* by using the Kleene star ($*$) and choice ($|$) operators. For instance, the path $x.(left \mid right)^*$ denotes a memory location that can be reached by starting on variable x and then dereferencing either the *left* or *right* field on each visited memory location.

The syntax of *heap paths* is depicted in Fig. 9 and corresponds to a very restrictive subset of the regular expressions syntax. A *heap path* always starts with a variable name (v) followed by sequences of field labels (f), repeating subpath expressions under a Kleene operator (C^*), and choices of field labels (C). We syntactically restrict *heap paths*, with respect to regular expressions, by only allowing choices of field labels guarded by a Kleene operator, and repetitions of choices of single field labels (not sequences). For instance, the path $x.(left \mid right^*)$ is not a valid *heap path* expression.

Each repeating subpath is always associated with a label. This is used to identify the subpath guarded by the star and we can rewrite $C_A^*.P$ as $A.P$ where $A = C^*$. As we shall see later, this label will be used to identify subpath expressions that denote the same concrete path in the heap. We may also denote the repetition sequence with a bar on top of the star, e.g., $x.C_A^{\bar{*}}$. This will be

$$
\begin{array}{lcl}
\Phi(x \mapsto [f_1 : y, \ldots, f_n : z], x, y) & = & x.f_1 \\
\Phi(p(\vec{i}, \vec{o}), x, y) & = & hp \text{ where } x \in \vec{i} \wedge \delta_p^+(x) \wedge y \in \vec{o} \wedge \delta_p^-(y) \wedge hp = \Gamma(p, x, y) \\
\Phi(S * S', x, y) & = & \mathsf{concat}(\Phi(S, x, z), \Phi(S', z, y)) \\
 & & \text{where exists path from } x \text{ to } z \text{ in } S \text{ and from } z \text{ to } y \text{ in } S' \\
\mathsf{concat}(x.P, z.P') = x.P.P' & &
\end{array}
$$

Fig. 10. Rules for transforming a symbolic heap into a heap path

used to distinguish between different interpretations, of *heap path* expressions contained in read- and write-sets.

We now define the semantics of *heap paths* with relation to concrete stacks and heaps through function $\mathcal{S}[\![H]\!]_{s,h,l}$ in Fig. 9. According to this definition a *heap path* expression denotes the set of all memory locations that can be reachable by following it in a concrete memory, $\mathcal{S}[\![H]\!] \subseteq \mathsf{Locations}$. Abstract read- or write-sets are sets of *heap paths*. We write HPaths for the set of all *heap path* expressions.

In the following developments we interpret read-sets, may write-sets, and must write-sets in three different ways. For read-sets we always consider the saturation of the read-set with the denotations of all prefixes of its heap-paths. For *must* write-sets we consider one under-approximation where a heap-path H represents exactly one location in the set $\mathcal{S}[\![H]\!]$. For *may* write-sets we consider the over-approximation by saturating the set with the expansion of the $\bar{*}$ repetition annotation. For instance, a heap path expression $x.C^{\bar{*}}.f$ in a may *write-set*, denotes write operations on all fields f for all locations of the set $\mathcal{S}[\![x.C^{\bar{*}}]\!]$.

4.3 From Symbolic Heaps to Heap Paths

During the symbolic execution, we generate *heap paths* based on the information given by the symbolic heap. Recall that the only information given by the user to the verification tool is a description of the state at the beginning of the transaction using a symbolic heap, everything else is inferred.

Given a memory location l pointed by some variable x, if there is a path in the symbolic heap from some other variable s, where $s \in \mathsf{SVars}$, to variable x, then we can generate a *heap path* that represents the path from the shared variable s to the memory location l. Moreover, the computation of a *heap path* from the symbolic heap requires a transformation function that given a predicate and its arguments returns a *heap path*. In this case, the separation conjunction operator ($*$) corresponds to the concatenation in the *heap path*. See Fig. 10 for the whole set of transformation rules. Function $\mathsf{concat}(x.P, z.P')$ concatenates the path described by P' to the *heap path* $x.P$. Note that this concatenation is sound, given the pre-condition that $x.P$ represents the same memory location as variable z, which is true in the case above.

The rule for transforming a predicate $p(\vec{i}, \vec{o})$ into a *heap path* relies on a function Γ that returns a *heap path* given a predicate and a pair of variables. A predicate definition can be transformed into a DFA (Deterministic Finite Automaton) where states correspond to predicates and transitions' labels correspond to fields. Then, we can generate a *heap path* expression, from the automaton, using well know automata to regular expressions transformation techniques. Consider the example of a *heap path* generated for the list segment predicate:

Example 1 (Heap Path of the List Segment Predicate).

$$\mathit{List}(x, y) \Leftrightarrow x \neq y \wedge \left(x \mapsto [\mathit{next} : y] \vee \exists z' .\ x \mapsto [\mathit{next} : z'] * \mathit{List}(z', y)\right)$$

Given the *List* predicate definition, the *heap path* that represents the memory location pointed by y reachable from x is:

$$\Gamma(\mathit{List}(x, y), x, y) = x\ .\ \mathit{next}_A^+$$

We abbreviate repeating sequences with at least one field label using symbol $+$ (e.g. next^+). The label A is fresh in the context of the symbolic state where the *heap path* is computed. Notice that *heap path* expressions containing repetitions and choices are only generated when transforming recursive predicates into *heap paths*.

5 Symbolic Execution

Next, we define the symbolic execution for the core language presented in Section 3 taking inspiration from [8]. In our case, the symbolic execution defines the effect of statements on symbolic states composed by a symbolic heap, a path map, and a read- and write-set. We represent a symbolic state as: $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W} \rangle \in (\mathsf{SHeaps} \times (\mathsf{Vars} \rightharpoonup \mathsf{HPaths}) \times \mathsf{Rs} \times \mathsf{Ws})$ where SHeaps is the set of all symbolic heaps, $(\mathsf{Vars} \rightharpoonup \mathsf{HPaths})$ is the map between program variables and *heap path* expressions, Rs is the set of all read-sets, and Ws is the set of pairs of all *may* and *must* write-sets. We write $\mathsf{SStates}$ for denoting the set of all symbolic states.

The path map $\mathcal{M}$ is a map that associates variables to *heap path* expressions. In each state of the symbolic execution, a variable x in this map is associated with a *heap path* expression that represents the memory location pointed by x. The purpose of this map is to keep a *heap path* expression less abstract than the one that we can capture from the symbolic heap. For instance, in the map, we may have the information that we only accessed the *left* field of each node of a tree, but from the symbolic heap we get the information that we accessed the *left* or *right* fields in each node. The symbolic execution will always maintain the invariant $S_p \subseteq G_p$ where S_p is the *heap path* in the path map and G_p is the *heap path* from the symbolic heap, for a variable x. The subset relation means that all paths described by S_p are described by G_p.

Each transactional method is annotated with the **@Atomic** annotation describing the initial symbolic heaps for that transaction. The symbolic execution will analyze only transactional methods and all methods present in the invocation tree that occurs inside their body. In the beginning of the analysis we have

the specification of the symbolic heaps for each transactional method. An empty path map and empty read- and write-sets are associated to each initial symbolic heap, thus creating a set of initial symbolic states for each transactional method. The complete information for each method is composed by:

- the initial symbolic states, which can be given by the programmer or be computed by the analysis;
- the final symbolic states resulting from the method's execution. These final symbolic states are computed by the analysis and, in the special case of the transactional methods, are the final result of the analysis.

For each method, given one initial symbolic state, the analysis may produce more than one symbolic states. The symbolic execution is defined by function exec that yields a set of symbolic states or an error ($\top$), given a method body (from Stmt) and an initial symbolic state (from $\mathsf{SStates}$):

$$\mathsf{exec} : \mathsf{Stmt} \times \mathsf{SStates} \to \mathcal{P}(\mathsf{SStates}) \cup \{\top\}$$

To support inter-procedural analysis we also need the auxiliary function spec, that given a method signature ($\mathit{fun}(\vec{x}) \in \mathsf{Sig}$), yields a mapping from symbolic heaps to sets of symbolic states: $\mathsf{SHeaps} \to \mathcal{P}(\mathsf{SStates})$.

$$\mathsf{spec} : \mathsf{Sig} \to (\mathsf{SHeaps} \to \mathcal{P}(\mathsf{SStates}))$$

For non-transactional methods, called inside transactions, the initial symbolic state is computed in the course of the symbolic execution, which is inferred from the symbolic state of the calling context. Recursive functions are currently not supported by our analysis technique.

5.1 Past Symbolic Heap

In our analysis we need a special kind of predicates, which we call *past predicates*, and are denoted as $\widehat{p}(\vec{e})$ or $x \widehat{\mapsto} [\rho]$. The past symbolic heap is composed by predicates and past predicates. The latter ones have an important role in the correctness for computing *heap paths*. *Heap paths* must always be computed with respect to the initial snapshot of memory, which is shared between transactions, and corresponds to the initial symbolic heap. Otherwise we may fail to detect some shared memory access due to some memory privatization pattern. We illustrate this problem by means of an example:

Example 2. Given an initial symbolic heap, where $x \in \mathsf{SVars}$ is a shared variable:

$$\{\} | \mathit{List}(x, y) * y \mapsto [\mathit{next} : z] * z \mapsto \mathsf{nil}$$

The *heap paths* representing the locations pointed by each variable are:

$$x \equiv x \qquad y \equiv x.(\mathit{next})^{+}_{A} \qquad z \equiv x.(\mathit{next})^{+}_{A}.\,\mathit{next}$$

If we update the location pointed by y by assigning its *next* field to nil we get

$$\{\}|\, List(x, y) * y \mapsto [next : \mathsf{nil}] * z \mapsto \mathsf{nil}$$

After the update, the *heap paths* representing the locations pointed by x and y remain the same. However, z is no longer reachable from a shared variable, and hence, we have lost the information that in the context of a transaction, z is still a shared memory location subject to concurrent modifications.

This example shows that the *heap path* representing a memory location, that is reachable by a shared variable in the beginning of the transaction, must not be changed by the updates in the structure of the heap. So, in order to compute the correct *heap path* we need to use a "past view" of the current symbolic heap. To get the past view we need *past* predicates, which are added to the symbolic heap whenever an update is made to the structure of the heap. In the case of the previous example, the result of updating variable y would give the following symbolic heap:

$$\{\}|\, List(x, y) * y \mapsto [next : \mathsf{nil}] * y \widehat{\mapsto} [next : z] * z \mapsto \mathsf{nil}$$

The *past* predicate $y \widehat{\mapsto} [next : z]$ denotes that there was a *link* between variable y and z in the initial symbolic heap. Now, if there is a read access to a field of the memory location pointed by variable z, we compute the *heap path* of this location in the past view of the symbolic heap. We define a function that given a symbolic heap returns the past view of such symbolic heap:

Definition 2 (Past Symbolic Heap). *Let* $\mathsf{Past}(H)$ *be the set of past predicates in* H*, and* $\mathsf{NPast}(\Pi|\Sigma) = \{S \mid \Sigma = S * \Sigma' \wedge \neg\, \mathsf{hasPast}_{\Pi|\Sigma}(S)\}$*. Then we define the past symbolic heap by*

$$\mathsf{PSH}(\Pi|\Sigma) \triangleq \Pi|\, \circledast_{S \in \mathsf{NPast}(\Pi|\Sigma)}\, S * \circledast_{\widehat{S} \in \mathsf{Past}(\Pi|\Sigma)} \widehat{S}$$

This function makes use of the $\mathsf{hasPast}$ function to assert if there is already a *past predicate*, in the symbolic heap, with the same *entry* parameters. We define $\mathsf{hasPast}$ as:

Definition 3 (Has Past).

$$\begin{array}{l} \mathsf{hasPast}_{\mathcal{H}}(x \mapsto [\rho]) \Leftrightarrow \mathcal{H} \vdash x \widehat{\mapsto} [\rho] * true \\ \mathsf{hasPast}_{\mathcal{H}}(p(\vec{i}, \vec{o})) \quad \Leftrightarrow \forall i \in \vec{i} : \delta_p^+(i) \wedge \exists i \in \vec{i} : \mathcal{H} \vdash \widehat{p}(\ldots, i, \ldots) * true \end{array}$$

The result of the past heap function applied to the previous example is:

$$\begin{array}{c} \mathsf{PSH}(\{\}|\, List(x, y) * y \mapsto [next : \mathsf{nil}] * y \widehat{\mapsto} [next : z] * z \mapsto \mathsf{nil}) \\ \triangleq \{\}|\, List(x, y) * y \mapsto [next : z] * z \mapsto \mathsf{nil} \end{array}$$

Which corresponds to the initial symbolic heap of Example 2. Thus we can calculate correctly the *heap paths* of the locations pointed by x, y and z.

We also define a function $\mathsf{PastOf}_{\mathcal{H}}(x \mapsto [\rho])$ that if the symbolic heap $\mathcal{H}$ does not contain a past points-to predicate for a points-to predicate $x \mapsto [\rho]$, it creates a new past predicate $x \widehat{\mapsto} [\rho]$.

$$\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, S\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R}', \mathcal{W}'\rangle \quad \vee \quad \langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, S\rangle \Longrightarrow \top$$
$$I(e) ::= e.f := x \mid x := e.f$$

$$\frac{\mathcal{H} \vdash y = \mathsf{nil}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, I(y)\rangle \Longrightarrow \top}(\textsc{Heap Error})$$

$$\frac{x' \text{ is fresh}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := e\rangle \Longrightarrow \langle x = e[x'/x] \wedge \mathcal{H}[x'/x], \mathcal{M}[x \mapsto \mathcal{M}(e)], \mathcal{R}, \mathcal{W}\rangle}(\textsc{Assign})$$

$$\frac{\begin{array}{c} p = \mathsf{GenP}(\mathsf{PSH}(\mathcal{H}), \mathcal{M}, y) \qquad \mathcal{M}' = \mathsf{uMap}(\mathcal{M}, \mathcal{H}, y, p)[x \mapsto p.f] \\ \mathcal{H}' = x = z[x'/x] \wedge \mathcal{H}[x'/x] \qquad x' \text{ is fresh} \end{array}}{\langle \mathcal{H} * y \mapsto [f : z], \mathcal{M}, \mathcal{R}, \mathcal{W}, x := y.f\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R} \cup \{p.f\}, \mathcal{W}\rangle}(\textsc{Heap Read})$$

$$\frac{\begin{array}{c} p = \mathsf{GenP}(\mathsf{PSH}(\mathcal{H} * x \mapsto [f : z]), \mathcal{M}, x) \qquad \mathcal{M}' = \mathsf{uMap}(\mathcal{M}, \mathcal{H}, x, p) \\ \mathcal{H}' = \mathcal{H} * x \mapsto [f : e] * \mathsf{PastOf}_{\mathcal{H}}(x \mapsto [f : z]) \end{array}}{\langle \mathcal{H} * x \mapsto [f : z], \mathcal{M}, \mathcal{R}, \mathcal{W}, x.f := e\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R}, \mathcal{W} \Cup \{p.f\}\rangle}(\textsc{Heap Write})$$

$$\frac{x' \text{ is fresh}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := \mathsf{new}\rangle \Longrightarrow \langle \mathcal{H}[x'/x] * x \mapsto [], \mathcal{M}[x \mapsto \epsilon], \mathcal{R}, \mathcal{W}\rangle}(\textsc{Allocation})$$

$$\frac{}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, \mathsf{return}\, e\rangle \Longrightarrow \langle ret = e \wedge \mathcal{H}, \mathcal{M}[ret \mapsto \mathcal{M}(e)], \mathcal{R}, \mathcal{W}\rangle}\ (\textsc{Return})$$

$$\frac{\begin{array}{c} \langle \mathcal{H}'', \mathcal{M}', \mathcal{R}', \mathcal{W}'\rangle \in \mathsf{spec}(fun(\vec{z}))(\mathcal{H}') \quad \mathcal{H} \vdash \mathcal{H}'[\vec{y}/\vec{z}] * Q \quad \mathcal{H}''' = Q * \mathcal{H}''[\vec{y}/\vec{z}] \\ \mathcal{R}'' = \mathcal{R}'[\vec{y}/\vec{z}] \qquad \mathcal{W}'' = \mathcal{W}'[\vec{y}/\vec{z}] \qquad \mathcal{M}'' = \mathsf{uAMap}(\mathcal{R}'' \cup \mathcal{W}'', \mathcal{M}, \mathcal{H}''') \\ r.P' = \mathcal{M}'(ret) \qquad \mathcal{M}''' = \mathcal{M}''[x \mapsto \mathsf{GenP}(\mathsf{PSH}(\mathcal{H}'''), \mathcal{M}'', r).P'] \\ \mathcal{R}''' = \mathcal{R} \cup \{\mathcal{M}'''(v).P \mid v.P \in \mathcal{R}''\} \qquad \mathcal{W}''' = \mathcal{W} \Cup \{\mathcal{M}'''(v).P \mid v.P \in \mathcal{W}''\} \end{array}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := fun(\vec{y})\rangle \Longrightarrow \langle x = ret \wedge \mathcal{H}''', \mathcal{M}''', \mathcal{R}''', \mathcal{W}'''\rangle}(\textsc{FCall})$$

$$\mathsf{alias}_{\mathcal{H}}(x) \triangleq \{y \mid \mathcal{H} \vdash x = y\} \cup \{x\}$$
$$\mathsf{uMap}(\mathcal{M}, \mathcal{H}, x, p) \triangleq \{v \mapsto s \mid v \mapsto s \in \mathcal{M} \wedge v \notin \mathsf{alias}_{\mathcal{H}}(x)\} \cup \{a \mapsto p \mid a \in \mathsf{alias}_{\mathcal{H}}(x)\}$$
$$\mathsf{uAMap}(V, \mathcal{M}, \mathcal{H}) \triangleq \{s \mid v.P \in V \wedge p = \mathsf{GenP}(\mathsf{PSH}(\mathcal{H}), \mathcal{M}, v) \wedge s \in \mathsf{uMap}(\mathcal{M}, \mathcal{H}, v, p)\}$$

Fig. 11. Operational Symbolic Execution Rules

Definition 4 (Generate Past Predicate).

$$\mathsf{PastOf}_{\mathcal{H}}(x \mapsto [\rho]) \triangleq \begin{cases} \mathsf{emp} & \textit{if } \mathsf{hasPast}_{\mathcal{H}}(x \mapsto [\rho]) \\ x \widehat{\mapsto} [\rho] & \textit{otherwise} \end{cases}$$

5.2 Symbolic Execution Rules

The symbolic execution is defined by the rules shown in Fig. 11.

The rule Assign, when executed in a state $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}\rangle$ adds the information that in the resulting state, x is equal to e. As in standard Hoare/Floyd style assignment, all the occurrences of x, in $\mathcal{H}$ and e, are replaced by a fresh existential quantified variable x'. We also compute a new path map where we

associate variable x with the *heap path* of expression e. If e is null then we associate variable x with empty ϵ. The read- and write-set are not changed because there are no changes in the heap.

The HEAP READ rule adds an equality, to the resulting state, between x and the content of the field f of the location pointed by y. Every time we access the heap, for reading or writing, we compute a new path map. In this case we generate a *heap path* for variable y using the symbolic heap and the current path map. Note that the *heap path* generated is computed in the past symbolic heap as described in Section 5.1. This operation, denoted as GenP, is also responsible for abstracting the representation of *heap paths*, we will describe it in detail in Section 5.4. Given the new computed *heap path* p we compute a new path map by associating path p with variable y, and all its aliases. We use function uMap to perform these operations. Then we associate variable x with the result of the concatenation of path p, which represents the memory location pointed by y, with field f. Finally, we add to the read-set the memory access represented by the *heap path* p and the field f.

The HEAP WRITE rule denotes an update to the value of field f in the location pointed by x. Variable x is associated with the generated *heap path* p ($\mathsf{uMap}(\mathcal{M}, \mathcal{H}, x, p)$) in a new path map. The symbolic heap is extended with a past predicate representing the link between variable x and the record $[f : z]$ that just ceased to exist. The resulting write-set is extended with the field access $\{p.f\}$ ($\mathcal{W} \Cup \{p.f\}$). The operation $\mathcal{W} \Cup \{p.f\}$, denotes the adding of $\{p.f\}$ to both components of the write set $\mathcal{W}$, to the *may* write-set $\mathcal{W}^{>}$ and to the *must* write-set $\mathcal{W}^{<}$. While adding an *heap path* access $p.f$ to the *must* write-set $\mathcal{W}^{<}$ is straightforward, adding $p.f$ to the *may* write-set $\mathcal{W}^{>}$ is a bit more involved. If $\mathcal{W}^{>}$ already contains $p.f$, then we replace all repeating sequences in p, by repeating sequences of the kind $\bar{*}$. For instance, in the previous example, if $p.f = x \,.\, next^{*}_{A} \,.\, next$ is already in $\mathcal{W}^{>}$, the may write-set after adding $p.f$ contains $x \,.\, next^{\bar{*}}_{A} \,.\, next$ instead.

When a new memory location is allocated, rule ALLOCATION, and is assigned to variable x we update the path map entry for variable x with *empty* (ϵ).

In the FCALL rule, the function spec is used to get the symbolic state $\langle \mathcal{H}'', \mathcal{M}', \mathcal{R}', \mathcal{W}' \rangle$ which corresponds to one of the final states of the symbolic execution of a function *fun*. The read- and write-set are composed by *heap path* expressions, where each expression $v.P$ represents a memory location where variable v is the root of the path. This variable is a root variable in the context of function *fun* but in the context of the function that is being analyzed where *fun* was invoked, variable v might point to a memory location that is represented by a *heap path* expression $v'.P'$ where $v' \neq v$. This means that a memory location that is represented by the expression $v.P$ in the context of *fun*, is represented by the expression $v'.P'.P$ in the context of the calling site of *fun* where $v'.P'$ is the expression that represent the memory location pointed by v in the context of the calling site. We need to update all *heap path* expressions of all variables that are in the returned read- ($\mathcal{R}'$) and write-set ($\mathcal{W}'$). We use the uAMap function to iterate over all variables and generate a new *heap path* expression and update

the path map accordingly. The return value of function *fun* is assigned to variable x and therefore we update the path map entry for variable x with the *heap path* expression that represents memory location pointed by the special return variable *ret* in the context of the calling site. In the last step, we merge the read- and write-sets using the updated path map $\mathcal{M}'''$ by concatenating the *heap path* $\mathcal{M}'''(v)$ with the remaining path returned from the read- ($\mathcal{R}'$) or write-set ($\mathcal{W}'$). The final symbolic heap $\mathcal{H}'''$ is computed in the typical way for inter-procedural analysis using separation logic that is by combining the frame of the function call (in this case Q)[5], and the postcondition of the spec $\mathcal{H}''$ [9].

Since we are not aiming at verifying execution errors, we silently ignore the symbolic error states ($\top$) produced by Heap Error rule in our analysis.

5.3 Rearrangement Rules

The symbolic execution rules manipulate object's fields. When these are hidden inside abstract predicates both Heap Read and Heap Write rules require the analyzer to expose the fields they are operating on. This is done by the function rearr defined as:

Definition 5 (Rearrangement).

$$\mathsf{rearr}(\mathcal{H}, x.f) \triangleq \{\mathcal{H}' * x \mapsto [f : y] \mid \mathcal{H} \vdash \mathcal{H}' * x \mapsto [f : y]\}$$

5.4 Fixed Point Computation and Abstraction

Following the spirit of abstract interpretation [6] and the jStar work [9] to ensure termination of symbolic execution, and to automatically compute loop invariants, we apply abstraction on sets of symbolic states. Typically, in separation logic based program analyses, abstraction is done by rewriting rules, also called abstraction rules which implement the function $\mathsf{abs} : \mathsf{SHeaps} \rightarrow \mathsf{SHeaps}$. For each analyzed statement we apply abstraction after applying the execution rules. The abstraction rules accepted by StarTM have the form:

$$\frac{premises}{\mathcal{H} \vdash \mathsf{emp} \rightsquigarrow \mathcal{H}' \vdash \mathsf{emp}} (\textsc{Abstraction Rule})$$

This rewrite is sound if the symbolic heap $\mathcal{H}$ implies the symbolic heap $\mathcal{H}'$. An example of some abstraction rules, for the $List(x, y)$ predicate, is shown in Fig. 2.

The *heap path* expressions that are stored in the path map ($\mathcal{M}$) need also to be abstracted because otherwise we would get expressions with infinite sequences of fields. Since the symbolic heap is abstracted we can use it to compute an abstract *heap path* expression. The abstraction procedure is done by the $\mathsf{GenP}(\mathcal{H}, \mathcal{M}, v)$ function. This function receives a symbolic heap $\mathcal{H}$, a path map

[5] The frame of a call is the part of the calling heap which is not related with the precondition of the callee.

$$
\begin{array}{lll}
\mathsf{compress}(f_1.f_2) = (f_1)^+_A & \text{if } f_1 = f_2 & \text{where } A \text{ is fresh} \\
\mathsf{compress}(f_1.f_2) = (f_1|f_2)^+_A & \text{if } f_1 \neq f_2 & \text{where } A \text{ is fresh} \\
\mathsf{compress}((\mathcal{C})^+_C.f_1) = (\mathcal{C})^+_C & \text{if } f_1 \in \mathcal{C} & \\
\mathsf{compress}((\mathcal{C})^+_C.f_1) = (\mathcal{C}|f_1)^+_C & \text{if } f_1 \notin \mathcal{C} & \\
\mathsf{compress}(f_1.f_2.P) = \mathsf{compress}(\mathsf{compress}(f_1.f_2).P) & &
\end{array}
$$

Fig. 12. Compress abstraction function

$\mathcal{M}$, and a variable v for which will be computed the *heap path* representing the memory location pointed by such variable.

The *heap path* stored in the path map $\mathcal{M}$ for variable v will be denoted as S, and the *heap path* computed from the symbolic heap will be denoted as G. The analysis will always ensure the invariant that $S \subseteq G$. This subset relation means that all paths described by S are also described by G.

The result of this function is a *heap path*, denoted as E which satisfies the following invariant: $S \subseteq E \subseteq G$. Since the symbolic heap is proven to converge into a fixed point, the *heap path* E will also converge into a fixed point because it is a subset of G.

The procedure to compute the path E is based on a pattern matching approach. Taking G as the most abstract path we generate a pattern from it that must match in S. This pattern is generated by taking G and substituting all its repeating sequences with wildcards. For instance, if $G = x.(\mathit{left} \mid \mathit{right})^+_A.\mathit{right}$ then the pattern would be $Pt = x.\alpha.\mathit{right}$ where α is a wildcard. We also denote α_G as the subpath in G that is associated to the wildcard α, and in this case, $\alpha_G = (\mathit{left} \mid \mathit{right})^+_A$.

We take this pattern and try to apply it to S and check which subpath expression of S matches the wildcard. For instance, if $S = x.\mathit{left}.\mathit{left}.\mathit{right}$, then the wildcard α of pattern $Pt = x.\alpha.\mathit{right}$ will match $\mathit{left}.\mathit{left}$ denoted as α_S. The pattern can only be matched successfully if the wildcard in S (α_S) and the wildcard in G (α_G) satisfy the following invariant: $\alpha_S \subseteq \alpha_G$, which is the case in our example.

Now we apply an abstraction operation over the wildcard to generate a more abstract subpath. We denote this operation as $\mathsf{compress}$ and is defined in Fig. 12. The result of applying the abstraction function to wildcard α_S is $\mathsf{compress}(\alpha_S) = \mathit{left}^+_B$. Notice that the abstracted subpath satisfies the invariant $\alpha_S \subseteq \mathsf{compress}(\alpha_S) \subseteq \alpha_G$. Finally, we substitute the wildcards in the pattern for the computed abstract subpath expressions. In our example we get the final expression $E = x.\,\mathit{left}^+_B.\mathit{right}$ which is a subset of G.

5.5 Write-Skew Detection

The result of the symbolic execution is a set of symbolic states $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W} \rangle$ for each transactional method. In this section, we define the *write-skew* test, which

is based on the abstract read- and write-set $(\mathcal{R}, \mathcal{W})$ and on the satisfiability of the condition of Definition 1 (see example in Fig. 3).

Recall that the interpretation of read-sets contain all prefixes of its heap paths. Hence, to compute the satisfiability of the *write-skew condition* we must compute the set of prefixes of the heap-paths in both read-sets. We define $\mathsf{prefix}(x.P)$ for a heap path expression $x.P$ as follows:

$$\mathsf{prefix}(P.f) \triangleq \{P.f\} \cup \mathsf{prefix}(P) \qquad \mathsf{prefix}(P.C_A^*) \triangleq \{P.C_A^*\} \cup \mathsf{prefix}(P)$$
$$\mathsf{prefix}(x.f) \triangleq \{x.f\} \qquad \mathsf{prefix}(x.C_A^*) \triangleq \{x.C_A^*\}$$

and define it for sets of heap paths $\mathsf{prefix}(\mathcal{R})$ as

$$\mathsf{prefix}(\mathcal{R}) \triangleq \bigcup_{p \in \mathcal{R}} \mathsf{prefix}(p).$$

For instance, the prefixes of the read-set $\mathcal{R} = \{this.head.(next)_A^*.next\}$ are:

$$\mathsf{prefix}(\mathcal{R}) = \{this.head,\ this.head.A,\ this.head.A.next\}$$

For the sake of simplicity, we denote repeating sequences by their unique label. Given the sets, $\mathcal{R}_1^\star = \mathsf{prefix}(\mathcal{R}_1)$, $\mathcal{R}_2^\star = \mathsf{prefix}(\mathcal{R}_2)$, $\mathcal{W}_1^<$, $\mathcal{W}_1^>$, $\mathcal{W}_2^<$, and $\mathcal{W}_2^>$, the *write-skew* condition is the following:

$$\mathcal{R}_1^\star \cap \mathcal{W}_2^> \neq \emptyset \quad \wedge \quad \mathcal{W}_1^> \cap \mathcal{R}_2^\star \neq \emptyset \quad \wedge \quad \mathcal{W}_1^< \cap \mathcal{W}_2^< = \emptyset$$

From this condition we generate a set of (in)equations, on the labels of repeating sequences, necessary to reach satisfiability. For instance, given the sets:

$$\mathcal{R}^\star = \{this.head,\ this.head.A,\ this.head.A.next,\ this.head.A.next.next\}$$
$$\mathcal{W}^> = \{this.head.B.next\}$$

The condition $\mathcal{R}^\star \cap \mathcal{W}^> \neq \emptyset$ is satisfied if there is a possible instantiation of A and B such that:

$$B.next \leq A \quad \vee \quad B = A \quad \vee \quad B = A.next$$

In inequation $B.next \leq A$, the operator $\leq$ denotes prefixing, in this case that $B.next$ is a prefix of A. After generating the (in)equation system on labels (A, B) needed to satisfy the *write-skew* condition, we use an SMT solver to check their satisfiability. The consequence of that result is that a *write-skew* may occur between the two transactions being analyzed. Notice that when comparing read- and write-sets we make the correspondence between concrete paths in the heap through the unique labels of repeating sequences.

5.6 Soundness

Our approach is sound for the detection of the *write-skew* anomaly between pairs of transactions. We argue that, by analyzing the satisfiability test described in

section 5.5, if no *write-skew* anomaly is detected by our algorithm then there is no possible execution of the program that contains a *write-skew*. Our analysis computes an over-approximation of the concrete read- and write-sets (the may write-set), and also an under-approximation of the concrete write-set (the must write-set), for all possible executions of the program.

The question then remains whether an occurrence of a write-skew condition at runtime is captured by our test. To see this, let's assume that $\mathcal{R}_1^c$, $\mathcal{W}_1^c$, $\mathcal{R}_2^c$, $\mathcal{W}_2^c$ are concrete, exact read- and write- sets for transactions T_1 and T_2. Notice that a write-skew condition occurs between T_1 and T_2 if

$$\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset$$

Our analysis computes abstract over-approximations of read-sets ($\mathcal{R}_1$ and $\mathcal{R}_2$), write-sets ($\mathcal{W}_1^>$ and $\mathcal{W}_2^>$), and under-approximation of write-sets ($\mathcal{W}_1^<$ and $\mathcal{W}_2^<$) related to the concrete read- and write-sets as follows:

$$\mathcal{R}_1^c \subseteq \mathcal{R}_1, \quad \mathcal{R}_2^c \subseteq \mathcal{R}_2, \quad \mathcal{W}_1^c \subseteq \mathcal{W}_1^>, \quad \mathcal{W}_2^c \subseteq \mathcal{W}_2^>, \quad \mathcal{W}_1^< \subseteq \mathcal{W}_1^c, \quad \mathcal{W}_2^< \subseteq \mathcal{W}_2^c$$

These set relations allow us to prove that the condition on abstract sets is implied by the condition on concrete sets:

$$\begin{aligned}(\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset) \Rightarrow \\ (\mathcal{R}_1 \cap \mathcal{W}_2^> \neq \emptyset \quad \wedge \quad \mathcal{W}_1^> \cap \mathcal{R}_2 \neq \emptyset \quad \wedge \quad \mathcal{W}_1^< \cap \mathcal{W}_2^< = \emptyset)\end{aligned}$$

Hence we can conclude that if a real write-skew exists in an execution this will be detected by our test, and this, as consequence, makes our method sound. The implication above also shows that our method may present false positives: it may detect a write-skew that will never occur at runtime. This is a classical unavoidable effect of conservative methods based on abstract interpretation.

6 Experimental Results

StarTM is a prototype implementation of our static analysis applied to Java byte code, using the Soot toolkit [20] and the CVC3 SMT solver [1]. We applied StarTM to three STM benchmarks: an ordered linked list, a binary search tree, and the Intruder test program of the STAMP benchmark. In the case of the list we tested two versions: the unsafe version called List and the safe version called List Safe. The List Safe version has an additional update in the remove method as discussed in Section 2.

Table 1 shows the detailed results of our verification for each transactional method of the examples above. The results were obtained in a Intel Dual-Core i5 650 computer, with 4 GB of RAM. We show the time (in seconds) taken by StarTM to verify each example, the number of lines of code, and the number of states produced during the analysis. The last column in the table shows the pairs of transactions that may actually trigger a *write-skew* anomaly.

Table 1. StarTM applied to STM benchmarks.

Bench.	Method	Time	LOC	States	*Write-Skews*
List	add	5	16	2	(add, remove) (remove, remove)
	remove		14	2	
	contains		11	1	
	revert		11	4	
List Safe	add	6	16	2	-
	remove		15	2	
	contains		11	1	
Tree	treeAdd	11	21	3	-
	treeContains		15	2	
Intruder	atomicGetPacket	24	9	2	(atomicProcess, atomicGetComplete)
	atomicProcess		173	7	
	atomicGetComplete		15	2	

The expected results for the two versions of the linked list benchmark were confirmed by our tool. The tool detects the existence of two *write-skew* anomalies, in the unsafe version of the linked list, resulting from the concurrent execution of the add and remove methods. The safe version is proven to be completely safe when executing all transactions under SI.

In the case of the Tree benchmark, the treeAdd method performs a tree traversal and inserts a new leaf node. StarTM proves that the concurrent execution of all transactions of the Tree benchmark is safe.

StarTM detects a *write-skew* anomaly in the Intruder example, which is triggered by the concurrent execution of atomicProcess and atomicGetComplete transactions. This happens when transaction atomicProcess pushes an element into a stack and transaction atomicGetComplete pops an element from the same stack, which result on writes on different parts of the memory. However, the Intruder example is not entirely analyzed, there is a small part of the code that is not analyzed due to the use of arrays and cyclic data-structures, which are not currently supported by our tool.

These promising results together with the known performance advantages [7] support the key idea of using relaxed isolation levels in transactional memory systems.

7 Related Work

Software Transactional Memory (STM) [18,11] systems commonly implement the full serializability of memory transactions to ensure the correct execution of concurrent programs. To the best of our knowledge, SI-STM [17] is the only existing implementation of a STM using snapshot isolation. This work focuses on improving the transactional processing throughput by using a snapshot isolation algorithm. It proposes a SI safe variant of the algorithm, where anomalies are

dynamically avoided by enforcing additional validation of read-write conflicts. Our approach avoids this validation by using static analysis and correcting the anomalies before executing the program.

In our work, we aim at providing the serializability semantics under snapshot isolation for STM and Distributed STM systems. This is achieved by performing a static analysis of the program and asserting that no SI anomalies will ever occur when executing a transactional application. This allows to avoid tracking read accesses in both read-only and read-write transactions, thus increasing performance throughput.

The use of snapshot isolation in databases is a common place, and there are some previous works on the detection of SI anomalies in this domain. Fekete et al. [10] developed the theory of SI anomalies detection and proposed a syntactic analysis to detect SI anomalies for the database setting. They assume applications are described in some form of pseudo-code, without conditional (*if-then-else*) and cyclic structures. The proposed analysis is informally described and applied to the database benchmark TPC-C [19] proving that its execution is safe under SI. A sequel of that work [12], describes a prototype which is able to automatically analyze database applications. Their syntactic analysis is based on the names of the columns accessed in the SQL statements that occur within the transaction.

Although targeting similar results, our work deals with different problems. The most significant one is related to the full power of general purpose languages and the use of dynamically allocated heap data structures. To tackle this problem, we use separation logic [16,8] to model operations that manipulate heap pointers. Separation logic has been the subject of research in the last few years for its use in static analysis of dynamic allocation and manipulation of memory, allowing one to reason locally about a portion of the heap. It has been proven to scale for larger programs, such as the Linux kernel [4].

The approach described in [15] has a close connection to ours. It defines an analysis to detect memory independences between statements in a program, which can be used for parallelization. They extended separation logic formulae with labels, which are used to keep track of memory regions through an execution. They can prove that two distinct program fragments use disjoint memory regions on all executions, and hence, these program fragments can be safely parallelized. In our work, we need a finer grain model of the accessed memory regions. We also need to distinguish between read and write accesses to shared and separated memory regions.

The work in [14] informally describes a similar static analysis to approximate read- and write-sets using escape graphs to model the heap structure. Our shape analysis is based on separation logic, and, as far as we understand, heap-paths give a more fine-grain representation of memory locations at a possible expense in scalability.

Some aspects of our work are inspired by jStar [9]. jStar is an automatic verification tool for Java programs, based on separation logic, that enables the automatic verification of entire implementations of several design patterns. Al-

though our work has some aspect in common with jStar, the properties being verified are completely different.

8 Concluding Remarks

We describe a novel and sound approach to automatically verify the absence of the *write-skew* snapshot isolation anomaly in transactional memory programs. Our approach is based on a general model for fine grain abstract representation of accesses to dynamically allocated memory locations. By using this representation, we accurately approximate the concrete read- and write-sets of memory transactions, and capture *write-skew* anomalies as a consequence of the satisfiability of an assertion based on the output of the analysis, the abstract read- and write-sets.

We present StarTM, a prototype implementation of our theoretical framework, unveiling the potential for the safe optimization of transactional memory Java programs by relaxing isolation between transactions. Our approach is not without limitations. Issues that require further developments range from the generalization of the *write-skew* condition for more than two transactions, the support for richer dynamic data structures, to the support for array data types. Together with a runtime system support for mixed isolation levels, we believe that our approach can scale up to significantly optimize real-world transactional memory systems.

Acknowledgments We are grateful to the anonymous reviewers for several insightful comments that significantly improved the paper.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). LNCS, vol. 4590, pp. 298–302. Springer-Verlag (2007)
2. Berenson, H., Bernstein, P., Gray, J.N., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: SIGMOD '95: Proc. of the 1995 ACM SIGMOD international conference on Management of data. pp. 1–10. ACM, New York, NY, USA (1995)
3. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Proc. of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 101–112. POPL '08, ACM, New York, NY, USA (2008)
4. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proc. of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 289–300. POPL '09, ACM, New York, NY, USA (2009)
5. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC '08: Proc. IEEE Int. Symp. on Workload Characterization (2008)

6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977)
7. Dias, R.J., Lourenço, J.M., Preguiça, N.M.: Efficient and correct transactional memory programs combining snapshot isolation and static analysis. In: 3rd USENIX conference on Hot topics in parallelism (HotPar'11). HotPar'11, Usenix Association (2011)
8. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 3920, pp. 287–302. Springer Berlin / Heidelberg (2006)
9. Distefano, D., Parkinson, M.J.: jstar: towards practical verification for Java. In: Proc. of the 23rd ACM SIGPLAN conference on Object-oriented Programming Systems Languages and Applications (OOPSLA'08). pp. 213–226. ACM, New York, NY, USA (2008)
10. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. 30(2), 492–528 (2005)
11. Herlihy, M., Luchangco, V., Moir, M., William N. Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC '03: Proc. of the twenty-second annual symposium on Principles of distributed computing. pp. 92–101. ACM, New York, NY, USA (2003)
12. Jorwekar, S., Fekete, A., Ramamritham, K., Sudarshan, S.: Automating the detection of snapshot isolation anomalies. In: VLDB '07: Proc. of the 33rd international conference on Very large data bases. pp. 1263–1274. VLDB Endowment, Vienna, Austria (2007)
13. Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with Java STM. In: MultiProg 2010: Programmability Issues for Heterogeneous Multicores (2010)
14. Prabhu, P., Ramalingam, G., Vaswani, K.: Safe programmable speculative parallelism. In: Proc. of the 2010 ACM SIGPLAN conf. on Prog. language design and implementation. pp. 50–61. PLDI '10, ACM, New York, NY, USA (2010)
15. Raza, M., Calcagno, C., Gardner, P.: Automatic parallelization with separation logic. In: Proc. of the 18th European Symposium on Programming Languages and Systems (ESOP'09): Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. pp. 348–362. Springer-Verlag, Berlin, Heidelberg (2009)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS '02, IEEE Computer Society, Washington, DC, USA (2002)
17. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: TRANSACT'06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing. Ottawa, Canada (2006)
18. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95: Proc. of the 14th annual ACM symposium on Principles of distributed computing. pp. 204–213. ACM, New York, NY, USA (1995)
19. Transaction Processing Performance Council: TPC-C benchmark, revision 5.11 (2010)
20. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proc. of the 1999 conference of the Centre for Advanced Studies on Collaborative research. pp. 13–. CASCON '99, IBM Press (1999)