Detection of Snapshot Isolation Anomalies in Software Transactional Memory: A Static Analysis Approach

Ricardo J. Dias, João M. Lourenço and João Costa Seco*
CITI and Departamento de Informática
Universidade Nova de Lisboa, Portugal
{rjfd, joao.lourenco, joao.seco}@di.fct.unl.pt

Technical Report: UNL-DI-5-2011

May 2011

Abstract

Some performance issues of software transactional memory are caused by unnecessary abort situations where non serializable and yet non conflicting transactions are scheduled to execute concurrently. By smartly relaxing the isolation properties of transactions, transactional systems may overcome these performance issues and attain considerable gains. However, it is known that relaxing isolation restrictions may lead to execution anomalies causing programs to yield unexpected results. Some database management systems make that compromise and allow the option to operate with relaxed isolation mechanisms. In this cases, developers must accept the fact that they must explicitly deal with anomalies. In software transactional memory systems, where transactions protect the volatile state of programs, execution anomalies may have severe and unforeseen semantic consequences. In this setting, the balance between a relaxed isolation level and the ability to enforce the necessary correctness in general purpose language programs is harder to achieve.

The solution we devise in this paper is to statically analyze programs to detect the kind of execution anomalies that emerge under snapshot isolation. We propose a semantic approach to the challenging scenario of analyzing programs with dynamic allocated data structures. Although limited to acyclic data structures, our analysis is able to detect anomalies in well know examples of anomalous code. Our approach allows a compiler to either warn the developer about the possible snapshot isolation anomalies in a given program, or to possibly inform automatic correctness strategies to ensure execution under full serializability.

1 Introduction

Using relaxed isolation levels, such as Snapshot Isolation (SI) [2], is a well known and long used strategy in database transactional systems to increase performance. Software Transactional Memory [11, 5] (STM) usually requires full serializability but, in principle, can also use SI, although with different semantic guaranties.

Unlike full-fledged STM systems that provide strict isolation between transactions, STM systems providing relaxed isolation levels allow for transactions to interfere, and generate non-serializable execution schedules. Such interferences are commonly known as *serializability anomalies* [2]. One of these anomalies, the *write-skew*, occurs when two transactions are writing on disjoint memory locations and are entangled in reading data that is being modified by the other. For instance, consider the implementation of an ordered linked list that is being accessed in the context of a transactional memory program. The *insert* operation has a footprint of read(`R`)/write(`W`) operations (on the `next` field of `list` nodes) that can be depicted as follows:

$$\textit{insert}:\ \texttt{R} \longrightarrow \ \ldots \ \longrightarrow \texttt{R} \longrightarrow \texttt{R/W}$$

The invariant here is that the *insert* operation does not read beyond the node it writes into. If transaction A reads

*This research was partially supported by the EU COST Action IC1001 (Euro-TM) and the Portuguese Fundação para a Ciência e Tecnologia in the research projects PTDC/EIA/74325/2006 (Byzantium), PTDC/EIA-CCO/104583/2008 (StreamLine), and PTDC/EIA-EIA/113613/2009 (Synergy-VM), and the research grant SFRH/BD/41765/2007.

a node that is being modified by transaction B, from the invariant above we can infer that either transaction A is writing on the same node as transaction B (and one of them will necessarily abort), or transaction A will modify some other node ahead of the one modified by transaction B. Hence, if two transactions A and B are set to insert a value in the list then no anomaly can occur.

If one considers the *remove* operation which reads the field `next` of the node being removed, its footprint is:

$$\textit{remove}: \texttt{R} \longrightarrow \ldots \longrightarrow \texttt{R} \longrightarrow \texttt{R/W} \longrightarrow \texttt{R}$$

When a *remove* operation is set to run concurrently with other *remove* or *insert* operations, then a write-skew anomaly may be observed. There is a crossed read/write situation when one transaction removes a node while the other is removing or inserting a new node just ahead of it. By introducing a minor correction in the code, an additional (dummy) write in the node to be removed, the footprint for a new *remove'* operation becomes

$$\textit{remove'}: \texttt{R} \longrightarrow \ldots \longrightarrow \texttt{R} \longrightarrow \texttt{R/W} \longrightarrow \texttt{R/W}$$

and *insert* and *remove'* operations can now be executed concurrently under SI with no anomalies ever occurring.

In this paper we introduce a static analysis technique and corresponding algorithm that mechanically asserts that a given transactional memory application executing under Snapshot Isolation behaves as if executed under full serializability. More precisely, we detect the kind of anomalies of Snapshot Isolation that are known to lead to non-serializable histories [2]. We build on related work directed to database transactions [4, 6] and extend it to the very different domain of transactional memory. Our technique allows for the automatic identification of anomalous code patterns which were previously detected only by ad-hoc code inspection. To the best of our knowledge, SI has only been applied to STM in semantically harmless situations [9], where anomalies do not occur. Our technique allows SI to be used as the default isolation level for STM applications by signaling conflicting programs and allowing for explicit code corrections.

We take a standard approach to define our analysis, and use a simple core imperative language with limited use of pointer indirection and support for heap allocated data. Applications written in main stream programming languages are translated to the core language and then analyzed. For each transaction in the original application we create a separate program in the core language. We assume that all transactions in the application can be run concurrently, and define an intra-procedural data-flow analysis that extracts the information necessary to detect SI anomalies.

The analysis starts by extracting compact read/write sets and defining static dependencies between programs, thus creating a static dependency graph. We then apply an algorithm to that graph to determine if the concurrent execution of such transactional programs is serializable under SI.

The analysis of heap allocated data follows a modified shape analysis technique [10], that combines shape graphs with sets of read and written data items. We define a set of new properties for shape graphs in order to avoid the state explosion that would result from the application of the original definition to our setting. Unlike in standard shape analysis procedures, termination in our approach does not depend on the comparison of shape graphs but rather on the collected read/write-sets, whose computation converges faster. A drawback of our definition is that it can only be applied to acyclic data structures, but we foresee that the model can be extended with backward pointers to support more general cases.

Like in any other static analysis procedure, we allow for some kind of false positive results. Nevertheless, we guarantee that if the analysis procedure does not detect any serializability anomaly, then all possible executions will be anomaly-free and correspond to a possible interleaving of the transactions. On the other hand, if some anomalies are found, they should be considered as potentially harmful. At this stage, analysis results can be further refined and code can be modified to avoid undesired interferences. The application of our approach to real programs is more adequate if allied the aggregation of memory transactions in independent modules.

In summary, we introduce

- a shape graph definition that supports acyclic data structures and limits the explosion of the state space;
- an intra-procedural data-flow analysis that extracts fine grained read and write set information from transactional programs;
- a procedure to identify static program dependencies from the results of the data-flow analysis phase;
- an enhanced algorithm to detect SI anomalies based on the graph of the static program dependencies.

We now proceed by introducing the background work in Section 2, namely the properties of snapshot isolation and the detection of snapshot isolation anomalies using dependencies between programs. We follow in Section 3 with the description of an intra-procedural data-flow analysis for a simple imperative language using extended shape graphs. Given the results of the data-flow

```
void Withdraw(boolean b, int value) {
  if (x + y > value)
    if (b) x = x - value;
    else y = y - value;
}
```

Figure 1: Withdraw program.

analysis, in Section 4 we describe the construction of a graph of program dependencies, which is used by the algorithm described in Section 5 for detecting SI anomalies. We proceed by presenting the results obtained when applying the technique to a known benchmark in Section 6, and relating with the work of others in Section 7. We end with the concluding remarks and some references to future work.

2 Background

2.1 Snapshot Isolation

Snapshot Isolation [2] is a relaxed isolation level where each transaction executes with relation to a private copy of the system state, taken in 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 refers 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.

Although appealing for performance reasons, the application of SI may lead to non-serializable executions, resulting in two kinds of consistency anomalies: *write-skew* and *SI read-only anomaly* [4]. Consider the following example that suffers from the *write-skew* anomaly. A bank client can withdraw money from two possible accounts represented by two shared variables, x and y. The program listed in Figure 1 can be used in several transactions to perform bank operations customized by its input values. The behavior is based on a parameter b and on the sum of the two accounts. Let the initial value of x be 20 and the initial value of y be 80. If two transactions execute concurrently, one calling the Withdraw(**true**, 30) (T_1) and the other calling the Withdraw(**false**, 90) (T_2), then one possible execution history of the two transactions under SI is:

$$H = R_1(x,20)\ R_2(x,20)\ R_1(y,80)\ R_2(y,80)\ R_1(x,20)\ W_1(x,-10)\ C_1\ R_2(y,80)\ W_2(y,-10)\ C_2$$

After the execution of these two transactions the final sum of the two accounts will be -20, which is a negative value. Such execution would never be possible under Serializable isolation level, as the last transaction to commit would abort because it read a value that was written by the first transaction.

2.2 Program Dependencies and Anomalies

Serialization graphs are the most common formal tool to define serializable anomalies. In a serialization graph, nodes correspond to transactions and edges correspond to data dependencies between transactions. These graphs are build upon dynamic information collected during execution of programs. Three types of transaction dependencies can de defined [4]:*write-read* (wr), *write-write* (ww) and *read-write* (rw).

The static counter-part of the (dynamic) serialization graphs is the *static dependency graph* (SDG). In a SDG, nodes are programs instead of transactions and edges are data dependencies between programs. If executed more than once, a single program P may define the behavior of multiple transactions.

There is a static dependency between programs P_n and P_m, written $P_n \xrightarrow{x-\rho} P_m$, where x is a state variable and ρ is a dependency type ($\rho \in \{wr, ww, rw\}$) if, for any two transactions T_n and T_m resulting from the execution of P_n and P_m respectively, there is a transactional dependency $T_n \xrightarrow{x-\rho} T_m$ on any variable x.

A static dependency between programs P_n and P_m is said to be vulnerable if it is of type rw and the corresponding transactions T_n and T_m may execute concurrently. A vulnerable static dependency on a variable x between programs P_n and P_m is represented as $P_n \xRightarrow{x-rw} P_m$. Note that a single program P may generate multiple transactions, hence in a SDG the program may have dependencies to itself. In summary, the $SDG(\mathcal{A})$ of an application $\mathcal{A}$ is a graph with programs $P_1, \ldots, P_k$ of $\mathcal{A}$ as nodes, and with edges labeled as $P_n \xrightarrow{x-\rho} P_m$ (non-vulnerable) or $P_n \xRightarrow{x-rw} P_m$ (vulnerable) representing static dependencies.

Serializable anomalies can be signaled by the presence of certain kinds of dangerous structures in the SDG of an application. Fekete et al. [4] define the concept of *dangerous structure* in a static dependency graph. The $SDG(\mathcal{A})$ of an application $\mathcal{A}$ has a dangerous structure

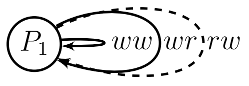

Figure 2: Static dependency graph of the Withdraw program.

if it contains three nodes P, Q and R (not necessarily distinct) such that there are vulnerable edges from R to P and from P to Q, and there is a path from Q to R. Node P is called the *pivot* node. If the $SDG(\mathcal{A})$ of an application $\mathcal{A}$ has a dangerous structure, then some executions of $\mathcal{A}$ may be non-serializable. In the opposite, is the $SDG(\mathcal{A})$ has no dangerous structures, then all executions of $\mathcal{A}$ are serializable.

From the Withdraw program depicted in Figure 1 we can generate the static dependency graph in Figure 2. There is one node in the graph representing program Withdraw (P_1) and there are three edges[1]: non-vulnerable $P_1 \xrightarrow{ww} P_1$ and $P_1 \xrightarrow{wr} P_1$, and vulnerable $P_1 \overset{rw}{\Longrightarrow} P_1$. The vulnerable edge is represented by a dashed arrow in the diagram. Intuitively the three edges represent the following situations: the dependency $P_1 \xrightarrow{ww} P_1$ results from sequentially calling the program P_1 twice with the same value for parameter b, thus generating non-concurrent transactions; dependency $P_1 \xrightarrow{wr} P_1$ results from sequentially calling P_1 twice with different values for parameter b; and dependency $P_1 \overset{rw}{\Longrightarrow} P_1$ results from calling P_1 twice with different values for parameter b, and the two corresponding transactions are concurrent.

According to the definition, the SDG in Figure 2 has a dangerous structure. Note that this does not imply that the corresponding application will necessarily have a SI anomaly, but only that it may have one.

We next show how to build a SDG by analyzing the source code of an application, and how to detect dangerous structures that point to possible SI anomalies.

3 Static Analysis

We now define a data-flow analysis based on a small imperative language with heap allocated memory [7] and then describe how to build a SDG from the results of the analysis. The abstract syntax of the language is the following:

$$\begin{array}{lcl} R & ::= & x \mid x.sel \\ E & ::= & R \mid n \mid E\ op\ E \mid \textbf{true} \mid \textbf{false} \mid \textbf{not}\ E \mid \textbf{null} \\ S & ::= & R := E \mid \textbf{skip} \mid S\,;S \mid \textbf{new}\ R \\ & \mid & \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{while}\ E\ \textbf{do}\ S \end{array}$$

[1]The variable identifiers are omitted for the sake of simplicity.

```
//x points to the head of a list
//v is the value to be inserted
p := x;
n := p.next;
tv := n.value;
while(tv < v) {
  p := n;
  n := p.next;
  tv := n.value;
}
if (tv != vv) {
  new y;
  y.value := v;
  y.next := n;
  p.next := y;
}
```

Figure 3: Example of an insertion of a value into a linked list.

Where a program is a statement S. The values of the language are integer values (n), boolean values, pointers to heap cells or the special value **null**. Arithmetic operators are only defined for integer values. Boolean expressions are extended to support *equality of pointers* and to support the comparison of pointers with the special value **null**. The statement **new** R assigns the pointer of a new heap cell to R.

Figure 3 describes a program that implements an *insert* function in an ordered singly linked list. By convention, the value to be inserted is given in variable v and the pointer for the head of the list is given by variable x. The program uses local variables n, tv and vv, and the nodes in this list implementation are heap cells with two selectors $value$ and $next$.

We follow a general model of a heap cell where its contents are only accessible through a finite number of selectors (cf. fields in structures). For instance, one may instantiate a cell and assign the resulting pointer to a variable x by using the statement **new** x. The contents of the cell may be changed by assigning to a selector val, as in statement $x.val := 1$. Note that statement $y := x$ results in a state where x and y point to the same heap cell, and that statement $x := 1$ changes the value of variable x from a pointer to an integer value. Selectors work like a dereferencing operator and also provide language support for dynamically allocated structure-like values.

As expected, the semantics of our language is defined with relation to a state and a heap. The state is a mapping of variables to values, and the heap is a mapping from heap cells to values. The size of a program state is bounded by the number of variables that occur in the source code of the program, and the heap is unlimited in size and structure. In order to analyze the executions of a program we use a compact and bounded representation of the state and heap, similar to the standard notion of *Shape Graph* [10].

In the remaining of this document, we present the definition of shape graphs that will be used to identify heap accesses during the data-flow analysis. We then define a data-flow analysis that generates, for each analyzed program, a set of read/write states for variables and heap accesses. The information retrieved from the analysis is used to generate a static dependency graph which is traversed by an algorithm to search for dangerous structures.

3.1 Shape Graphs

A shape graph is an abstract representation of the state and heap of a program, defined from the notion of abstract location, which is the representative for one (or more) heap cells in the program heap. A shape graph is composed by an abstract state $\mathcal{S}$, which is a mapping from variable names to abstract locations, and an abstract heap $\mathcal{H}$, which is a mapping from abstract locations to abstract locations by means of selectors. We write n_X to denote an abstract location, where $X \subseteq \mathit{Var}$ is the set of state variables pointing to that location. In the general case, abstract locations are associated to one (and only one) heap cell. When $X = \emptyset$ we call $n_\emptyset$ the summary location. In this particular case, $n_\emptyset$ is the representative of more than one heap cell, more precisely, of all the heap cells that are not directly pointed by a state variable. For instance, $n_{\{x,y\}}$ is the abstract location that represents a heap cell pointed by the state variables x and y. This also means that variables x and y are aliases of the same heap cell.

The original definition of a shape graph by Sagiv et al. [10] also includes information about sharing of abstract locations, signaling those that are reachable by more than one abstract location. On the one hand, we adopt a simpler version, dropping this kind of information, and hence we only represent acyclic memory data structures. On the other hand, we extend the notion of summary node. Instead of using a single summary node $n_\emptyset$, we use a set of summary nodes indexed by an integer number $n^i_\emptyset$ ($i \in \mathbb{N}_0$). Summary node $n^0_\emptyset$ plays the usual role of aggregating a set of heap cells. We usually omit the index if it is 0 and write only $n_\emptyset$. Summary nodes with positive indexes, $n^i_\emptyset$ with $i > 0$, represent abstract locations occurring in the middle of a path of an acyclic data structure which are not directly pointed by state variables.

We define an abstract state of a program $\mathcal{S} \triangleq \mathit{Var} \times \mathit{ALoc}$, mapping from variables to abstract locations, and an abstract heap $\mathcal{H} \triangleq \mathit{ALoc} \times \mathit{Sel} \times \mathit{ALoc}$, mapping from abstract locations to abstract locations via selectors. Here, ALoc is the infinite set of all abstract locations and Sel is the finite set of all selectors. We use $\mathit{Locs}(\mathcal{S})$ and $\mathit{Locs}(\mathcal{H})$ to signify the set of abstract locations used in a abstract state or abstract heap. The set of all Shape Graphs is thus $\mathcal{SG} \triangleq \mathcal{S} \times \mathcal{H}$, and a shape graph SG is a pair (S, H) with $S \in \mathcal{S}$ and $H \in \mathcal{H}$. We refine the notion of shape graph and define an *Acyclic Shape Graph* as a shape graph that satisfies some conditions to cope with acyclic data structures.

Definition 1 (Acyclic Shape Graph)**.** *For all program states $S \in \mathcal{S}$ and heaps $H \in \mathcal{H}$*

1. *A state variable can only point to an abstract location:*

   $\forall n_V, n_W \in \mathit{Locs}(S) \cup \mathit{Locs}(H) : (V = W) \vee (V \cap W = \emptyset)$

2. *State variables pointing to an AL are referred in its variable set.*

   $\forall (x, n_X) \in S : x \in X$

3. *A selector of an AL points at most to one AL.*

   $\forall (n_V, s, n_W), (n_V, s, n_{W'}) \in H : (W = W')$

4. *No AL is pointed by any two ALs (the heap is acyclic).*

   $\forall (n_V, s, n_X), (n_W, s', n_X) \in H : V \neq W \Rightarrow X = \emptyset$

5. *The summary node only points to itself.*

   $\forall (n^0_\emptyset, s, n_V) \in H : V = \emptyset$

Since we only generate acyclic shape graphs and for the sake of simplicity, in the remaining of this paper, and unless stated otherwise, we will refer to acyclic shape graphs simply as shape graphs.

Figure 4 depicts a shape graph representing a heap containing a singly linked list. Circles represent state variables and rectangles represent abstract locations. Edges between abstract locations are labelled with the corresponding selector. This shape graph is defined as follows:

$$\begin{aligned} SG \triangleq (&\{(x, n_{\{x\}}), (y, n_{\{y\}}), (z, n_{\{z,w\}}), (w, n_{\{z,w\}})\},\\ &\{(n_{\{x\}}, \mathit{next}, n_{\{y\}}), (n_{\{y\}}, \mathit{next}, n_{\{z,w\}}),\\ &(n_{\{z,w\}}, \mathit{next}, n_\emptyset), (n_\emptyset, \mathit{next}, n_\emptyset)\}) \end{aligned}$$

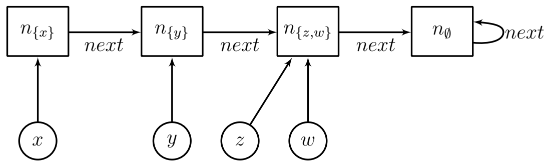

Figure 4: Shape Graph representing a singly linked list.

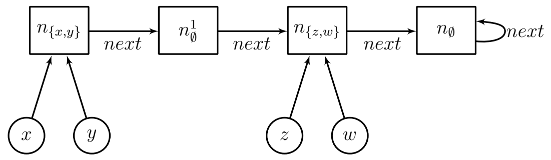

Figure 5: Result of the assignment $y := x$.

A shape graph SG is modified by the evaluation of assignments and by the cell creation statement (**new** x). We will now illustrate the effect of these statements starting with SG of Figure 4 and divide the assignments in four different fundamental forms: from variable to variable ($x := y$), from selector to variable ($x := y.sel$), from variable to selector ($x.sel := y$) and from selector to selector ($x.sel_1 := y.sel_2$).

Variable to variable: $[x := y]$ If x is a pointer, the first effect of this type of assignments is that the variable x must be removed from the set of variables of the abstract location pointed by x. In the case where the abstract location is only referred by variable x, written $n_{\{x\}}$, removing the reference results on replacing it by a summary location. We replace it by a new summary location $n^{i+1}_\emptyset$ where i is the maximum index currently found in the abstract heap.

The second visible effect of the assignment is the creation of new bindings. This only occurs when the value being assigned to variable x is a pointer. In this case, the abstract location pointed by y, n_Y is updated to $n_{Y\cup\{x\}}$.

Take the example of the shape graph of Figure 4. The effect of the assignment $y := x$ is the removal of the binding of state variable y to the abstract location $n_{\{y\}}$. This results in the shape graph of Figure 5 where $n_{\{y\}}$ is replaced by $n^1_\emptyset$, and the update of the bindings of both variables x and y result in replacing the abstract location $n_{\{x\}}$ with $n_{\{x,y\}}$.

Selector to Variable $[x := y.sel]$ As before, the bindings of x must first be removed, and if $y.sel$ is not a pointer this is the only effect of the assignment.

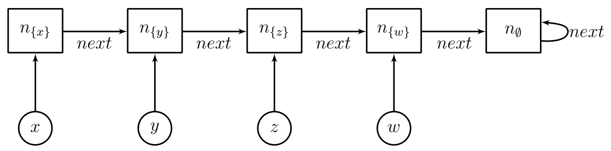

Figure 6: Result of the assignment $w := z.next$.

When $y.sel$ is a pointer, it either points to an abstract location n_V with $V \neq \emptyset$, to an intermediate summary location $n^i_\emptyset$ with $i > 0$, or to the summary location $n_\emptyset$.

In the first case, the bindings described in the abstract location n_V are updated to $n_{V\cup\{x\}}$, and the connections in the new graph are such that the state variable x now points to $n_{V\cup\{x\}}$.

In the second case, when $y.sel = n^i_\emptyset$ with $i > 0$, we have that the abstract location is replaced by $n_{\{x\}}$, which is pointed by the state variable x in the resulting graph.

In the third case, when $y.sel = n_\emptyset$, we must *materialize* a new abstract location $n_{\{x\}}$ from the summary location. All links from $n_\emptyset$ to itself must now be created (with the same selectors) from $n_{\{x\}}$ to $n_\emptyset$. Also, $y.sel$ must be updated to point to $n_{\{x\}}$.

Figure 6 illustrates the effect of assignment $w := z.next$ in a heap represented by the shape graph of Figure 4. There is a new abstract location in the graph, $n_{\{w\}}$ pointed by w, and the abstract location pointed by z changed to $n_{\{z\}}$. The cyclic reference that the summary node $n_\emptyset$ has to itself in Figure 4, through selector $next$, causes a reference from the new abstract location $n_{\{w\}}$ to $n_\emptyset$ to appear in Figure 6. The reference from $n_{\{z\}}$ to $n_\emptyset$ in Figure 4 is modified in Figure 6 to a reference from $n_{\{z\}}$ to $n_{\{w\}}$, again, through selector $next$.

Variable to selector $[x.sel := y]$ When the value denoted by y is an abstract location, the resulting graph after this assignment has an updated edge from n_X, where $x \in X$, to the abstract location pointed by y, through selector sel.

Selector to selector $[x.sel_1 := y.sel_2]$ The effect of this assignment statement can be understood by composition of a sequence of previous assignment forms. It can be re-written by the sequence: $t := y.sel_2;\ x.sel_1 := t;\ t :=$ **null**, where variable t is temporary.

Cell creation [**new** x] In the shape graph resulting from evaluating a cell creation statement, the old binding of variable x is removed as in all other cases, and a new abstract location $n_{\{x\}}$ pointed by x is added to the shape graph. In the case of the statement **new** $x.sel$, it can be explained by the sequence: **new** t; $x.sel :=$ t; $t :=$ **null**, where variable t is temporary.

We use shape graphs in the data-flow analysis to model the heap cells used by a transaction. We model the usage of a heap cell by a triple with a variable name, an integer number, and a selector. The pair composed by the variable name x and the integer value n refers to the set of abstract locations that are reachable from the variable x in n number of hops. All abstract locations identified by the same variable and distance measure are indistinguishable from now on, which is a conservative approach in terms of the analysis. So, when modeling the usage of one particular heap cell we are really referring to a set of abstract locations. The usage of an abstract location is completed with a selector. This distance measure of an abstract location corresponds to the length of the path from a state variable in the heap, its root variable, and the abstract location.

The distance of an abstract location to the program state is calculated by the maximum number of edges in the path that lead from one to the other, ignoring the number of edges pointing to summary nodes $n^i_\emptyset$ with $i > 0$. Given a shape graph SG and an abstract location n_x, function $dist(SG, n_x)$ computes a pair with the root variable x for n_x, the farthest variable in the program state, and the distance between them. For example, in the shape graph of Figure 5, the result of $dist(SG, n_{\{x,y\}})$ is $(x, 1)$ and the result of $dist(SG, n_{\{z,w\}})$ is $(x, 2)$. When more than one state variable can be root variable we choose the lexicographically smaller one.

In the subsequent analysis, we use function id that retrieves the root variable and corresponding distance for an expression accessing an abstract location. Function id is defined as follows $id : \mathcal{SG} \times Var \times Sel \longrightarrow Var \times \mathbb{N} \times Sel$ where $id(SG, x, sel) \triangleq (dist(SG, n_x), sel)$ where n_x is such that $SG = (S, H)$ and $(x, n_x) \in S$. For example, when analyzing the expression $z.next :=$ **null** with relation to the shape graph of Figure 5, our algorithm computes $id(SG, z, next)$ and obtains $(x, 2, next)$.

3.2 Read-Write Analysis

In our language, transactional applications consist of several programs, each one representing a separate transaction. In order to define static dependencies between these programs we need to know which data items they write or read. We start with a finite set of shared variables that all programs can read and write. We then use a standard data-flow analysis to obtain the set of variables and heap accesses that each program performs. We hereafter refer to both shared variables and heap accesses as data items.

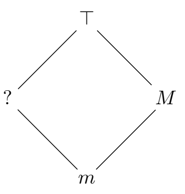

Figure 7: Order relation $\sqsubseteq$ diagram in Γ.

Our data-flow analysis associates data items to a read-/write state. A read/write state of a data item is a pair of values from the set $\Gamma = \{?, M, m, \top\}$. The first component of the pair indicates the read state of the data item, and the second component indicates its write state. A ? value in the read/write state for a data access x means that x **is not** read/written by the program. A M value in the read/write state for a data access x means that x **is indeed** read/written by the program. A m value in the read/write state for a data access x means that x **may be** read/written by the program (at least by one possible execution path). Looking at a pair of the form $(M, m)_x$, it means that data item x is read in all possible executions of the program and that it is written at least in of the possible executions of the program.

We now define the data-flow transfer functions over a lattice defined from the set $\Upsilon = \Gamma \times \Gamma \times (Cell \cup Var)$, and the order relation $\sqsubseteq$ depicted in Figure 7 on Γ. Remember that Var is the set of all state variable and consider $Cell$ as the set of all heap accesses defined as $Cell = Var \times \mathbb{N} \times Sel$. Notice that a heap access refers to all abstract locations at a certain depth from a root state variable in the data structure. In spite of a precision loss, the usage of heap accesses is conservative with relation to singular abstract locations and is still highly informative.

Besides the read/write states for data items, the transfer functions of our analysis also compute a set of shape graphs on each node of the control-flow graph. Shape graphs are used to identify the abstract locations being read or written by the statements. We need to manually make the bootstrap of the shape graph by choosing the shape graph that better describes the invariant of the heap, and associating it to the starting node of the control-flow graph. For instance, to analyze the code of a program based on a singly linked list we need to use a shape graph as depicted in Figure 8. The automatic generation of bootstrap shape graphs is out of the scope of this work.

We define our analysis in a lattice over the set $\Lambda \triangleq \Upsilon \times \mathcal{P}(\mathcal{SG})$ with the order relation applied to the set Υ

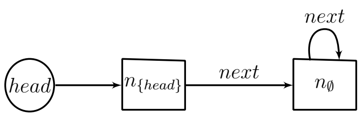

Figure 8: A bootstrap shape graph of a singly linked list.

SG / Vars	**x**	**y**	**z**
A	1	2	3
B	1	1	2
$A(d = 1)$	1	1	2

Table 1: Variable distances for shape graphs A and B.

and the transfer functions $\mathcal{R}_{en}$, which computes the entry point of a node of the control flow graph, and $\mathcal{R}_{ex}$ that computes the exit point of a node of the control flow graph. We define the functions such that the analysis proceeds *forward* in the graph.

A key point for the convergence of the data-flow analysis is that only the set Υ is taken into account in the order relation, thus making the functions monotonic. Shape graphs in the node annotations are only used to identify the used heap cells.

Each transfer function works over an ordered pair where the first entry is the lattice Υ and the second entry is the set of shape graphs that represent the state and the heap. Below, we use the notation ${}^1p$ to denote the first component of a given pair p, and ${}^2p$ to denote the second component. The entry function $\mathcal{R}_{en}$ is defined as follows:

Definition 2 (Entry function)**.**

$$\mathcal{R}_{en}(l) = \begin{cases} (\emptyset, \{SG\}) & \textit{if } l = init(P) \\ (\bigsqcap\{{}^1\mathcal{R}_{ex}(l') \mid (l', l) \in flow(P)\}, \\ \quad \bigoplus\{{}^2\mathcal{R}_{ex}(l') \mid (l', l) \in flow(P)\}) & \textit{otherwise} \end{cases}$$

The case for the initial node of the CFG, the entry funciton defines that the read/write state set is empty, and that the set of shape graphs contains only the bootstrap shape graph SG. In the case for the other nodes, the results of the exit functions of the predecessors of the current node (which are pairs) are merged by means of the *greatest lower bound* ($\bigsqcap$) for the set of elements on the first component (of the resulting pairs), and using a choice operator $\bigoplus$ to combine the shape graphs in the second component of the pairs. The choice operator $\bigoplus$ merges shape graphs according to the relation $\mathcal{K}$ whose description follows. We use $flow(P)$ to denote the set of all edges in the control-flow graph.

Two shape graphs A and B are in the relation $\mathcal{K}$ in one of three cases. In the first case, we consider strict equivalence of graphs. In the second case, two shape graphs are in the relation $\mathcal{K}$ if both have the same state variables, and if all abstract (non-summary) locations have the same distance to the state of the program on both graphs. Recall the $dist(SG, ALoc)$ function defined earlier on this section. In the third scenario, we consider that two shape graphs are equivalent if by decreasing once the distance of a well determined set of abstract locations to the program state, we obtain an equivalent shape graph by one of the other two cases.

To explain how we build the set of abstract locations to be decreased we use the example depicted in Figure 9. Shape graph A is equivalent to shape graph B by the third case since the merging of the two abstract locations $n_{\{x\}}$ and $n_{\{y\}}$ in shape graph A results in a shape graph equivalent to B.

Pick a number $d \in \mathbb{N}$ to use as threshold, we say that A is equivalent to B if for all non-summary abstract locations in A, whose distance is less or equal than d, their corresponding distances in B are the same, and for all non-summary abstract locations in A, with distance greater than d, their distances in A are greater by one than the corresponding distances in B. Table 1 shows the distance of all non-summary abstract locations for shape graphs A and B in Figure 9. By choosing $d = 1$ as a threshold, and decreasing all the distances greater than d, shape graph A becomes equivalent to shape graph B. In this case, it is visible that shape graph A subsumes shape graph B. We use this to define the choice operator to return shape graph A and discard shape graph B.

Before we define the exit function $\mathcal{R}_{ex}$ we need to define some auxiliary functions: $sameDist$, $write$ and $read$. Function $sameDist$ asserts if an abstract location pointed by a state variable is at the same distance in all elements of a given set of shape graphs. Function $write$ computes the write state of an expression, if the expression denotes a pointer, then it makes use of the shape graph to retrieve an identifier. Function $read$ is dual to function $write$ for read states. Functions $read$ and $write$ use a binary operator $\lhd$, which takes a complete read-/write state and a new state for a particular variable, and returns a state with that variable replaced by the given state.

Definition 3 (Record a Read Operation)**.**

$$read(U, SG, E) = \begin{cases} U \lhd \{(M, \hat{\sigma}_x(U))_x\} \text{ if } E = x \\ U \lhd \{(M, \hat{\sigma}_i(U))_i \mid sg \in SG \wedge i = id(sg, x, sel)\} \\ \qquad \text{if } E = x.sel \wedge sameDist(SG, x) \\ U \lhd \{(m, \hat{\sigma}_i(U))_i \mid sg \in SG \wedge i = id(sg, x, sel)\} \\ \qquad \text{if } E = x.sel \wedge \neg sameDist(SG, x) \end{cases}$$

We write $\hat{\sigma}$ to denote the previous write state of a data item.

If the expression is a variable access then the read state is updated to value M and the write value in the state is kept as it was ($\hat{\sigma}_x(U)$). In the case of a selector access, x must be a pointer and if the distance to the program state of that abstract location is the same in all possible heap configurations (shape graphs), then the value is updated to M since we are certain this abstract location is going to be read. If we cannot establish a single distance for all shape graphs, then the value is updated to m. Definition of function $write$ is very similar to the definition above but it updates the write state part and keeps the read state untouched.

We also use function $eval(SG, S)$ to obtain the effect of executing a statement in a heap modeled by a given shape graph according. We omit a formal definition of this function (that can be found in a technical report [*omitted for review purposes*]) since its intuitive semantics is given in Section 3.1. Function $blocks(P)$ denotes the set of all elementary blocks in the control-flow graph.

We can now define the exit function $\mathcal{R}_{ex}$ that computes the read/write states of all data items:

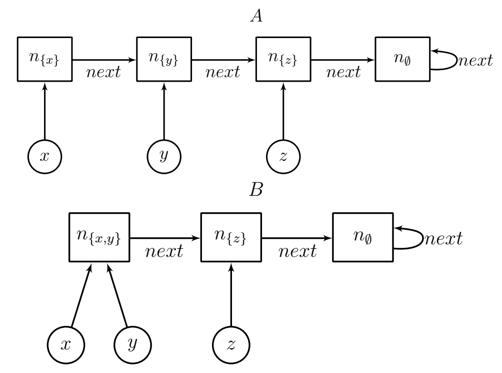

Figure 9: Example of two equivalent shape graphs.

Definition 4 (Exit Function)**.**

$$\mathcal{R}_{ex}(l) = \begin{cases} (write(U, SG, R), \{sg' \mid sg \in SG \\ \qquad \wedge\, sg' = eval(sg, [R := E])\}) \\ \qquad \textit{if } B^l = [R := E], \textit{ where } B^l \in blocks(P) \textit{ and } E \neq (y|y.sel'), \\ \qquad \textit{and } \mathcal{R}_{en}(l) = (U, SG) \\ (write(U, SG, R) \lhd read(U, SG, E), \{sg' \mid sg \in SG \\ \qquad \wedge\, sg' = eval(sg, [R := E])\}) \\ \qquad \textit{if } B^l = [R := E], \textit{ where } B^l \in blocks(P) \textit{ and } E = (y|y.sel'), \\ \qquad \textit{and } \mathcal{R}_{en}(l) = (U, SG) \\ (read(U, SG, E), SG) \\ \qquad \textit{if } B^l = [E], \textit{ where } B^l \in blocks(P) \textit{ and } E = (x|x.sel), \textit{ and } \mathcal{R}_{en}(l) = (U, SG) \\ (write(U, SG, R), \{sg' \mid sg \in SG \\ \qquad \wedge\, sg' = eval(sg, [newR])\}) \\ \qquad \textit{if } B^l = [newR], \textit{ where } B^l \in blocks(P) \textit{ and } R = (x|x.sel), \textit{ and } \\ \qquad \mathcal{R}_{en}(l) = (U, SG) \\ \mathcal{R}_{en}(l) \qquad \textit{otherwise} \end{cases}$$

On all cases of the exit function, we compute a new shape graph based on the effect of the statement in the current node, and record the effect of writing or reading a data item in the resulting set.

The first case records the effect of an assignment of a literal value to a state variable or heap cell, it records a writing on the abstract location denoted by the left hand side of the assignment. The second case defines the effect of assigning a pointer to a state variable or heap cell, it records a writing on the abstract location on the left hand side and reading the abstract location on the right hand side. The third case is of an expression reading a data item. The forth case is a creation of a heap cell and records a writing on the given state variable (x) or abstract location ($x.sel$). In all other cases, the effect is void and the set is the same as it is in the entry of the node.

The final result of this analysis will be a set of read-/write states for several shared variables and several heap accesses (sets of abstract locations). For instance the result of the analysis when applied to the program of Figure 3 is:

$$\{(M,?)_x, (M,?)_{(x,1,next)}, (m,m)_{(x,2,next)}, (m,?)_{(x,2,value)}, (M,?)_{(x,3,value)}\}$$

In the cases where we have more than one type of selector that is used as a path selector in a data structure (e.g., binary tree), we rename these selectors to a common name, thus conservatively merging the information of heap accesses read/write state once again. In the next section we present the procedure to build a static dependency graph from the data-flow analysis results.

4 Generating Static Dependencies

By considering an application as a set of programs that may be executed in parallel, and given the read-write analysis of the application, as described in Section 3.2, we compare the set of read/write states of each program with all other programs to create a Static Dependency Graph (SDG). This approach needs $\binom{n}{2} + n$ comparisons to build the graph. These is arguably a large number comparisons, specially if we consider a large number of concurrent programs. Since we don't know which programs will execute in parallel *a priori*, we conservatively assume that all programs are concurrent with each other and even with other instances of themselves. The complexity of the graph construction algorithm may be significantly reduced by using orthogonal techniques such as the May-Happen-in-Parallel analysis [3], to determine which programs may execute in parallel. Domain specific information provided by the developer may also be used to restrict the set of programs to be analyzed.

For all pairs of programs (P_i, P_j) of an application, we compare the two corresponding sets of read/write states resulting from the data-flow analysis—which are subsets of set Υ—and produce a set of program dependencies, thus defining a static dependency graph.

The kind of a dependency that exists between two programs is determined by two factors: the value of the read-/write states of data items in both programs and whether they can be executed concurrently. We say that two programs P_i and P_j are not concurrent if they both have a write state M for the same variable. By the *First-Commiter-Wins* rule, the execution of these two programs is always synchronized, and if they run in parallel one must abort.

There is a dependency relation between two programs if both access the same data item and at least one of those accesses is an update. Static dependencies are defined from the results of the analysis as follows:

Definition 5. *[Static Dependencies] For all programs P_i, P_j in an application, and all read/write states $U_i, U_j \subseteq \Upsilon$ where U_i is the read/write state of P_i and U_j is the read/write state of P_j.*

(Variable Dependencies) *If there is a variable x such that $(\_, w)_x \in U_i$ and $(r, \_)_x \in U_j$ where $w \neq ?$ and $r \neq ?$ then:*

1. *if $(\_, \alpha)_x \in U_i$ and $(\_, \beta)_x \in U_j$ where $\alpha \neq ?$ and $\beta \neq ?$ then $P_i \xrightarrow{x-ww} P_j$*

2. *if $(\_, \alpha)_x \in U_i$ and $(\beta, \_)_x \in U_j$ where $\alpha \neq ?$ and $\beta \neq ?$ then $P_i \xrightarrow{x-wr} P_j$*

3. *if $(\alpha, \_)_x \in U_i$ and $(\_, \beta)_x \in U_j$ where $\alpha \neq ?$ and $\beta \neq ?$, and P_i and P_j are not concurrent then $P_i \xrightarrow{x-rw} P_j$*

4. *if $(\alpha, \_)_x \in U_i$ and $(\_, \beta)_x \in U_j$ where $\alpha \neq ?$ and $\beta \neq ?$, and P_i and P_j are concurrent then $P_i \xRightarrow{x-rw} P_j$*

(Heap Access Dependencies) *If there is a heap access $a = (x, n_1, sel)$ and $b = (y, n_2, sel)$ such that $(\_, w)_a \in U_i$ and $(r, \_)_b \in U_j$ where $w \neq ?$ and $r \neq ?$ then:*

1. *if $(\_, \alpha)_{(x,n_1,sel)} \in U_i$ and $(\_, \beta)_{(y,n_2,sel)} \in U_j$ where $\alpha \neq ?$ and $\beta \neq ?$ then $P_i \xrightarrow{((x,n_1),(y,n_2))-ww} P_j$*

2. *if $(\_, \alpha)_{(x,n_1,sel)} \in U_i$ and $(\beta, \_)_{(y,n_2,sel)} \in U_j$ where $\alpha \neq ?$ and $\beta \neq ?$ then $P_i \xrightarrow{((x,n_1),(y,n_2))-wr} P_j$*

3. *if $(\alpha, \_)_{(x,n_1,sel)} \in U_i$ and $(\_, \beta)_{(y,n_2,sel)} \in U_j$ where $\alpha \neq ?$ and $\beta \neq ?$, and P_i and P_j are not concurrent then $P_i \xrightarrow{((x,n_1),(y,n_2))-rw} P_j$*

4. *if $(\alpha, \_)_{(x,n_1,sel)} \in U_i$ and $(\_, \beta)_{(y,n_2,sel)} \in U_j$ where $\alpha \neq ?$ and $\beta \neq ?$, and P_i and P_j are concurrent then $P_i \xRightarrow{((x,n_1),(y,n_2))-rw} P_j$*

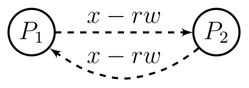

Figure 10: A SDG with a cycle of read-write dependencies for the same variable.

It is important to note that the comparison between programs P_1 and P_2 is a two-way procedure. Most of the times this comparison generates dependencies in both ways. For instance, if we consider programs P_1 with the analysis yielding $R_1 = \{(M, m)_x\}$ and P_2 with the analysis yielding $R_2 = \{(m, M)_x\}$, comparing them generates two dependencies: $P_1 \xrightarrow{wr} P_2$ since P_1 may write variable x that is being read by P_2; and $P_2 \xrightarrow{wr} P_1$ because P_2 writes variable x which may being read by P_1.

If we know that programs P_i and P_j are not concurrent, we generate non-vulnerable read-write dependencies.

4.1 Incompatible Variable Static Dependencies

Definition 5 generates pairs of static dependencies that cannot co-exist in a real execution, and although they are harmless, they will be detected as dangerous structures. For instance, consider the two programs P_1 and P_2 where $P_1 = P_2$ and where the analysis yields $\{(m, m)_x\}$. We obtain the simplified SDG of Figure 10.

According to definition of dangerous structures, the SDG for programs P_1 and P_2 contains a dangerous structure. However, no execution of these programs will trigger a SI anomaly. This is due to the existence of some pairs of edges that are incompatible with each other, thus we cannot blindly apply the algorithm to detect dangerous structures, as presented in Section 2.2, as we would find too many false positives.

When traversing the edges in the SDG depicted in Figure 10, one can observe that P_1 has an incoming vulnerable edge from P_2, has an outgoing vulnerable edge to P_2, and there is cyclic path from P_2 to itself. Now, consider a history with two committed transactions T_1 and T_2, resulting from the execution of P_1 and P_2 respectively. We claim that is not possible to define such history under Snapshot Isolation if there is a transactional dependency $T_1 \xrightarrow{x-rw} T_2$ and a transactional dependency $T_2 \xrightarrow{x-rw} T_1$. Consider that T_1 is executing concurrently with T_2, and there are dependencies $T_1 \xrightarrow{x-rw} T_2$ and $T_2 \xrightarrow{x-rw} T_1$. These dependencies mean that T_1 reads variable x and T_2 writes variable x, and that T_2 reads variable x and T_1 writes variable x. From the above one can infer that T_1 and T_2 are necessarily concurrent, that both write to variable x, and that both commit. However, by the *First-Committer-Wins* rule, one of the two transactions should have aborted, hence it is not possible to define such a history. This incompatibility between edges also applies to the other types of dependencies.

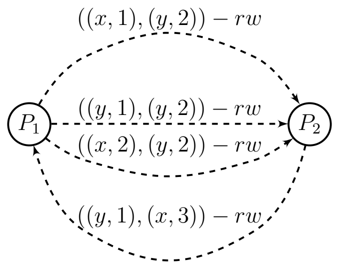

Figure 11: A SDG with a cycle of read-write dependencies for heap accesses.

Given these observations, if we apply the definition of dangerous structures to the SDG of Figure 10 and follow the edge $P_1 \xRightarrow{x-rw} P_2$, we can ignore the same type of edge for the same variable in the opposite direction, i.e., the edge $P_2 \xRightarrow{x-rw} P_1$.

4.2 Incompatible Heap Access Static Dependencies

The processing of static dependencies for heap accesses must also be enhanced, as there are pairs of heap access static dependencies that can not exist in a real execution. Consider an example with two programs $P_1 = \{(M, ?)_{(x,1,sel)}, (M, ?)_{(x,2,sel)}, (?, M)_{(x,3,sel)}, (M, ?)_{(y,1,sel)}\}$ and $P_2 = \{(M, ?)_{(y,1,sel)}, (M, M)_{(y,2,sel)}\}$. Figure 11 depicts the simplified SDG for these two programs, which resulted from their static dependencies. Looking at this SDG is easy to identify a dangerous structure between P_1 and P_2, but this is actually a false positive.

In this example the dependency $P_2 \xRightarrow{((y,1),(x,3))-rw} P_1$ is incompatible with the other three opposite dependencies. When we say that there is a static dependency $P_1 \xRightarrow{((x,1),(y,2))-rw} P_2$, it means that we are assuming that the heap cell accessed by P_1 with base variable x and distance 1, is the same heap cell accessed by P_2 with base variable y and distance 2. If the heap cell $(x, 1)$ in P_1, that from now on we will represent as $P_1(x, 1)$, is the same as $P_2(y, 2)$ than it is impossible that the heap cell $P_1(x, 3)$ be the same as the $P_2(y, 1)$, as cell $P_1(x, 3)$ comes after cell $P_1(x, 1)$ and cell $P_2(y, 1)$ comes before

$P_2(y, 2)$, and we are assuming only acyclic data structures. This can be illustrated in the following diagram:

$$\begin{array}{llc|c|cccc}
\cline{4-4} P_1 : x & & & 1 & \rightarrow & 2 & \rightarrow & 3 \\
P_2 : y & 1 & \rightarrow & 2 & & & & \\ \cline{4-4}
\end{array}$$

Also, if we assume $P_1(x, 2)$ to be the same as $P_2(y, 2)$, then we have the following diagram:

$$\begin{array}{llc|c|cc}
\cline{4-4} P_1 : x & 1 & \rightarrow & 2 & \rightarrow & 3 \\
P_2 : y & 1 & \rightarrow & 2 & & \\ \cline{4-4}
\end{array}$$

In the case of the dependency $P_1 \xRightarrow{((y,1),(y,2))-rw} P_2$, we are assuming that variable y in P_1 is the same as variable y in P_2, but the opposite dependency assume that variable y in P_2 is the same as variable x in P_1, and therefore they are incompatible.

If the dependency $P_2 \xRightarrow{((y,1),(x,3))-rw} P_1$ is incompatible with the other three opposite dependencies, then the SDG depicted in Figure 11 has no dangerous structures.

5 Dependency Detection Algorithm

The algorithm for detecting dangerous structures, described in Section 2.2, must be extended to ignore incompatible dependencies as defined in Sections 4.1 and 4.2. The pseudo-code in Algorithm 1 defines the new procedure for detecting dangerous structures in an SDG. It looks for cyclic paths in the graph with at least two consecutive vulnerable edges. This pattern identifies a dangerous structure in the SDG and signals the corresponding anomaly. The application of this algorithm to the SDG finalizes our static analysis procedure. The overal output is a set of possible SI anomalies and the corresponding set of anomalous programs.

The main difference between our algorithm and the algorithm described in Section 2.2 is that we keep the history of visited dependencies and check for their compatibility in the rest of the graph. Function $compatible$ tests whether an edge e is compatible with the trail of edges already visited. The pseudo-code in Algorithm 2 defines function $compatible$, where function $isHeapAccess$ asserts that an edge is a dependency on a heap access. Functions $varF$, $distF$, $varS$, and $distS$ extract components from heap access tuples of the form $((x, d_1), (y, d_2))$, their results are respectively x, d_1, y, and d_2. Function var denotes the variable of a given variable dependency in the graph, and function $type$ denotes the type of a given dependency.

Algorithm 1: Detection of Dangerous Structures

```
Input: nodes[], edges[], visited[]
Result: true or false
initialization;
foreach Node n : nodes do
    foreach Edge in : incoming(n, edges) do
        if vulnerable(in) then
            add(visited, in);
            foreach Edge out : outgoing(n, edges) do
                if vulnerable(out) $\wedge$ compatible(out,
                visited) then
                    add(visited, out);
                    if existsPath(target(out),
                    source(in), visited) then
                        return true;
                    end
                end
            end
        end
    end
    clear(visited);
end
return false;
```

6 Validation

In order to validate our approach, we developed a prototype using Polyglot [8] to analyze Java programs extended with software transactional memory. In our language, transactional methods are identified by an @Atomic annotation accepting two parameters. One parameter establishes the initial state of the heap, and the other associates a group name to that transactional method. All transactional methods annotated with the same group name are considered, by the developer, as concurrent and are analyzed together in the same static dependency graph.

We apply our analysis to an implementation of the SmallBank benchmark [1], originally used to evaluate SI database systems crafted to dynamically avoid anomalies. Along with the description of the benchmark, the authors also provide its SDG, identify a dangerous structure in the graph and corresponding anomaly. Our goal is to mechanically obtain the same results.

The benchmark contains five banking procedures [1]: Balance, Deposit Check, Transaction Saving, Amalgamate and Write Check. These procedures operate over three relations: Account(Name, CustomerId), Saving(CustomerId, Balance), and Checking(CustomerId, Balance). The Account relation

Algorithm 2: *compatible* function

```
Input: edge, visited[]
Result: true or false
foreach Edge v : visited do
    if source(edge) = target(v) ∧ target(edge) = source(v) then
        if isHeapAccess(edge) ∧ isHeapAccess(v) then
            if varF(edge) = varS(v) ∧ varS(edge) = varF(v) then
                if distF(edge) = distS(v) ∧ distS(edge) = distF(v) ∧ type(edge) = type(v) then
                    return false;
                else if (distF(edge) ≥ distS(v) ∧ distS(edge) ≤ distF(v)) ∨ (distF(edge) ≤ distS(v) ∧ distS(edge) ≥ distF(v)) then
                    return false;
                end
            else if varF(edge) = varS(e) ∧ varS(edge) ≠ varF(e) then
                return false;
            else if varF(edge) ≠ varS(e) ∧ varS(edge) = varF(e) then
                return false;
            end
        else if not (isHeapAccess(edge) ∨ isHeapAccess(v)) then
            if var(edge) = var(v) ∧ type(edge) = type(v) then
                return false;
            end
        else
            return false;
        end
    end
end
return true;
```

identifies the customers of the bank. The Balance attribute in both Saving and Checking relations quantifies the amount in each kind of account for each customer. In our implementation of the benchmark, we represent the three relations using an ordered singly linked list of triples, containing a name and a balance for each kind of account. Each banking procedure is implemented as a Java method, tagged with the @Atomic annotation, and operating over the triples in the list. We divide the transactional methods in two groups, one for the list operations and another for the five SmallBank benchmark operations.

$$\begin{aligned} add =&\{(M,?)_h, (M,?)_{(h,1,next)}, (m,m)_{(h,2,next)}\}, \\ remove =&\{(M,?)_h, (M,?)_{(h,1,next)}, (m,m)_{(h,2,next)}, \\ &(m,?)_{(h,3,next)}\} \\ get =&\{(M,?)_h, (M,?)_{(h,1,next)}, (m,?)_{(h,2,next)}\} \end{aligned}$$

Table 2: Analysis on the list group.

$$\begin{aligned} &add \overset{rw}{\Longrightarrow} remove \overset{rw}{\Longrightarrow} add \text{ (write-skew)} \\ &remove \overset{rw}{\Longrightarrow} remove \overset{rw}{\Longrightarrow} remove \text{ (write-skew)} \\ &get \overset{rw}{\Longrightarrow} add \overset{rw}{\Longrightarrow} remove \text{ (False Positive)} \\ &get \overset{rw}{\Longrightarrow} remove \overset{rw}{\Longrightarrow} add \text{ (False Positive)} \end{aligned}$$

Table 3: Anomalies

The list group contains three transactional methods: add, remove and get. The list nodes are stored in heap cells with selector $next$ pointing to the next node in the list, and selector val storing the node's value. The result of the data-flow analysis on this group, focused on the selector $next$, is presented in Table 2. The selector val is irrelevant for detecting anomalies. The SDG generated from the result of the data-flow analysis is depicted in Figure 12, where we can identify the dangerous structures described in Table 3. Two of the dangerous structures, involving the get method, are false positives.

The write-skew anomalies detected by our analysis, involving the transactions add and remove, correspond to the example introduced in Section 1. These anomalies can be corrected by forcing full serializability on the execution of add and remove methods, or by adding a dummy write operation on the node to be removed by the remove method.

The results of the analysis of the five transactional methods in the benchmark group are presented in Table 4. The SDG generated from the result of the data-flow analysis is depicted in Figure 13. The only dangerous structure identified is $Bal \overset{rw}{\Longrightarrow} WC \overset{rw}{\Longrightarrow} TS$, where WC is the *pivot*. This dangerous structure is the one originally referred in the benchmark [1].

Our prototype is able to mechanically identify all the expected anomalies in this sample code, but for the first time through a semantic analysis, and targeting the challenging setting of transactional memory programs written in a general purpose language.

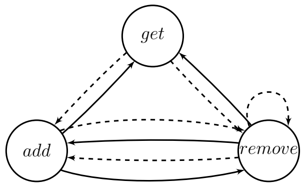

Figure 12: SDG of an ordered linked list.

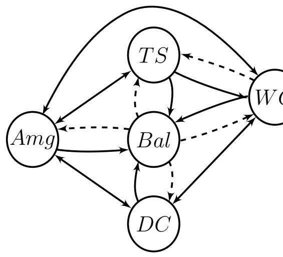

Figure 13: SDG of the SmallBank benchmark.

$$\begin{aligned}
Bal =& \{(M,?)_{(a,1,chec)}, (M,?)_{(a,1,savi)}\} \\
DC =& \{(M,M)_{(a,1,chec)}\} \\
TS =& \{(M,M)_{(a,1,savi)}\} \\
Amg =& \{(M,M)_{(a1,1,chec)}, (M,M)_{(a1,1,savi)}, (M,M)_{(a2,1,chec)}\} \\
WC =& \{(M,M)_{(a,1,chec)}, (M,?)_{(a,1,savi)}\}
\end{aligned}$$

Table 4: Analysis on the benchmark group.

7 Related Work

Software Transactional Memory (STM) [11, 5] systems commonly implement full serializability to ensure the correct execution of transactional memory programs. To the best of our knowledge, SI-STM [9] is the only implementation of a STM using Snapshot Isolation. Their work focuses on the improvement of transactional processing throughput by using a snapshot isolation algorithm. They also propose a SI safe variant of the algorithm where anomalies are automatic and dynamically avoided by enforcing validation of read/write conflicts. In our work, we aim at providing the serializability semantics under snapshot isolation STM systems. This is achieved by performing a static analysis to assert that no SI anomalies will occur when executing a transactional application.

The use of Snapshot Isolation in databases is a common place, and there is some previous works on the detection of SI anomalies in this domain. Our work is clearly inspired in [4], which presents the theory of SI anomaly detection and 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 [12]. A sequel of that work [6], 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. They also discuss some solutions to reduce the number of false positives produced by their analysis. Although targeting similar results, our work deals with significantly different problems. The more significant one is related to the full power of general purpose languages and the use of dynamically allocated heap data structures. Besides that, we treat the full control-flow of imperative programs. Our work strives for reducing the state explosion during the analysis by conservatively merging some kinds of information.

8 Concluding Remarks

In this paper we define a new static analysis technique and present the corresponding algorithm to mechanically assert that a given transactional memory application executing under Snapshot Isolation will not suffer from SI anomalies. Our work allows for the automatic identification of anomalous transactional code, by signaling conflicting programs and allowing for explicit code corrections. Thus, we enable the use of SI as the default isolation level for STM applications.

A major difference of our analysis to existing related work [4, 6] is that, instead of targeting languages with limited expressiveness like SQL, we target a general purpose imperative language with support for dynamically allocated data structures. We define a data-flow analysis that extracts fine grained read/write state information for each state variable and heap cell manipulated by transactional programs. We adapt a standard shape analysis technique [10] to track heap cell manipulation during the analysis.

The results of our data-flow analysis are then used to build dependency graphs on programs from which we can detect connection patterns that correspond to SI anomalies. Our algorithm may be used in the future as input for manual or automatic procedures of anomalous code correction.

To validate our approach, we implemented a prototype of the analysis framework capable of identifying the same anomalies in complete Java programs as related benchmarks detect in SQL procedures [1]. We also show that our analysis identifies known anomalies (write-skew) in simple acyclic dynamic data structures and illustrate the procedure on an ordered singly linked list implementation. We also propose an enhancement to the algorithm that uses a compatibility relation between dependencies to reduce the number of false positives introduced by the static analysis. We foresee that other techniques, such as using information from the type system, can also be used to further reduce the presence of false-positives.

Looking at the short term evolution of our framework, we plan to extend the shape graph model to include data structures with backward pointers, and enable the automatic correction of SI anomalies by source code transformation. In the long term, we aim at extending the correctness guarantees provided by our analysis to support further studies of SI in the STM setting, such as with distributed transactional memory.

References

[1] M. Alomari, M. Cahill, A. Fekete, and U. Röhm. The cost of serializability on platforms that use snapshot isolation. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 576–585, Washington, DC, USA, 2008. IEEE Computer Society.

[2] 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.

[3] E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV4: Proceedings of the symposium on testing, analysis, and verification*, pages 36–48, New York, NY, USA, 1991. ACM.

[4] 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.

[5] 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.

[6] 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.

[7] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[8] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. Technical Report TR2002-1883, Cornell University, Nov. 2002.

[9] 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.

[10] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.

[11] 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.

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