Star™: Automatic Verification of Snapshot Isolation in Transactional Memory Java Programs

Ricardo J. Dias[1] Dino Distefano[2,3] João M. Lourenço[1] João Costa Seco[1] *

[1] CITI, Universidade Nova de Lisboa, Portugal
[2] Queen Mary University of London, UK
[3]Monoidics Ltd, UK

{rjfd, joao.lourenco, joao.seco}@di.fct.unl.pt ddino@eecs.qmul.ac.uk

Abstract

This paper presents Star™, an automatic verification tool for transactional memory Java programs executing under relaxed isolation levels. 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 tool builds on a novel shape analysis technique based on Separation Logic to statically approximate the read- and write-sets of a transactional memory Java program. This technique is particularly challenging due to the presence of dynamically allocated memory.

We implement our technique and apply our tool to a set of intricate examples. 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 we identify transactions that potentially trigger the *write-skew* anomaly.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Verification, Reliability

* This work was partially supported by Fundação para a Ciência e Tecnologia (FCT/MCTES).

Keywords Snapshot Isolation, Serializability Anomalies, Separation Logic, Transactional Memory, Static Analysis

1. Introduction

Full-fledged Software Transactional Memory (STM) [11, 17] usually provide 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, may in some cases perform considerably better. The interference among non-serializable transactions are commonly known as *serializability anomalies* [3]. Snapshot Isolation (SI) [3], a relaxed isolation level largely used in database transactional systems to improve performance, is an appealing alternative for the STM setting if used under controlled circumstances. For example, the serializability anomaly allowed by SI, called the *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 Star™, a verification tool for STM Java programs that statically detects if any two transactions may cause a *write-skew* anomaly. Such analysis may be used to optimize code by letting memory transactions run in snapshot isolation whenever possible, and by explicitly requiring the full serializability semantics otherwise.

Our tool performs shape analysis [9] based on Separation Logic [15] to compute memory locations in the read- and write-sets for each distinguished transaction in a Java program. Our analysis starts by producing ranges of memory locations that approximate the read- and write-sets of a transaction, based on a notion of distance to shared variables. For instance, a range of the form `a.next[1,N : N > M]`

describes the access to field `next` on all locations reachable from variable `a`, in a number of steps between 1 and a number `N` that is greater than another number `M`. These ranges are then used to approximate the intersection of read- and write-sets for the numbers `N` and `M`. For this, we resort to an SMT solver with arithmetic theory.

We ran our tool against implementations of a Linked List and a Binary Search Tree, and also against a Java implementation of the STAMP [6] Intruder benchmark. 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 [16]; ii) it is possible to build a transactional insert method in a Binary Search Tree that is safe to execute under SI; and iii) the verification of the STAMP Intruder benchmark proved to trigger the *write-skew* anomaly.

The main contributions of this paper are:

- The first automatic verification technique to statically detect the *write-skew* anomaly in transactional memory programs;
- The first approach able to verify transactional memory programs even in presence of deep-heap manipulation thanks to the use of shape analysis techniques;
- We develop a model that captures fine-grained manipulation of memory locations based on a range of distances to shared variables;
- We have implemented our technique and we have applied our tool to a set of intricate examples. The experimental results are very promising.

The remainder of the paper describes the development, implementation and validation process of our tool. We start by describing a step-by-step example of applying Star™ 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. Star™ by Example

Star™ analyzes Java multithreaded programs that make use of memory transactions. Transactions are defined by 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. 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 by 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 an abstraction function which, in case of infinite state spaces, allows the convergence of the analysis algorithm. The abstraction function is defined by a set of abstraction rules as in the jStar tool [8]. The user defined predicates and abstraction rules are defined 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.

We use as running example the implementation of an ordered singly linked list, adapted from the DeuceSTM [13] samples, shown in Figure 1. The corresponding predicates and abstractions rules are defined in Figure 2. Predicate

```
1  @Predicates(file="list_pred.sl")
2  @Abstractions(file="list_abs.sl")
3  public class List {
4
5    public class Node {
6      int value;
7      Node next;
8      Node(int v, Node n) {value = v; next = n;}
9      int getValue() {return value;}
10     Node getNext() {return next;}
11     void setNext(Node n) {next = n;}
12   }
13
14   private Node head;
15
16   public List() {
17     Node min = new Node(Integer.MIN_VALUE);
18     Node max = new Node(Integer.MAX_VALUE);
19     min.next = max;
20     head = min;
21   }
22
23   @Atomic(state=
24     "| this -> [head:h'] * List(h', nil)")
25   public void add(int value) {
26     boolean result;
27     Node previous = head;
28     Node next = previous.getNext();
29     while (next.getValue() < value) {
30       previous = next;
31       next = previous.getNext();
32     }
33     if (next.getValue() != value) {
34       Node n = new Node(value, next);
35       previous.setNext(n);
36     }
37   }
38
39   @Atomic(state=
40     "| this -> [head:h'] * List(h', nil)")
41   public void remove(int value) {
42     boolean result;
43     Node previous = head;
44     Node next = previous.getNext();
45     while (next.getValue() < value) {
46       previous = next;
47       next = previous.getNext();
48     }
49     if (next.getValue() != value) {
50       previous.setNext(next.getNext());
51       next.setNext(null);
52     }
53   }
54 }
```

Figure 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) );;

/*** Lengths ***/
dist Node(x,n) : n -> x = 1 ;;
dist List(x,y) : y -> x = N : N > 0 ;;


// list_abs.sl file

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

Figure 2. Predicates and Abstraction rules of Linked List

```
# Method boolean add(int value)
Result 1:
ReadSet: { this.head[0, 0],
           this.next[1, 1+N : N > 0],
           this.value[2, 1+N+1 : N > 0] }
WriteSet: { }

Result 2:
ReadSet: { this.head[0, 0],
           this.next[1, 1+N : N > 0],
           this.value[2, 1+N+1 : N > 0] }
WriteSet: { this.next[1+N, 1+N : N > 0] }


# Method boolean remove(int value)
Result 1:
ReadSet: { this.head[0, 0],
           this.next[1, 1+N : N > 0],
           this.value[2, 1+N+1 : N > 0] }
WriteSet: { }

Result 2:
ReadSet: { this.head[0, 0],
           this.next[1, 1+N+1 : N > 0],
           this.value[2, 1+N+1 : N > 0] }
WriteSet: { this.next[1+N, 1+N+1 : N > 0] }
```

Figure 3. Sample of Star™ result output for the Linked List example.

`Node(+x,-y)` defined in Figure 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 different memory locations and there is a chain of nodes leading from the memory location pointed by x and the memory location pointed by y. It can also be the case where both y and the last node in the chain point to nil.

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 4.2.1. In Figure 1, we annotate the add(**int**) and remove(**int**) methods as transactions with the initial state:

```
| 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[1] h', which is the entry point of a list with at least one element.

Star™ performs an intra-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. On every step of the symbolic execution a new abstract state is computed. The read- and write-sets are defined as sets of memory location intervals (or ranges). A memory location is represented by its distance to some shared variable, where the distance corresponds to the number of dereferences necessary to reach that memory location from the shared variable.

The sample of the results of our analysis, depicted in Figure 3, include two possible pairs of read- and write-sets for method add(**int**). The first result has an empty write-set, and thus corresponds to a read-only execution of the method add(**int**), where the ranges in the read-set can be interpreted as follows. The range `this.head[0, 0]` asserts that method add(**int**) reads the field head from the memory location pointed by shared variable **this**, as the distance in this range is 0. The range `this.next[1, 1+N : N > 0]` asserts that the field next is read on all memory cells at a distance between 1 and $1 + N$, starting from the shared variable **this**, for some number $N \in \mathbb{N}$. Notice that the scope of N is delimited by the pair of read- and write-sets within the same result. Range `this.value[1, 1+N+1 : N > 0]` is similar to the previous range. The second pair of read- and write-sets of method add(**int**) in Figure 3 contain the same read-set and a different write-set. In this case, range

[1] Throughout this paper we consider primed variable as tacitly existentially quantified.

`this.next[1+N, 1+N : N > 0]` asserts that the method add(**int**) updates the field next of the memory location which is at a distance of $1 + N$ from variable **this**.

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**). Note that N is distinct for each pair of read- and write-sets. In the other result for method remove(**int**), field next is read for all memory locations whose distance to shared variable **this** is between 1 and $1 + N + 1$. In the case of the write-set, field next is updated for memory locations whose distance to shared variable **this** is $1 + N$ and $1 + N + 1$. Notice that the two update accesses identified in the write-set correspond to the lines 50 and 51 in Figure 1. The update operation in line 51, although unnecessary in terms of the list semantics, is essencial to make the Linked List implementation safe under Snapshot Isolation. The precise justification for this claim is laid out in the remainder of this section.

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

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

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

We will consider that each result (a pair of a read- and a write-set) corresponds to a single transaction instance. In this example we have four possible transactions, and we use T_{add_1} and T_{add_2} to denote the first and second result of method add(**int**); and T_{rem1} and T_{rem2} to denote the first and second possible results of method remove(**int**). To detect a *write-skew* anomaly we compare all possible pairs of transactions including the ones that contain a transaction in concurrency with itself. In this case we need to test the following pairs:
$(\mathrm{T}_{add_1}, \mathrm{T}_{add_1})$, $(\mathrm{T}_{add_1}, \mathrm{T}_{add_2})$, $(\mathrm{T}_{add_2}, \mathrm{T}_{add_2})$,
$(\mathrm{T}_{rem_1}, \mathrm{T}_{rem_1})$, $(\mathrm{T}_{rem_1}, \mathrm{T}_{rem_2})$, $(\mathrm{T}_{rem_2}, \mathrm{T}_{rem_2})$,
$(\mathrm{T}_{add_1}, \mathrm{T}_{rem_1})$, $(\mathrm{T}_{add_1}, \mathrm{T}_{rem_2})$, $(\mathrm{T}_{add_2}, \mathrm{T}_{rem_1})$, and
$(\mathrm{T}_{add_2}, \mathrm{T}_{rem_2})$.
The number of pairs to be analyzed can be significantly reduced because some pairs are trivially safe, meaning that they will not trigger a *write-skew* anomaly. Since transactions T_{add_1} and T_{rem_1} are read-only, we do not need to test pairs that contain any of these transactions. We need only to test the pairs:
$(\mathrm{T}_{add2}, \mathrm{T}_{add2})$, $(\mathrm{T}_{rem2}, \mathrm{T}_{rem2})$ and $(\mathrm{T}_{add2}, \mathrm{T}_{rem2})$
In the cases above, the *write-skew* condition is not satisfiable. As an example we examine in detail the pair $(\mathrm{T}_{add_2}, \mathrm{T}_{rem_2})$. We will only consider the intersection of intervals associated with the next field because these transactions never write on the value field. 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_2}, and T_{rem_2} (relative to field next) are

$$\begin{aligned}
\mathcal{R}_{add_2} &= [1, 1 + N] \text{ where } N \in \mathbb{N} \\
\mathcal{W}_{add_2} &= [1 + N, 1 + N] \text{ were } N \in \mathbb{N} \\
\mathcal{R}_{rem_2} &= [1, 2 + M] \text{ where } M \in \mathbb{N} \\
\mathcal{W}_{rem_2} &= [1 + M, 2 + M] \text{ were } M \in \mathbb{N}
\end{aligned}$$

Given theses read- and write-sets, it is not possible to find two numbers N and M, such that the *write-skew* condition is true, hence these transaction can execute concurrently without ever triggering a *write-skew* anomaly.

If we consider the case of the code in Figure 1, without the update in line 51 (irrelevant to the semantics of the remove operation). The write-set resulting from the analysis is `this.next[1+N, 1+N : N > 0]`. The read-set is the same as before. Hence, the read- and write-sets to test satisfiability are:

$$\begin{aligned}
\mathcal{R}_{add_2} &= [1, 1 + N] \text{ where } N \in \mathbb{N} \\
\mathcal{W}_{add_2} &= [1 + N, 1 + N] \text{ were } N \in \mathbb{N} \\
\mathcal{R}_{rem_2} &= [1, 2 + M] \text{ where } M \in \mathbb{N} \\
\mathcal{W}_{rem_2} &= [1 + M, 1 + M] \text{ were } M \in \mathbb{N}
\end{aligned}$$

In this case, if we consider $N = 2$ and $M = 1$, the *write-skew* condition holds, and hence the concurrent execution of the two transactions could possibly cause a *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 essencial features such as object creation (new), field dereferencing ($x.f$), assignment ($x := e$), and function and procedure invocation ($func(\vec{x})$ and $proc(\vec{x})$). The syntax of the language is defined by the grammar in Figure 4. A program in this language is a set of procedure and function definitions. We do not explicitly represent transactions or an explicit entry point in the syntax and we assume that all procedures are transactions that can be called concurrently.

The operational semantics for the language is defined over configurations of the form $\langle s, h, S \rangle$, where $s \in \mathsf{Stacks}$ is a stack (a mapping from variables and primed variables to values), $h \in \mathsf{Heaps}$ is a (concrete) heap (a mapping from locations to values through field labels). We assume given a countable set of program variables Vars (ranged over by $x, y, \ldots$).

$$\begin{aligned}
\mathsf{Stacks} &= \mathsf{Vars} \to \mathsf{Values} \\
\mathsf{Heaps} &= \mathsf{Locations} \rightharpoonup \mathsf{Fields} \to \mathsf{Values} \\
\mathsf{Values} &= \mathbb{Z} \cup \mathsf{Locations} \cup \{\mathsf{nil}\}
\end{aligned}$$

We define a semantic function $\mathcal{A} : \mathsf{Exp} \to \mathsf{Stacks} \to \mathsf{Values}$ to evaluate expressions, where $\oplus_a$ represents the

$$
\begin{array}{lcll}
e & ::= & & \textit{(expression)} \\
 & & x & \text{(variables)} \\
 & | & n & \text{(constant)} \\
 & | & e \oplus_a e & \text{(arithmetic op)} \\
 & | & \mathsf{null} & \text{(null value)} \\
b & ::= & & \textit{(boolean exp)} \\
 & & e \oplus_b e & \text{(boolean op)} \\
 & | & \mathsf{true} & \text{(true value)} \\
 & | & \mathsf{false} & \text{(false value)}
\end{array}
\qquad
\begin{array}{lcll}
A & ::= & & \textit{(assignments)} \\
 & & x := e & \text{(local)} \\
 & | & x := y.f & \text{(heap read)} \\
 & | & x := \mathit{func}(\vec{y}) & \text{(function call)} \\
 & | & x.f := e & \text{(heap write)} \\
 & | & x := \mathsf{new} & \text{(allocation)}
\end{array}
$$

$$
\begin{array}{lcll}
S & ::= & & \textit{(statements)} \\
 & & S\,;S & \text{(sequence)} \\
 & | & A & \text{(assignment)} \\
 & | & \mathit{proc}(\vec{y}) & \text{(procedure call)} \\
 & | & \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S & \text{(conditional)} \\
 & | & \mathsf{while}\ b\ \mathsf{do}\ S & \text{(loop)} \\
 & | & \mathsf{return}\ e & \text{(return)} \\
 & | & \mathsf{skip} & \text{(Skip)} \\
 & | & \mathsf{error} & \text{(Error)}
\end{array}
$$

$$
P \quad ::= \quad \mathit{func}(\vec{x}) = S \ \mid\ \mathit{proc}(\vec{x}) = S \ \mid\ P \qquad \textit{(program)}
$$

Figure 4. Core language syntax

arithmetic binary operations $+, -, \times, \ldots$

$$
\mathcal{A}[\![e]\!]_s = \begin{cases} n, & \text{if } e = n \\ s(x), & \text{if } e = x \\ \mathsf{nil}, & \text{if } e = \mathsf{null} \\ \mathcal{A}[\![e_1]\!]_s \oplus_a \mathcal{A}[\![e_2]\!]_s, & \text{if } e = e_1 \oplus_a e_2 \end{cases}
$$

Likewise, boolean expressions are evaluated according to the semantic function $\mathcal{B} : \mathsf{BExp} \rightarrow \{\mathit{true}, \mathit{false}\}$, where $\oplus_b$ represents the boolean binary operations $=, \neq, <, \leq, \ldots$

$$
\mathcal{B}[\![b]\!]_s = \begin{cases} \mathit{true}, & \text{if } b = \mathsf{true} \\ \mathit{false}, & \text{if } b = \mathsf{false} \\ \mathcal{A}[\![e_1]\!]_s \oplus_b \mathcal{A}[\![e_2]\!]_s, & \text{if } b = e_1 \oplus_b e_2 \end{cases}
$$

The small step structural operational semantics of the language is defined by the set of rules in Figure 5.

4. Abstract Domain

The abstract domain used in the symbolic execution is defined as a set of triples of the form $(\mathcal{H}, \mathcal{R}, \mathcal{W})$: where $\mathcal{H}$ is a symbolic heap, defined using a fragment of separation logic formulae, and $\mathcal{R}$ and $\mathcal{W}$ are read- and write-sets.

The fragment of separation logic formulae that we use to describe symbolic heaps is defined by the grammar in Figure 6. The corresponding semantics is given by a satisfaction relation $\models$ between a concrete stack, a concrete heap and a symbolic heap $s, h \models \mathcal{H}$, and sample clauses of the semantics appear in Figure 7.

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 $(x'_1, \ldots, x'_n)$ 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

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

Figure 6. Separation logic syntax

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

Figure 7. Separation Logic semantics

variables (e.g., $x = \mathsf{nil}$), but are not concerned with heap allocated objects. The spatial part is the $*$ conjunction of spatial predicates, i.e., related to heap facts. In separation

$$\boxed{\langle s, h, S\rangle \Longrightarrow \langle s', h', S'\rangle}$$

$$\frac{\langle s, h, S_1\rangle \Longrightarrow \langle s', h', S_1'\rangle}{\langle s, h, S_1\ ; S_2\rangle \Longrightarrow \langle s', h', S_1'\ ; S_2\rangle}(\textsc{Seq 1})$$

$$\frac{\langle s, h, S_1\rangle \Longrightarrow \langle s', h', \mathsf{skip}\rangle}{\langle s, h, S_1\ ; S_2\rangle \Longrightarrow \langle s', h', S_2\rangle}(\textsc{Seq 2})$$

$$\frac{\mathcal{B}[\![e]\!]_s = \mathit{true}}{\langle s, h, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \Longrightarrow \langle s, h, S_1\rangle}(\textsc{Cond 1})$$

$$\frac{\mathcal{B}[\![e]\!]_s = \mathit{false}}{\langle s, h, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \Longrightarrow \langle s, h, S_2\rangle}(\textsc{Cond 2})$$

$$\frac{\mathcal{B}[\![e]\!]_s = \mathit{true}}{\langle s, h, \mathsf{while}\ e\ \mathsf{do}\ S\rangle \Longrightarrow \langle s, h, S\ ; \mathsf{while}\ e\ \mathsf{do}\ S\rangle}(\textsc{Loop 1})$$

$$\frac{\mathcal{B}[\![e]\!]_s = \mathit{false}}{\langle s, h, \mathsf{while}\ e\ \mathsf{do}\ S\rangle \Longrightarrow \langle s, h, \mathsf{skip}\rangle}(\textsc{Loop 2})$$

$$\frac{P(\mathit{proc}) = (\vec{x})S \qquad S' = S\{\vec{y}/\vec{x}\}}{\langle s, h, \mathit{proc}(\vec{y})\rangle \Longrightarrow \langle s, h, S'\rangle}(\textsc{PCall})$$

$$\frac{\mathcal{A}[\![e]\!]_s = v}{\langle s, h, \mathsf{return}\ e\rangle \Longrightarrow \langle s[\mathit{ret} \mapsto v], h, \mathsf{skip}\rangle}(\textsc{Return})$$

$$\frac{\mathcal{A}[\![e]\!]_s = v}{\langle s, h, x := e\rangle \Longrightarrow \langle s[x \mapsto v], h, \mathsf{skip}\rangle}(\textsc{Assign})$$

$$\frac{s(y) = l \qquad l \in \mathsf{dom}(h) \qquad h(l)(f) = v}{\langle s, h, x := y.f\rangle \Longrightarrow \langle s[x \mapsto v], h, \mathsf{skip}\rangle}(\textsc{Heap Read})$$

$$\frac{s(x) = l \qquad l \in \mathsf{dom}(h) \qquad \mathcal{A}[\![e]\!]_s = v}{\langle s, h, x.f := e\rangle \Longrightarrow \langle s, h[l \mapsto f \mapsto v], \mathsf{skip}\rangle}(\textsc{Heap Write})$$

$$\frac{P(\mathit{func}) = (\vec{x})S \qquad S' = S\{\vec{y}/\vec{x}\}}{\langle s, h, x := \mathit{func}(\vec{y})\rangle \Longrightarrow \langle s, h, S'; x := \mathit{ret}\rangle}(\textsc{FCall})$$

$$\frac{l \notin \mathsf{dom}(h) \qquad l \neq \mathsf{nil}}{\langle s, h, x := \mathsf{new}\rangle \Longrightarrow \langle s[x \mapsto l], h[l \mapsto \_]\rangle}(\textsc{Allocation})$$

Figure 5. Structural operation semantics

logic, the formula

$$S_1 * S_2$$

holds in a heap that can be split into two disjoint parts where in one of them the only allocated memory is described by S_1 and in the other only by S_2.

We use a *field splitting model* [8], i.e., in our model, objects are considered to be a compound entities composed by fields which can be split by $*$. Notice that if S_1 and S_2 describe the same field of an object than $S_1 * S_2$ implies false. We call SHeaps to the set of all valid symbolic heaps.

4.2 Memory Representation

The concrete read-/write-sets of a transaction is a set of actual memory locations which are accessed for reading/writing during its execution. In our work, we want to statically determine the read- and write-sets of memory transactions which are impossible to compute at compile time. Hence, we need to find a suitable memory representation based on static information.

We start by only describing memory locations that are accessible (shared) by different threads, which means that they are reachable starting from some shared variable. We define shared memory reachability in a concrete stack and a concrete heap as follows:

Definition 2 (Memory Reachability)**.** *Given a state* $s \in \mathsf{Stacks}$ *and a heap* $h \in \mathsf{Heaps}$*, we define* $\mathsf{ReachS}_{(s,h)}(l)$ *and* $\mathsf{ReachH}_h(l', l)$ *as*

$$\mathsf{ReachS}_{(s,h)}(l) \triangleq \exists\, v \in \mathsf{dom}(s), l' \in \mathsf{dom}(h) : \mathsf{shared}(v) \wedge s(v) = l' \wedge \mathsf{ReachH}_h(l', l)$$

$$\mathsf{ReachH}_h(l', l) \triangleq \exists f \in \mathsf{Fields} : h(l')(f) = l \vee (h(l')(f) = l'' \wedge \mathsf{ReachH}_h(l'', l))$$

For the sake of simplicity consider that $\mathsf{shared}(v)$ is always known for any program. We represent a shared memory location, allocated at runtime, by a pair with a shared variable and a field path. A field path is a sequence of fields that is used to reach the memory location. A memory location directly pointed by a shared variable has an empty field path.

Definition 3 (Field Path)**.** *A field path is a sequence of fields, which is represented as* $\langle f_1, f_2, \ldots, f_n\rangle$*, and is calculated using the following function:*

$$\mathsf{FPath}_h(l, l') \triangleq \begin{cases} \langle\rangle & \text{if } l = l' \\ \langle f\rangle \cup \mathsf{FPath}_h(l'', l') & \text{if } l \neq l' \wedge \exists f \in \mathsf{Fields} \\ & \quad h(l)(f) = l'' \end{cases}$$

For the sake of comprehension we are not considering multiple paths from one shared variable to some shared location, although that may happen in reality. If that is the case we always pick the path with maximum length. A shared location is then identified by a set of pairs, of the form $(v, \langle f_1, \ldots, f_n\rangle)$, for each shared variable v that has a path to such shared location.

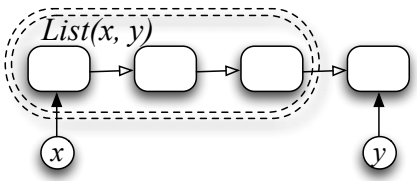

Figure 8. Graph representation of the $List(x, y)$ predicate.

Considering that the heap may be infinite, a field path may also have an infinite length. For program analysis purposes a more convenient representation should be used with the tradeoff of losing some precision. We abstract this representation by instead of keeping a field path, we keep the length of the field path and hence, we represent a memory location by a set of pairs of the form (v, n) where v is the shared variable and n is the length of the field path. While the previous representation uniquely identifies a shared location, the new abstract representation may identify two distinct shared locations at the same time. But with this new representation it is easy to represent shared locations that may have a length described as a restriction in the domain of natural numbers. For instance, (v, n) where $n > 3$ means a shared location that has a length of at least 4 from the shared variable v. We will use this abstract representation to identify shared locations in our static analysis.

4.2.1 Static Memory Accesses

Since our analysis models the concrete states of execution using *symbolic heaps* $\mathcal{H} = \Pi|\Sigma$, we have to define the length of a field path in terms of $\mathcal{H}$. In symbolic heaps, memory locations are pointed directly by program variables (e.g., v) or by existencial prime variables (e.g., v'), or are abstracted inside predicates. Predicates are abstractions of a set of one or more memory locations and abstract their graph structure. For example, as depicted in Figure 8, the $List(x, y)$ predicate abstract a set, of unbound cardinality, of memory locations where each location is linked to another location by its next field.

Every predicate $p(\vec{e})$ has a subset of its parameters which are the entry points for reaching every memory location that it abstracts. We call these parameters *entry* parameters, and every predicate has at least one *entry* parameter. 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 Figure 8 we can observe that the predicate $List(x, y)$ has one *entry* parameter x and one *exit* parameter y. Users of Star™ are required to indicate which parameters of a predicate are *entry* or *exit* by using an unary operator as a prefixed of the parameter. Operators + and − mean *entry* and *exit* respectively. In our formal definitions we can query if a parameter p is of *entry* or *exit* type with the $\delta^+(p)$ or $\delta^-(p)$ operators respectively. For the special case of predicate ($\mapsto$), we consider that the variable on the left side of ($\mapsto$) is an *entry* parameter and the variables at the right side of ($\mapsto$) are *exit* parameters.

Besides defining *entry* and *exit* parameters, the user must specify, for each predicate, the abstract length between each pair of *exit* and *entry* parameters of the predicate. For instance, in the $List(x, y)$ predicate, the user must specify the abstract length between variable y (*exit*) and variable x (*entry*). This length is semantically equivalent to the length of the field path presented in the previous section. For the $List(x, y)$ predicate the user may define that the length between y and x is $n > 0$. Meaning that the length of this predicate could be of 1 or more. In the case of the $Node(x, y)$ predicate, where $\delta^+(x)$ and $\delta^-(y)$, the length should be defined as 1. All lengths between pairs of *entry* parameters are defined as 0.

During symbolic execution, the same predicate holding for different parameters in the same symbolic heap, or holding for the same parameters in different symbolic heaps, may have different length expressions for the corresponding parameters. This situation occurs due to the use of abstractions [9], which abstract a set of predicates into another set of predicates. The length expressions of the resulting set of predicates are recalculated and associated to those predicates. These abstraction computations are described in detail in Section 5.4.

We denote $\Delta^{List}(y, x)$ as the function to retrieve the length between variable y and x of the predicate $List$, where y is an *exit* parameter $\delta^-(y)$ and x is an *entry* parameter $\delta^+(x)$. In Star™ the user has to specify, for each predicate and pair of *entry* and *exit* parameters, the corresponding length. We now define a small language $\mathcal{L}$ used to specify such lengths.

L	::=		(*Length*)
		$n \in \mathbb{N}_0$	(constant)
	|	N	(natural variable)
	|	$L + L$	(length sum)
N	::=	$v[C]$	(*Natural Variable Definition*)
C	::=	$(> \mid \geq \mid =)\ L$	(*Condition*)

A length, in this language, can be a natural constant or a variable with a domain condition. The domain condition can be composed by other variables or natural constants. These variables, are not variables in the sense of traditional programming languages, this variables are just identifiers associated with a condition. The only restriction is just we do not allow the use of the same identifier in a variable present in a nested condition. An example of a valid length in this language is: $N[\geq 1 + M[> 0]] + 2$. The mathematical meaning of this expression is the length: $\exists N, M \in \mathbb{N}_0 : 2 + N$ where $M > 0 \ \wedge \ N \geq M + 1$. During symbolic execution, the identifiers of these lengths will be generated according to well defined rules for convergence purposes.

We can define two notions of length equality. The first one is based on identifier equality and is defined as follows:

Definition 4 (Identifier Length Equality)**.**

$$\begin{aligned}
&\mathsf{lequal}_{id}(n_1, n_2) \Leftrightarrow n_1 = n_2\\
&\mathsf{lequal}_{id}(v_1[c_1], v_2[c_2]) \Leftrightarrow v_1 = v_2\\
&\mathsf{lequal}_{id}(l_1 + l_2, l_3 + l_4) \Leftrightarrow (\mathsf{lequal}_{id}(l_1, l_3) \wedge \mathsf{lequal}_{id}(l_2, l_4))\\
&\quad \vee (\mathsf{lequal}_{id}(l_1, l_4) \wedge \mathsf{lequal}_{id}(l_2, l_3))
\end{aligned}$$

The second notion of equality is defined based on the structure of the expression meaning that two expressions l_1 and l_2 of language $\mathcal{L}$ are equal if have the same conditions and constants in the same order.

Definition 5 (Structural Length Equality)**.**

$$\begin{aligned}
&\mathsf{lequal}(n_1, n_2) \Leftrightarrow n_1 = n_2\\
&\mathsf{lequal}(l_1 + l_2, l_3 + l_4) \Leftrightarrow (\mathsf{lequal}(l_1, l_3) \wedge \mathsf{lequal}(l_2, l_4))\\
&\quad \vee (\mathsf{lequal}(l_1, l_4) \wedge \mathsf{lequal}(l_2, l_3))\\
&\mathsf{lequal}(v_1[\alpha_1\, l_1], v_2[\alpha_2\, l_2]) \Leftrightarrow \alpha_1 = \alpha_2 \wedge \mathsf{lequal}(l_1, l_2)\\
&\quad \text{where } \alpha \in [>, \geq, =]
\end{aligned}$$

For our symbolic execution, we will also need to compare inequalities between lengths. For instance, given $l_1 = 1 + N_1[\geq\ N_2[\geq\ 2] + 1]$ and $l_2 = 1 + N_2[\geq\ 2]$ compare if $l_1 \leq l_2$. For this purpose we make use of an SMT solver, and we translate our length expressions in SMT formulae and check the satisfiability of inequalities.

We may now define the function that calculates the length of the field path between two variables in a symbolic heap $\mathcal{H}$, and the function that calculates the length of such path. We first define a function to test the reachability between two variables and then we use it to calculate the respective path.

Definition 6 (Reachability)**.**

$$\begin{aligned}
&\mathsf{reach}_\Pi(x \mapsto [f_1 : x_1, \ldots, f_n : x_n], y, z) \Leftrightarrow\\
&\quad \Pi \vdash y = x \wedge \exists k \in [1, n] : \Pi \vdash z = x_k\\
&\mathsf{reach}_\Pi(p(\vec{i}, \vec{o}), y, z) \Leftrightarrow\\
&\quad \delta^+(\vec{i}) \wedge \delta^-(\vec{o}) \wedge \exists i \in \vec{i}, o \in \vec{o} : \Pi \vdash y = i \wedge \Pi \vdash z = o\\
&\mathsf{reach}_\Pi(x \mapsto [f_1 : x_1, \ldots, f_n : x_n] * \Sigma, y, z) \Leftrightarrow\\
&\quad \Pi \vdash y = x \wedge \exists k \in [1, n] : \mathsf{reach}_\Pi(\Sigma, x_k, z)\\
&\mathsf{reach}_\Pi(p(\vec{i}, \vec{o}) * \Sigma, y, z) \Leftrightarrow\\
&\quad \delta^+(\vec{i}) \wedge \delta^-(\vec{o}) \wedge \exists i \in \vec{i}, o \in \vec{o} : \Pi \vdash y = i \wedge \mathsf{reach}_\Pi(\Sigma, o, z)\\
&\mathsf{reach}_\Pi(\Sigma, y, z) \Leftrightarrow \mathit{false} \text{ if none of the above conditions are met}
\end{aligned}$$

Definition 7 (Predicate Path)**.** *Returns the predicates that constitute the path between two variables.*

$$\begin{aligned}
&\mathsf{path}_\Pi(x \mapsto [f_1 : x_1, \ldots, f_n : x_n], y, z) \triangleq\\
&\qquad \{\langle x \mapsto [f_1 : x_1, \ldots, f_n : x_n]\rangle\}\\
&\quad \text{if } \mathsf{reach}_\Pi(x \mapsto [f_1 : x_1, \ldots, f_n : x_n], y, z)\\
&\mathsf{path}_\Pi(p(\vec{i}, \vec{o}), y, z) \triangleq \{\langle p(\vec{i}, \vec{o})\rangle\} \text{ if } \mathsf{reach}_\Pi(p(\vec{i}, \vec{o}), y, z)\\
&\mathsf{path}_\Pi(x \mapsto [f_1 : x_1, \ldots, f_n : x_n] * \Sigma, y, z) \triangleq\\
&\qquad x \mapsto [f_1 : x_1, \ldots, f_n : x_n] \otimes\\
&\qquad \{\mathsf{path}_\Pi(\Sigma, x_1, z), \ldots, \mathsf{path}_\Pi(\Sigma, x_n, z)\}\\
&\quad \text{if } \mathsf{reach}_\Pi(x \mapsto [f_1 : x_1, \ldots, f_n : x_n] * \Sigma, y, z)\\
&\mathsf{path}_\Pi(p(\vec{i}, \vec{o}) * \Sigma, y, z) \triangleq p(\vec{i}, \vec{o}) \otimes \{\mathsf{path}_\Pi(\Sigma, o, z) | \forall o \in \vec{o}\}\\
&\quad \text{if } \mathsf{reach}_\Pi(p(\vec{i}, \vec{o}) * \Sigma, y, z)\\
&\mathsf{path}_\Pi(\Sigma, y, z) \triangleq \mathsf{emp} \text{ if } \neg\, \mathsf{reach}_\Pi(\Sigma, y, z)
\end{aligned}$$

The function path makes use of the $\otimes$ operator wich joins the left side predicate with each predicate path in the set of the right side.

Example 1.

$$\begin{aligned}
&\mathit{List}(x, y) \otimes \{\langle \mathsf{emp}\rangle, \langle \mathsf{emp}\rangle, \langle \mathit{Node}(y, z)\rangle\}\\
&\quad = \{\langle \mathit{List}(x, y), \mathsf{emp}\rangle, \langle \mathit{List}(x, y),\\
&\qquad \mathsf{emp}\rangle, \langle \mathit{List}(x, y), \mathit{Node}(y, z)\rangle\}\\
&\quad = \{\langle \mathit{List}(x, y)\rangle, \langle \mathit{List}(x, y), \mathit{Node}(y, z)\rangle\}
\end{aligned}$$

The result of the *predicate path* function is a set of paths between two variables and its prefix paths. We can ignore the prefixes because they are just a consequence of the definition and are not relevant. If we get two different paths for the same pair of variables then we can soundly choose the longest one. So, given two variables y and z and a symbolic path between the two, we can calculate the length between y and z using the following function.

Definition 8 (Predicate Path Length)**.** *Returns the length of the predicate path.*

$$\begin{aligned}
&\mathsf{length}_\Pi(\langle x \mapsto [f_1 : x_1, \ldots, f_n : x_n], \Sigma\rangle, y, z) \triangleq 1\\
&\quad + \mathsf{length}_\Pi(\langle \Sigma\rangle, \mathsf{cvar}(x \mapsto [f_1 : x_1, \ldots, f_n : x_n], \Sigma), z)\\
&\mathsf{length}_\Pi(p(\vec{i}, \vec{o}), \Sigma\rangle, y, z) \triangleq \Delta^p(y, v) + \mathsf{length}_\Pi(\langle \Sigma\rangle, v, z)\\
&\quad \text{where } v = \mathsf{cvar}(p(\vec{i}, \vec{o}), \Sigma)\\
&\mathsf{length}_\Pi(\langle x \mapsto [f_1 : x_1, \ldots, f_n : x_n]\rangle, y, z) \triangleq 1\\
&\mathsf{length}_\Pi(\langle p(\vec{i}, \vec{o})\rangle, y, z) \triangleq \Delta^p(y, z)
\end{aligned}$$

This function makes use of the following function cvar, which calculates the variable that is common in two successive predicates in a path:

Definition 9 (Connection Variable)**.**

$$\mathsf{cvar}(p_1(\vec{i_1}, \vec{o_1}), p_2(\vec{i_2}, \vec{o_2})) \triangleq v$$
$$\text{if } \exists o_1 \in \vec{o_1}, i_2 \in \vec{i_2} : \delta^-(o_1) \wedge \delta^+(i_2) \wedge v = o_1 = i_2$$
$$\mathsf{cvar}(p_1(\vec{i_1}, \vec{o_1}), x \mapsto [f_1 : x_1, \ldots, f_n : x_n]) \triangleq x$$
$$\mathsf{cvar}(x \mapsto [f_1 : x_1, \ldots, f_n : x_n], p_2(\vec{i_2}, \vec{o_2})) \triangleq v$$
$$\text{if } \exists k \in [1, n], i_2 \in \vec{i_2} : \delta^-(x_k) \wedge \delta^+(i_2) \wedge v = x_k = i_2$$
$$\mathsf{cvar}(x \mapsto [f_1 : x_1, \ldots, f_n : x_n], y \mapsto [f_1 : y_1, \ldots, f_n : y_n]) \triangleq y$$

An abstract memory location is then represented by a set of pairs of the form (x, len) where x is a shared variable and $len \in \mathcal{L}$ is a symbolic length. An abstract memory access is pair of the form (m, f) where $m = (x, len)$ is an abstract memory location and $f \in \mathsf{Fields}$ is the field accessed in location m. For simplicity we represent an access, from now on, as triple of the form (x, f, len).

4.3 Memory Ranges

A memory range (or interval) is a sequence of memory locations that are reachable from the same shared variable and all the locations were accessed by the same field. If we have a set of memory accesses, of the form (x, f, len), we can group them by variable x and field f and create a range with the minimum and maximum length len. Each memory range has the form $(x, f, [l_1, l_2])$ where x is a shared variable, f is a field and l_1 and l_2 are abstract lengths. The set of all memory ranges is denoted as Ranges. The read- and write-set are defined as a subset of Ranges: $\mathcal{R} \subseteq \mathsf{Ranges}$ and $\mathcal{W} \subseteq \mathsf{Ranges}$.

During symbolic execution read and write accesses must be added to the read- and write-set respectively. But we need to define how a memory access is transformed into a range, or added into an existing range. We first define the operation $\oplus_r$ for inserting a memory access, of the form (x, f, l) where x is a shared variable, f is a field, and l is an abstract length, into an existing range:

Definition 10 (Range Add)**.**

$$(x, f, l) \oplus_r (x', f', [l_1, l_2]) \triangleq (x', f', [l_1, l_2]) \quad \text{if } x \neq x' \vee f \neq f'$$
$$(x, f, l) \oplus_r (x', f', [l_1, l_2]) \triangleq (x', f', [l_1, l_2]) \text{ if } l \subseteq [l_1, l_2]$$
$$(x, f, l) \oplus_r (x', f', [l_1, l_2]) \triangleq (x', f', [l, l_2]) \text{ if } l < l_1$$
$$(x, f, l) \oplus_r (x', f', [l_1, l_2]) \triangleq (x', f', [l_1, l]) \text{ if } l > l_2$$

Adding a memory access to a range only takes effect if the the access variable and field matches the ones in the range, and if the access length is outside the interval defined by the range. Please note that with this operation we are abstracting the memory accesses by keeping an over approximation instead of keeping a collection of concrete memory accesses.

Now we define the operation $\oplus_s$ for inserting a memory access into a set of ranges which constitutes a read- or write-set.

Definition 11 (Range Set Add)**.** *Given* $R \subseteq \mathsf{Ranges}$*:*

$$(x, f, l) \oplus_s R \triangleq \{(x, f, l) \oplus_r r \mid r \in R\}$$

The symbolic execution rules make use of these functions to generate the abstract read- and write-sets of each analyzed method.

5. Symbolic Execution

Next, we define our symbolic execution for the core language presented in Section 3 taking inspiration from [9]. In our case, the symbolic execution defines the effect of statements on symbolic states composed by a symbolic heap, and a read- and write-set. We represent a symbolic state as: $\langle \mathcal{H}, \mathcal{R}, \mathcal{W} \rangle \in (\mathsf{SHeaps} \times \mathsf{Rs} \times \mathsf{Ws})$ where SHeaps is the set of all symbolic heaps and Rs and Ws is the set of all read- and write-sets respectively. Please note that each read- write-set pair is always associated to a symbolic heap. We write $\mathsf{SStates}$ for denoting the set of all symbolic states.

Each transactional method is annotated with the **@Atomic** annotation describing the initial symbolic heaps for that transaction. The symbolic execution will analyse only transactional methods and all the invocation tree that occur inside their body. In the beginning of the analysis we have the specification of the symbolic heaps for each transactional method. To these initial symbolic heaps empty read- and write-sets are associated 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 is calculated by the analysis;
- the final symbolic states resulting from the method's execution. These final symbolic states are calculated by the analysis and, in the special case of transactional methods, are the final result of the analysis.

Each initial symbolic state can originate more than one final symbolic state as result of the method's analysis. The symbolic execution is a function that takes a procedure and a symbolic state, and returns the set of resulting symbolic states or an error ($\top$):

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

The specification of a method is defined as: $\mathsf{SStates} \to \mathcal{P}(\mathsf{SStates})$. The mapping between a method's signature and its specification is done by the following function:

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

Where Sig is the signature of the transactional method, which in our core language is represented as $proc(\vec{x})$ or

$func(\vec{x})$. The specification of methods that are called inside transactional methods is calculated by the symbolic execution. The method's initial symbolic state is inferred by the symbolic state of the calling context.

5.1 Past Symbolic Heap

Our separation logic fragment, depicted in Figure 6, contains a special definition of *past predicates* ($\widehat{S}$). These special predicates have an important role in the correctness for calculating abstract memory locations. It is important to note that updates made to the heap, done inside a transaction, are not visible to other concurrent transactions. This means that memory locations should always be calculated in respect to the initial snapshot of memory that is shared between concurrent transactions.

Example 2. *Given an initial specification of a symbolic heap, where x is a shared variable (*$\mathsf{shared}(x)$*):*

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

The representation of the abstract locations pointed by each variables is the following:

$$x \equiv (x, 0)$$
$$y \equiv (x, N[>0])$$
$$z \equiv (x, 1 + N[>0])$$

If we update the location pointed by y by modifying its $next$ field to nil *we get the following symbolic heap:*

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

The representation of the locations pointed by x and y remain the same. However after the update in this symbolic heap, z is no longer reachable from a shared variable. 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 representation of a memory location, that is reachable by a shared variable, must not be changed by the updates in the structure of the heap. So, in order to calculate the correct location representation we need to use a "past view" of the current symbolic heap. The get the past view we need *past* predicates, denoted by a hat on top of the predicate, 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, then we need to calculate the representation 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 12 (Past Symbolic Heap)**.** *Let* $\mathsf{NPast}(\Pi|\Sigma) = \{S \mid \Sigma = S * \Sigma' \wedge \neg\, \mathsf{hasPast}_{\Pi|\Sigma}(S)\}$ *and* $\mathsf{Past}(H)$ *the set of past predicates in H. Then*

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

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

Definition 13 (Has Past)**.**

$$\begin{array}{lcl} \mathsf{hasPast}_H(x \mapsto \rho) & \Leftrightarrow & H \vdash x\widehat{\mapsto}[\rho] * true \\ \mathsf{hasPast}_H(p(\vec{i},\vec{o})) & \Leftrightarrow & \forall i \in \vec{i} : \delta^+(i) \wedge \\ & & \exists i \in \vec{i} : H \vdash \widehat{p}(\ldots, i, \ldots) * true \end{array}$$

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

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

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

5.2 Symbolic Execution Rules

The definitions of the operational symbolic execution rules are described in Figure 9.

The rule ASSIGN, when executed in a state $\langle \mathcal{H}, \mathcal{R}, \mathcal{W} \rangle$ 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'. The read- and write-set are not changed because there are not any changes in the heap as well. The HEAP READ rule adds an equality between x and the content of the field f of object pointed by y to the resulting state. In this case, we are performing a read operation, in field f, on the memory location pointed by variable y, hence this read access is added to the read-set. The read function generates an abstract read access, based on the definitions described in Section 4.2.1, and updates the current read-set by inserting the new read access into the respective range. The read access is calculated over the past view of the current symbolic heap $\mathcal{H}$ as described in the previous section. The HEAP WRITE rules update the value of a field f of object x. There are two different rules to distinguish the case where a past predicate must be created. If already exists a past definition for variable x then we do not need to create another one because we only need the first, which corresponds to the initial symbolic heap. In these rules the write-set is properly updated to reflect the write-access made

$$\langle \mathcal{H}, \mathcal{R}, \mathcal{W}, S \rangle \Longrightarrow \langle \mathcal{H}', \mathcal{R}', \mathcal{W}' \rangle \qquad \vee \qquad \langle \mathcal{H}, \mathcal{R}, \mathcal{W}, S \rangle \Longrightarrow \top$$

$$I(e) ::= e.f := x \mid x := e.f$$

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

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

$$\frac{\mathcal{R}' = \mathsf{read}(\mathcal{H} * y \mapsto [f : z], \mathcal{R}, y, f) \qquad x' \text{ is fresh}}{\langle \mathcal{H} * y \mapsto [f : z], \mathcal{R}, \mathcal{W}, x := y.f \rangle \Longrightarrow \langle x = z[x'/x] \wedge (\mathcal{H} * y \mapsto [f : z])[x'/x], \mathcal{R}', \mathcal{W} \rangle} (\textsc{Heap Read})$$

$$\frac{\exists v \in \mathsf{Vars} : \mathsf{shared}(v) \wedge \mathsf{reach}_{\Pi}(\mathcal{H}, v, x) \qquad \mathcal{H} \not\vdash x \widehat{\mapsto} [\ldots] \qquad \mathcal{W}' = \mathsf{write}(\mathcal{H} * x \mapsto [f : z], \mathcal{W}, x, f)}{\langle \mathcal{H} * x \mapsto [f : z], \mathcal{R}, \mathcal{W}, x.f := e \rangle \Longrightarrow \langle \mathcal{H} * x \mapsto [f : e] * x \widehat{\mapsto} [f : z], \mathcal{R}, \mathcal{W}' \rangle} (\textsc{Heap Write 1})$$

$$\frac{(\forall v \in \mathsf{Vars} : \mathsf{shared}(v) \wedge \neg\, \mathsf{reach}_{\Pi}(\mathcal{H}, v, x)) \vee \mathcal{H} \vdash x \widehat{\mapsto} [\ldots] \qquad \mathcal{W}' = \mathsf{write}(\mathcal{H} * x \mapsto [f : z], \mathcal{W}, x, f)}{\langle \mathcal{H} * x \mapsto [f : z], \mathcal{R}, \mathcal{W}, x.f := e \rangle \Longrightarrow \langle \mathcal{H} * x \mapsto [f : e], \mathcal{R}, \mathcal{W}' \rangle} (\textsc{Heap Write 2})$$

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

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

$$\frac{\begin{array}{c}\mathsf{spec}_{inv}(func(\vec{x})) = \langle \mathcal{H}', \{\}, \{\} \rangle \rightarrow \langle \mathcal{H}'', \mathcal{R}', \mathcal{W}' \rangle \qquad \mathcal{H} \vdash \mathcal{H}'[\vec{y}/\vec{x}] * R \qquad \mathcal{H}''' = R * \mathcal{H}''[\vec{y}/\vec{x}] \\ \mathcal{R}'' = \mathsf{merge}_{\mathcal{H}'''}(\mathcal{R}, \mathcal{R}'[\vec{y}/\vec{x}]) \qquad \mathcal{W}'' = \mathsf{merge}_{\mathcal{H}'''}(\mathcal{W}, \mathcal{W}'[\vec{y}/\vec{x}])\end{array}}{\langle \mathcal{H}, \mathcal{R}, \mathcal{W}, x := func(\vec{y}) \rangle \Longrightarrow \langle x = ret \wedge \mathcal{H}''', \mathcal{R}'', \mathcal{W}'' \rangle} (\textsc{FCall})$$

$$\frac{\begin{array}{c}\mathsf{spec}_{inv}(proc(\vec{x})) = \langle \mathcal{H}', \{\}, \{\} \rangle \rightarrow \langle \mathcal{H}'', \mathcal{R}', \mathcal{W}' \rangle \qquad \mathcal{H} \vdash \mathcal{H}'[\vec{y}/\vec{x}] * R \qquad \mathcal{H}''' = R * \mathcal{H}''[\vec{y}/\vec{x}] \\ \mathcal{R}'' = \mathsf{merge}_{\mathcal{H}'''}(\mathcal{R}, \mathcal{R}'[\vec{y}/\vec{x}]) \qquad \mathcal{W}'' = \mathsf{merge}_{\mathcal{H}'''}(\mathcal{W}, \mathcal{W}'[\vec{y}/\vec{x}])\end{array}}{\langle \mathcal{H}, \mathcal{R}, \mathcal{W}, proc(\vec{y}) \rangle \Longrightarrow \langle \mathcal{H}''', \mathcal{R}'', \mathcal{W}'' \rangle} (\textsc{PCall})$$

Figure 9. Operational Symbolic Execution Rules

to the object pointed by x. Once again the write-access is calculated over the past view of the current symbolic heap.

In the FCALL and PCALL rules, the function spec_{inv} is used to get the symbolic state $\langle \mathcal{H}'', \mathcal{R}', \mathcal{W}' \rangle$ corresponding to the execution of a function *func* or procedure *proc*. The read- ($\mathcal{R}'$) and write-set ($\mathcal{W}'$) are renamed and merged into the read- $\mathcal{R}$ and write-set ($\mathcal{W}$) using the function merge wich recalculates the lengths of ranges in $\mathcal{R}'$ and $\mathcal{W}'$ and adds these updates ranges into $\mathcal{R}$ and $\mathcal{W}$ respectively.

Definition 14 (Read- and Write-Set Merge)**.**

$$\mathsf{merge}_{\Pi|\Sigma}(S, S') \triangleq \{s' \mid a \in \mathsf{genLens}_{\Pi|\Sigma}(S') \wedge s \in S \wedge s' = a \oplus_r s\}$$

Definition 15 (Lengths Generation)**.**

$$\mathsf{genLens}_{\Pi|\Sigma}(S) \triangleq \{(v, f, l_1 + l'), (v, f, l_2 + l') \mid (x, f, [l_1, l_2]) \in S, v \in \mathsf{vars}(\Sigma) : \mathsf{shared}(v) \wedge l' = \mathsf{length}_{\Pi}(\Sigma, v, x)\}$$

The final symbolic heap $\mathcal{H}'''$ is computed the same way as in other typical inter-procedural analysis using separation logic [8]. In the case of FCALL rule we add the equality $x = ret$ which corresponds to the returning value of function *func*.

Error states ($\top$) such as the ones produced by HEAP ERROR rule are not relevant for our analysis because is not our objective to verify the occurrence of execution exceptions. Therefore we silently discard these states from our set of results.

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 16 (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 [7] and the jStar work [8] to ensure termination of symbolic execution 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 the Star™ have the form:

$$\frac{premises}{\mathcal{H} \vdash \mathsf{emp} \rightsquigarrow \mathcal{H}' \vdash \mathsf{emp}} (\text{ABSTRACTION RULE})$$

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 show in Figure 2.

In our symbolic execution, the application of an abstraction rule does more than a simple rewrite operation. The parameter lengths of the *more* abstract predicates, which appear in $\mathcal{H}'$, are updated accordingly to the *less* abstract predicates in $\mathcal{H}$. This length update is essential to achieve very precise definitions of ranges but it is also necessary to define strict rules for this update in order to achieve convergence of the analysis.

Length updates are applied to all predicates except the points-to ($\mapsto$). For each predicate in $\mathcal{H}'$, is updated the length of each pair of *entry* and *exit* parameters in the context of the symbolic heap $\mathcal{H}$.

Definition 17 (Abstraction Rule)**.**

$$\frac{premises}{\Pi|\Sigma \vdash \mathsf{emp} \rightsquigarrow \mathcal{H}' \vdash \mathsf{emp}}$$

Where if $p(\vec{i}, \vec{o}) \in \mathcal{H}'$, $i \in \vec{i}, o \in \vec{o}$ *and* $l = \mathsf{length}_{\Pi}(\Sigma, i, o)$ *then* $\Delta^p(o, i) = K[\geq l]$

Notice that we are calculating the length l, of the path between i and o, in the less abstract symbolic heap, and then, we define the length of parameters i and o for predicate p in the more abstract symbolic heap ($\Delta^p(o, i)$) as a variable K (in $\mathcal{L}$ language) with the condition $[\geq l]$. This means that the length of the more abstract predicate is at least as the length of the path found in the less abstract symbolic heap.

The identifier of variable K is not chosen randomly. This identifier is the result of an *hash* function applied to the identifier of the predicate (p), the parameters (i and o) and the corresponding abstract rule. This identifier generation process ensures that a fixed-point is reached because the same sequence of abstraction rules generates the same identifiers. A problem may arise when length l already has in its deep-nested conditions a variable with the same identifier as K. We solve this case by partially evaluation the expression l by simplifying the condition where this same identifier occurs.

Example 3 (Partially Evaluation of Length)**.** *Give a length* l *denoting the following expression:*

$$l = n_1[\geq n_2[> 2]]$$

If the new identifier K *is equal to* n_2 *then we partially evaluate* l *returning the following expression:*

$$l = n_1[\geq 3]$$

And now we can create the new length, as described in the abstraction rule above, as $n_2[\geq n_1[\geq 3]]$.

This partially evaluation is actually counting the number of times the same abstraction is applied to the same symbolic heap $\mathcal{H}$ and this may break the convergence properties. To avoid this problem we compare lengths that appear inside read- and write-sets using Definition 4, which assert that two lengths are equal if have the same identifier.

5.5 Write-Skew Satisfiability Test

The result of the symbolic execution is a set of symbolic states for each transactional method. We can then ignore the symbolic heap part and keep only the set of pairs of read- and write-set. An example of this result is shown in Figure 3.

Each pair of read- and write-set for a given method m is an approximation of a possible concrete read- and write-set resulting from the actual execution of method m. We then have to test the satisfiability of the *write-skew* condition between pairs of results of different methods, but only in the case when such results have non-empty write sets.

We test the satisfiability by feeding an SMT solver with the information of the read-/write-set of each method and check the satisfiability of interval intersection as described at the beginning of this paper, in Section 2.

6. Experimental Results

Star™ is a prototype implementation of our static analysis applied to Java byte code, using the Soot toolkit [19] and the CVC3 SMT solver [2]. We applied Star™ to three STM benchmarks: an ordered linked list, a binary search tree, and the Intruder test program of the STAMP benchmark.

Due to limitations in the prototype implementation, which does not support arrays, we could only verify part

Benchmark	Method	Duration (sec)	#LOC	#Coverage (%)	#Final States	*Write-Skews*
List	*add*	8	16	100	6	0
	remove		15	100	6	
	contains		11	100	3	
	revert		11	100	3	
Tree	*treeAdd*	49	21	100	69	0
	treeContains		15	100	51	
Intruder	*atomicGetPacket*	51	9	77	2	(atomicProcess, *atomicGetComplete*)
	atomicProcess		173	89	17	
	atomicGetComplete		15	73	2	

Table 1. Star™ applied to STM benchmarks.

of the Intruder test. Our prototype is also limited when verifying graph-like data structures. Hence, our code coverage is between 73% and 89% for the operations in this particular test. Table 1 shows the detailed results of our verification for each transactional method of the examples above. The results were obtained in a Intel Core i5 650 computer, with 4 GB of RAM and an Intel dual-core processor.

We show the time that Star™ takes (in seconds) to verify each example, the number of lines of code, the code coverage, 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.

The linked list example, used in Section 2, is proven to be completely safe when executing all transactions under SI. The same result is reached for the BSTree example, but in this case the number of final states is much higher than in the List case.

7. Related Work

Software Transactional Memory (STM) [11, 17] systems commonly implement the full serializability of memory transactions to ensure the correct execution of the programs. To the best of our knowledge, SI-STM [16] 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 validation of read-write conflicts. Our approach avoids this validation by using static analysis and correcting the anomalies before executing the program. Bieniusa et al. [4] presents the implementation of a decentralized Dsitributed STM (DSTM) algorithm that executes under a variant of snapshot isolation. They also avoid anomalies by tracking read accesses in read-write transactions and validate the read-set at commit time.

In our work, we aim at providing the serializability semantics under snapshot isolation for STM and DSTM 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 allow 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 [18] 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 [9, 15] to model operations that manipulate heap pointers. Separation Logic has been 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 [5].

The approach described in [14] 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.

Some aspects of our work are inspired on jStar [8]. The 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. Although our work have some aspect in common with jStar, the properties being verified are completely different.

In the field of optimization, Afek et al. in [1] describes the use of standard static analysis techniques to optimize transactional code. They perform several optimizations and achieve speed ups of 29–50% for single thread runs, and of 19% for 32 threads runs. However, in the cases involving pointer manipulation, the analysis is not very precise. We believe that the use of shape analysis techniques, such as Separation Logic, would probably allow more precise and tailored code optimizations in this work.

8. Concluding Remarks

In this paper we described a novel approach to automatically verify the absence of the *write-skew* snapshot isolation anomaly in transactional memory Java programs.

The approach is based on a general model for fine grain abstract representation of memory accesses. Using this representation we accurately approximate the abstracts read- and write-sets of memory transactions. We define the *write-skew* as a consequence of the satisfiability of an assertion over abstracts read- and write-sets.

Star™ is the practical outcome of the theoretical framework laid out in this paper, unveiling the potential for further optimization of transactional memory Java programs.

References

[1] Y. Afek, G. Korland, and A. Zilberstein. In *The 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, October 2010. doi: 10.1109/IPDPS.2010.5470446.

[2] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[3] H. Berenson, P. Bernstein, J. N. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.

[4] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010. doi: 10.1109/IPDPS.2010.5470446.

[5] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proc. of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 289–300, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[8] D. Distefano and M. J. Parkinson. jstar: towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented Programming Systems Languages and Applications (OOPSLA'08)*, pages 213–226, New York, NY, USA, 2008. ACM.

[9] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of 12th International Conference (TACAS 2006)*, Lecture Notes in Computer Science, pages 287–302. Springer, 2006.

[10] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[11] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[12] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1263–1274, Vienna, Austria, 2007. VLDB Endowment.

[13] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MultiProg 2010: Programmability Issues for Heterogeneous Multicores*, 2010. http://www.deucestm.org/.

[14] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *Proceedings 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*, pages 348–362, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3.

[15] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9.

[16] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *TRANSACT'06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Canada, June 2006.

[17] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[18] Transaction Processing Performance Council. *TPC-C Benchmark, Standard Specification, Revision 5.11*. Feb. 2010.

[19] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.