Practical Verification of High-Level Dataraces in Transactional Memory Programs

Vasco Pessanha[1]
v.pessanha@fct.unl.pt

Ricardo J. Dias[1]
emailrjfd@di.fct.unl.pt

João M. Lourenço[1]
joao.lourenco@di.fct.unl.pt

Eitan Farchi[2]
farchi@il.ibm.com

Diogo Sousa[1]
dm.sousa@fct.unl.pt

[1]CITI and DI FCT Universidade Nova de Lisboa, Portugal
[2]IBM Haifa Research Laboratory, Haifa, Israel

ABSTRACT

In this paper we present MoTh, a tool that uses static analysis to enable the automatic verification of concurrency anomalies in Transactional Memory Java programs. Currently MoTh detects high-level dataraces and stale-value errors, but it is extendable by plugging-in *sensors*, each *sensor* implementing an anomaly detecting algorithm. We validate and benchmark MoTh by applying it to a set of well known concurrent buggy programs and by close comparison of the results with other similar tools. The results achieved so far are very promising, yielding good accuracy while triggering only a very limited number of false warnings.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Experimentation, Languages, Reliability, Verification

Keywords

Static Analysis, Testing, Verification, Concurrency, Software Transactional Memory

1. INTRODUCTION

Transactional Memory [10, 14] (TM) is an approach to concurrent programming that applies the concept of transaction, widely known from the databases community, to the management of data in memory. TM promises both, a more efficient usage of parallelism and a more powerful semantics for constraining concurrency.

While TM may ease the development of concurrent programs and reduce the number of concurrency errors, its usage does not imply by itself the correctness of programs. In lock-based programs, the absence or the wrong usage of a synchronization mechanism to protect a critical region in one thread, allows for invalid accesses from other threads, causing a synchronization error (datarace). In the case of TM, critical regions are encapsulated in transactions, and the failure to do so appropriately may also trigger a synchronization error [4, 12]. This anomaly in TM is analogous to the classical datarace anomaly, and is herein referred to as *low-level datarace*.

A program that is free from low-level dataraces is guaranteed not to have corrupted data data structures, and the values of all variables always correspond to a specific serial execution of all synchronized (with locks or transactions) code blocks. However, experience shows that in many programs this guarantee does not suffice to ensure a correct execution and, although data is not corrupted, no assumptions can be made about its consistency.

Data consistency is violated when two synchronized blocks have a non-synchronized (possibly empty) code block between them, and the programer intended to have those two synchronized code blocks running atomically, but mistakenly believed it was sufficient to ensure their individual atomicity. This anomaly is often referred as a *high-level datarace*.

To illustrate this situation, consider the code fragment present in Figure 1, showing a shared pair of variables that should be accessed atomically. When a thread wants to check the equality of the pair's elements, it reads both elements in separate synchronized blocks storing their values in local variables, and then compares them. However, due to interleaving with another thread running the method setPair() between the executions of getX() and getY(), the value of the pair could have changed. In this scenario the first thread observes an inconsistent pair, composed by the old value of x and the new value of y.

Besides the high-level dataraces, other high-level anomalies can occur in a program, frequently called *stale-values errors* or *atomicity violations*. A stale-value [5] is a variable replica that no longer reflects the true value of that variable, i.e., it is outdated.

```
synchronized void getX() {
  return pair.x;
}
synchronized void getY() {
  return pair.y;
}
synchronized void setPair(int x, int y){
  pair.x = x;
  pair.y = y;
}
boolean checkEqual(){
  int x = getX();
  int y = getY();
  return x == y;
}
```

Figure 1: Example of a high-level datarace

Figure 2 illustrates a case of a stale-value anomaly. In this example, a set of threads try to increment a shared variable x. First, the value of the variable is read and stored is the local variable tmp. Then, this value is incremented and stored back in x. However, if between the two synchronized code blocks another thread running the same code updates the value of x, the two threads would update the variable to the same value, resulting in an lost update.

For the sake of simplicity and unless stated otherwise, we use the term *datarace* to refer to both high-level dataraces and stale-value errors.

In this paper we present $\mathcal{M}$oTh, a tool that uses static analysis to automatically verify the existence of dataraces in transactional memory Java programs. The tool is extensible in that it supports different datarace detector algorithms as plugins to the system. We call these plugins *sensors*. All sensors detect dataraces over the same knowledge base, made up by the sets of memory accesses occurring in the atomic code blocks, extending the notion of *views* from [1].

We evaluate the precision of $\mathcal{M}$oTh by running a series of tests taken from relevant literature, and comparing our results with those from other related works.

The contributions of this paper include an extension of the notion of view consistency [1], to incorporate both a distinction between read and write accesses and a partial order relation between accesses to the same variable; and an extensible infrastructure for static analysis of Java Bytecode programs, together with the necessary plugins (sensors) to detect high-level dataraces and stale-value errors in Transactional Memory Java programs.

The remainder of this paper is organized as follows. We describe the theoretical framework used in our tool in Section 2. In Section 3 we discuss some of the main challenges that were addressed when implementing the framework, followed by the analysis of the effectiveness of $\mathcal{M}$oTh in Section 4. Finally, we discuss the related work in Section 5, and the conclusions and future work in the last section.

2. THE MOTH TOOL

$\mathcal{M}$oTh is a tool that statically analyzes the Bytecode of a Java program and checks for the presence of dataraces (both high-level dataraces and stale-value errors).

The tool workflow includes two main phases. First, it perform a symbolic execution to compute a set of *views* corresponding to the set of transactions present in the program,

```
synchronized void read () {
  return x;
}
synchronized void update (int value) {
  x = value;
}
void inc () {
  int tmp = read ();
  //tmp can be outdated
  update (tmp +1);
}
```

Figure 2: Example of a stale-value error

where each view reflects the memory accesses made within a memory transaction. In the second phase, it uses the computed *views* as input to the *sensors*, a set plugins that implement different algorithms for the detection of high-level dataraces, thus called *High-Level Datarace Sensors*. These phases are further discussed in the following sections.

2.1 Extended Views

A *view*, as described by Artho et al. in [1], expresses what variables are accessed inside a given synchronized code block. Since we want to distinguish between read and write accesses, we need to express the memory access operations rather than just in the accessed variables. We then add a partial order relation between all memory accesses to the same variables, which is used later by the High-level Datarace Sensors, as defined in Section 2.3. Since we are working in an object-oriented paradigm, likewise [1], we generalize the concept of shared variables to fields of object instances.

Let C be the set of classes used in a Java program, and let F be the set of all fields of all classes in C. Moreover, let $\mathsf{Vars} \subseteq C \times F$ be the set of variables of that program, represented by the composition of their main class' name and the field's name.

Let $\mathcal{A}$ be the set of all read and write accesses to variables of Vars inside a synchronized block. Notice that, in the specific context of TM programs, a synchronized block corresponds to a transactional block. An access $a \in \mathcal{A}$ is a triple (α, v, b) where $\alpha \in \{r, w\}$, $v \in \mathsf{Vars}$ and $b \in \{\circ, \bullet\}$. α represents the type of access (r-read or w-write), and v represents the variable being accessed. b helps keeping a *use-define* relation for each accessed variable in a transaction. In a read access, b keeps information of whether the value read will ($\bullet$) or will not ($\circ$) be latter overwritten inside this same block. On the other hand, in a write access, b keeps the information of whether the written variable was ($\bullet$) or was not ($\circ$) read before in this same block.

A *View* $v \subseteq \mathcal{A}$ of a synchronized block is a subset of $\mathcal{A}$, and includes all the variable accesses made inside that block. The set of all views is denoted by Views.

Assuming that each view has an unique identifier that, for the sake of simplicity, is not represented in this formalization, we define the following function that returns the synchronized block of a given view:

$$\Gamma : \mathsf{Views} \longrightarrow \mathit{SynchronizedBlock}$$

The set of *generated views* $V(t)$ of a thread t is the set of views of each synchronized block executed by t:

$$v \in V(t) \Leftrightarrow sb = \Gamma(v) \wedge \mathsf{executes}(t, sb)$$

$$
\begin{array}{lcll}
e & ::= & & (\textit{expression}) \\
 & & x & (\text{variables}) \\
 & | & n & (\text{constant}) \\
 & | & e \oplus e & (\text{binary op}) \\
 & | & \mathsf{null} & (\text{null value}) \\
 & & & \\
A & ::= & & (\textit{assignments}) \\
 & & x := e & (\text{local}) \\
 & | & x := y.f & (\text{heap read}) \\
 & | & x.f := e & (\text{heap write}) \\
 & & & \\
S & ::= & & (\textit{statements}) \\
 & & S\,;S & (\text{sequence}) \\
 & | & A & (\text{assignment}) \\
 & | & proc(\vec{x}) & (\text{procedure call}) \\
 & | & \mathsf{if}\, e \,\mathsf{then}\, S \,\mathsf{else}\, S & (\text{conditional}) \\
 & | & \mathsf{while}\, e \,\mathsf{do}\, S & (\text{loop}) \\
 & | & \mathsf{skip} & (\text{skip})
\end{array}
$$

Figure 3: Small imperative language syntax

Finally, since we want to distinguish read from write accesses, for each view we generate its Read View (V_r) and its Write View (V_w). Therefore, for $\alpha \in \{r, w\}$, we get:

$$V_\alpha(t) \triangleq \{(\alpha, v, b) | (\alpha, v, b) \in V(t)\}$$

2.2 Symbolic Execution

We now present the symbolic execution rules that generate the *views* of each thread. We defined a simple imperative language where the syntax of each procedure is shown in Figure 3. In this language, variables may be either integer values, or memory pointers. We encode boolean values using integer values 0 and 1 denoting false and true respectively. We assume that variables point to objects that are already allocated in the heap. We only create views for transactional methods, and the memory allocation operation is not relevant for this matter. In the scope of this section we assume that each procedure corresponds to a transactional method, and thus, we create a *view* for each procedure.

Figure 4 lists the operational symbolic execution rules of each statement in the syntax of the language. The symbolic execution of each procedure always starts with an empty *view*. A *view* is a set of memory accesses, and thus, whenever a read or write access is made to an object's field, we need to add that access to the current *view*.

The auxiliary function add, defined below and used in the symbolic execution rules Heap Read and Heap Write, adds an access of the form (α, v, γ) to the current *view* $\mathcal{V}$, allowing to compute the special *use-define* relation kept for each memory access. This *use-define* relation is a key element in the definition of the *Single Variable Sensor* described in Section 2.3.2.

Definition 1 (Add Access). *Inserts a memory access into the view $\mathcal{V}$.*

$$\mathsf{add} : \mathcal{A} \times \mathsf{Views} \to \mathsf{Views}$$

$$
\mathsf{add}((\alpha, v, \gamma), \mathcal{V}) \triangleq
\begin{cases}
\mathcal{V} \setminus \{(\alpha, v, \delta)\} \cup \{(\alpha, v, \gamma)\} & \text{if } (\alpha, v, \delta) \in \mathcal{V} \wedge \alpha = r \wedge \gamma \neq \delta \\
\mathcal{V} \setminus \{(r, v, \delta)\} \cup \{(r, v, \bullet)\} & \\
\quad \cup \{(\alpha, v, \bullet)\} & \text{if } (r, v, \delta) \in \mathcal{V} \wedge \alpha = w \\
\mathcal{V} \cup \{(\alpha, v, \gamma)\} & \text{otherwise}
\end{cases}
$$

$$\boxed{\langle \mathcal{V}, S \rangle \Longrightarrow \langle \mathcal{V}' \rangle}$$

$$\frac{\langle \mathcal{V}, S_1 \rangle \Longrightarrow \langle \mathcal{V}' \rangle \qquad \langle \mathcal{V}', S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle}{\langle \mathcal{V}, S_1; S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle}\,(\textsc{Seq})$$

$$\frac{}{\langle \mathcal{V}, x := y \rangle \Longrightarrow \langle \mathcal{V} \rangle}\,(\textsc{Assign})$$

$$\frac{c = \mathsf{typeof}(y) \qquad \mathcal{V}' = \mathsf{add}((r, (c, f), \circ), \mathcal{V})}{\langle \mathcal{V}, x := y.f \rangle \Longrightarrow \langle \mathcal{V}' \rangle}\,(\textsc{Heap Read})$$

$$\frac{c = \mathsf{typeof}(x) \qquad \mathcal{V}' = \mathsf{add}((w, (c, f), \circ), \mathcal{V})}{\langle \mathcal{V}, x.f := y \rangle \Longrightarrow \langle \mathcal{V}' \rangle}\,(\textsc{Heap Write})$$

$$\frac{\mathsf{spec}(proc) = \mathcal{V}_p \qquad \mathcal{V}' = \mathsf{merge}_p(\mathcal{V}_p, \mathcal{V})}{\langle \mathcal{V}, proc(\vec{x}) \rangle \Longrightarrow \langle \mathcal{V}' \rangle}\,(\textsc{Proc Call})$$

$$\frac{\langle \mathcal{V}, S_1 \rangle \Longrightarrow \langle \mathcal{V}' \rangle \qquad \langle \mathcal{V}, S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle}{\langle \mathcal{V}, \mathsf{if}\, e \,\mathsf{then}\, S_1 \,\mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{V}' \cup \mathcal{V}'' \rangle}\,(\textsc{Conditional})$$

$$\frac{\langle \mathcal{V}, S \rangle \Longrightarrow \langle \mathcal{V}' \rangle}{\langle \mathcal{V}, \mathsf{while}\, e \,\mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{V} \cup \mathcal{V}' \rangle}\,(\textsc{Loop})$$

$$\frac{}{\langle \mathcal{V}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{V} \rangle}\,(\textsc{Skip})$$

Figure 4: Operational Symbolic Execution Rules

The first branch of this function denotes the case where we want to add a read access of variable v with read state γ, and already exists a read access for the same variable with the opposite read state δ in the current *view*. In this case we replace the existing read access with the new one.

The second branch of this function denotes the case where we want to add a write access, and there is an existing read access in the current *view*. In this case we always need to update the existing read access state to $\bullet$, which means that its value was overwritten. We also need to add the write access with a state $\bullet$ as well, meaning there was a read access to the same variable in the current *view* previous to this write.

The third branch denotes all the cases where the conditions for the two first branches do not hold. In this cases we just add the new access to the current *view*.

In the case of the Proc Call rule, we first analyze the procedure being called and generate the corresponding *view*, which is then merged into the current *view*. Given a procedure identifier *proc*, the function spec retrieves its computed view $\mathcal{V}_p$. We then make use of function merge_p to join the result of *view* $\mathcal{V}_p$ and the current *view* $\mathcal{V}$. We define the function merge_p as:

Definition 2 (Function Merge). *Let $\mathcal{V}_p$ be the view retrieved from the specification of a procedure proc. Function* merge_p *merges the view $\mathcal{V}_p$ with the caller method's view $\mathcal{V}$.*

Please notice that all the write accesses in $\mathcal{V}_p$ are merged into $\mathcal{V}$ before merging the first read access.

$$\mathsf{merge}_p : \mathsf{Views} \times \mathsf{Views} \rightarrow \mathsf{Views}$$

$$\begin{aligned}\mathsf{merge}_p(\mathcal{V}_p, \mathcal{V}) = \mathcal{V}'' \text{ where}&\\ \mathcal{V}' &= \mathsf{merge}(\{(w, v, \gamma) \mid (w, v, \gamma) \in \mathcal{V}_p\}, \mathcal{V})\\ \wedge\, \mathcal{V}'' &= \mathsf{merge}(\{(r, v, \gamma) \mid (r, v, \gamma) \in \mathcal{V}_p\}, \mathcal{V}')\end{aligned}$$

$$\mathsf{merge}(\mathcal{V}_p, \mathcal{V}) \triangleq \begin{cases}\mathcal{V} & \text{if } \mathcal{V}_p = \emptyset\\ \mathsf{merge}(\mathcal{V}_p \setminus \{a\}, \mathsf{add}(a, \mathcal{V})) & \text{if } \exists a \in \mathcal{V}_p\end{cases}$$

The function merge_p inserts the accesses present in *view* $\mathcal{V}_p$ one by one into the current *view* $\mathcal{V}$. It first insert all the write accesses and then the read accesses. It is important to highlight that *view* $\mathcal{V}_p$ is computed independently from $\mathcal{V}$, the *view* of the calling context, making our symbolic execution more compositional. In rule PROC CALL, after getting *view* $\mathcal{V}_p$ from the callee, we merge the two *views* using the merge_p function.

We believe all the remaining symbolic execution rules are self explanatory.

In the end of the symbolic execution we have computed the common knowledge base, in the form of extended *views*, that is used by the *sensors* for detecting multiple kinds of dataraces.

2.3 Datarace Detection — HLDR Sensors

The detection of dataraces using MoTH is achieved by adding plugins to the tool. Each plugin detects a specific kind of dataraces, and we call *HLDR sensors* to those plugins aiming at the detection of High-level Dataraces. The set of conflicts detected by the Sensors are not necessarily disjoint, since all these sets are merged before presented to the user.

Each Sensor implements an algorithm aiming at detecting one specific type of dataraces. These algorithms are completely independent from each other and can be executed in parallel, improving considerably the efficiency of the tool.

So far, we have defined two types of Sensors that detect the majority of dataraces found in the literature. However, our tool is completely extensible: if a datarace is not detected by our current plugins, a new Sensor targeting this anomaly can be designed, implemented, and integrated into MoTH as a plugin.

2.3.1 View Consistency Sensor

This first Sensor is intended to detect every datarace related to partial accesses to atomic sets of variables. If two variables are accessed atomically by a thread t_1, but another thread t_2 accesses them separately, then we have a high-level datarace. The algorithm described below is an extension of the *view consistency* concept described in [1], which incorporates a distinction between read and write memory accesses and yields less false positives.

Maximal Views are the views that are not subsets of other views of the same thread, and represent the set of variables that should be accessed atomically. Therefore, partial accesses of these sets can generate a datarace. Similarly to the view generation procedure, we also distinguish Read Maximal Views (M_r) from Write Maximal Views (M_w). Therefore, for $\alpha \in \{r, w\}$, we get:

$$v_m \in M_\alpha(t) \Leftrightarrow v_m \in V_\alpha(t) \wedge (\forall v \in V_\alpha(t) : v_m \subseteq v \Rightarrow v = v_m)$$

Given a set of views of a thread t, and another thread's maximal view v_m, the read and write *overlapping views* of t with v_m are all non-empty intersections of views in $V(t)$ with v_m. Therefore, for $\alpha \in \{r, w\}$, we get:

$$\mathsf{overlap}_\alpha(t, v_m) \triangleq \{v_m \cap v \mid (v \in V_\alpha(t)) \wedge (v_m \cap v \neq \emptyset)\}$$

A set of views of a thread t is read/write compatible with a maximal view v_m of another thread, if and only if every read/write *overlapping views* of t with v_m form a chain. With this, for $\alpha \in \{r, w\}$, we have:

$$\mathsf{comp}_\alpha(t, v_m) \Leftrightarrow \forall v_1, v_2 \in \mathsf{overlap}_\alpha(t, v_m) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$$

Notice that, intentionally, nothing is said in the last two definitions about whether v_m is a read or write maximal view, as they apply to both cases.

Finally, the concept of *view consistency* is defined as the mutual compatibility of all threads. A thread can only have views that are compatible with all maximal views of all other threads. However, we exclude possible dataraces generated exclusively by read accesses. Therefore, we define the following property that has to be verified in order to guarantee the absence of this kind of dataraces:

PROPERTY 1 (VIEW SAFETY).

$$\begin{gathered}\forall t_1 \neq t_2, m_r \in M_r(t_1), m_w \in M_w(t_1) :\\ \mathsf{comp_w}(\mathsf{t_2}, \mathsf{m_r}) \wedge \mathsf{comp_r}(\mathsf{t_2}, \mathsf{m_w}) \wedge \mathsf{comp_w}(\mathsf{t_2}, \mathsf{m_w})\end{gathered}$$

If this property is not verified, then this sensor will yield a warning identifying a high-level datarace occurrence.

Despite the fact that this algorithm detects the majority of the dataraces presented in the literature, like the simpler version of [1], it is neither sound nor complete, and may yield both false positives and false negatives. Thus, it is still possible to have programs with dataraces undetected by this sensor, such as stale-value errors.

2.3.2 Single Variable Sensor

Even when extended with read and write accesses distinction, the concept of view consistency does not detect some other kinds of anomalies, such as stale-value errors. Therefore, in order to create a more complete and effective tool, we complement this algorithm with others that fill this and possibly other gaps.

As an example of a stale-value error, imagine that a thread checks the value of a variable $var \in \mathsf{Vars}$ in a transaction t_1 and then, based on that previously observed state, alters var in a different transaction t_2. The value of var could have changed in between, generating a lost update.

This kind of dataraces, usually referred as *atomicity violations* or *stale-values errors*, were also addressed by earlier approaches [2, 5, 7, 9, 15, 17, 18]. Our algorithm is based on Teixeira's RwW pattern [15].

A variable $var \in \mathsf{Vars}$ has a possible stale-value, if in a given thread t it is read in a synchronized block (without being overridden in that block), and is then written in another subsequent synchronized block.

$$\begin{aligned}psv(var, t) \Leftrightarrow \exists v_1 \neq v_2 \in V(t) :&\\ (r, var, \circ) \in v_1 \wedge (w, var, \delta) \in v_2&\\ \text{where } \delta \in \{\circ, \bullet\}&\end{aligned}$$

If this is the case, then the value of the shared variable can possibly escape from one synchronized block to another.

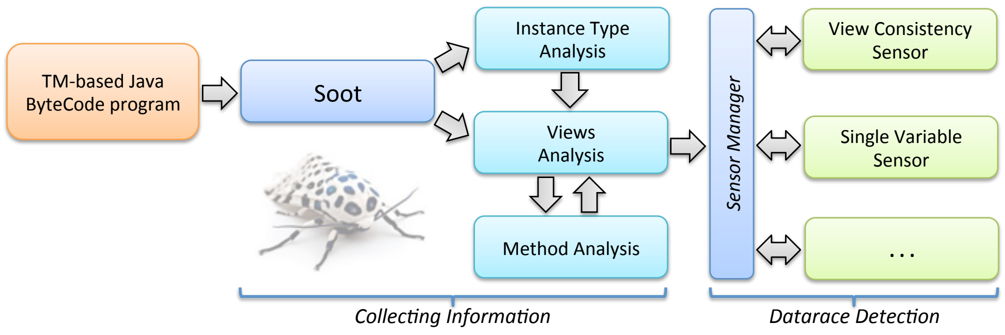

Figure 5: Datarace Detection Procedure

The partial order of the read and write accesses in a view, introduced by the *use-define* relation based in the annotation $\{\circ, \bullet\}$, allows to assume that the shared variable escaped the first block, since it was not overwritten in that block.

A thread t writes in a variable $var \in \mathsf{Vars}$ if there is a write access to that variable in a write view of that thread:

$$writes(var, t) \Leftrightarrow \exists v \in V_w(t) : (w, var, \delta) \in v \quad \text{where } \delta \in \{\circ, \bullet\}$$

Two threads verify the partial safety property if the first does not write in any variable that has a possible stale-value in the second thread:

$$pSafe(t_1, t_2) \Leftrightarrow \forall var \in \mathsf{Vars} : \neg writes(var, t_1) \vee \neg psv(var, t_2)$$

Finally, a given program is free of this type of dataraces if and only if all its threads are safe between themselves, i.e., it verifies the following property:

PROPERTY 2 (SINGLE VARIABLE SAFETY).

$$\forall t_1 \neq t_2 : pSafe(t_1, t_2) \wedge pSafe(t_2, t_1)$$

3. IMPLEMENTATION

The algorithms presented in Section 2 were used to statically identify dataraces in transactional memory programs written in Java.

We resorted to the Soot [13], one of the larger and more mature analysis frameworks for Java Bytecode, to implement our own Java Bytecode analyzer. Soot is an open source framework used for code optimizations, transformations and analysis. It analyzes the programs' bytecode and offers an intermediate language between Java source and bytecode, called Jimple. Our tool uses the generated Jimple code to collect all the useful information described in Section 2.1.

During the implementation of our algorithm, we encountered some challenges when analyzing some classes, which we discuss in the remainder of this Section.

Since we can not analyze native methods, we assume the worst case scenario. Therefore, when we don't have access to a method's body, we assume that it reads and writes in all its parameters, and also reads and writes the object in which it was called. This strategy may increase the number of false positives reported by the framework, but avoids yielding false negatives.

Another challenge we had to face was related to the concept of Dynamic Dispatch, i.e., the process of mapping a method call to a specific sequence of code. Which code should we analyze when a method is called in an object that has an interface type, and can be initialized with more than one class?

To address this issue, we start by looking to all initialization statements, collecting every possible implementing class for each variable. Then, for every variable $v \in \mathsf{Vars}$ with multiple implementing classes, the view of a method called in v is obtained as the union of views of that method in all its implementing classes.

Finally, if we have no access to the method's body nor information about its implementing classes, we treat it as a native method and assume that it reads and writes in all its parameters and also in the object in which it was called.

The approaches just described always assume the worst case scenario and may yield a large number of false positives. To minimize this effect, an annotation mechanism was implemented. The annotation mechanism allows the user to easily state in which parameters a method writes/reads, and if it writes/reads the object instance where that method was called. Therefore, when a method is assumed to be native, $\mathcal{M}$oTH automatically generates annotations for that method assuming the worst case scenario. The user is then alerted and allowed to revise the annotation, which will be considered in future runs of the tool.

Figure 5 illustrates the workflow just described. Firstly, a program's Java Bytecode is transformed to *jimple code* by the Soot framework. We then analyze this code in order to determine all implementing classes for each variable (*Instance Type Analysis*), and compute the views of each thread (*Views Analysis*). In this analysis, some method annotations are used to determine which read and write accesses are made in some specific methods (*Method Analysis*). Moreover, some native methods are automatically annotated in this process with the user's consent. Finally, all the information is provided to the Sensors for anomaly detection.

4. EXPERIMENTAL VALIDATION

To validate our approach we run a series of tests taken from the literature, already used in the past to illustrate dataraces in concurrent programs and/or to validate other related works. We had access to Teixeira's tool [15], and have implemented Artho's algorithm [1] in $\mathcal{M}$oTH (using static analysis instead of the dynamic approach followed in [1]). Since these two works are the basis for our work, we always report on their results as well. From the empiric comparison of all the results, we can have a deeper insight of the

```
public void swap() {
  int oldX;
  atomic{
    oldX = coord.x;
    coord.x = coord.y;
    coord.y = oldX;
  }
}
public void reset(){
  atomic{
    coord.x = 0;
  }// inconsistent state (0, y)
  atomic{
    coord.y = 0;
  }
}
```

Figure 6: Code snippet for Coordinates test

relevance and added analysis precision introduced by our approach in MoTH.

In Section 4.1, we present the set of tests used to benchmark MoTH, discussing in detail some of them. The results of the benchmarking process are summarized in Section 4.2. Finally, in Section 4.3, we discuss the limitations of our approach that motivate our future work.

4.1 Tests Description/Sources

The universe for the validation and benchmarking of our approach was composed by 15 tests, all taken from the literature [1,2,3,15,17] except for the *Allocate Vector* test, which was taken from the IBM concurrency benchmark repository [11]. All the tests but this last one were also used in the past to validate Teixeira's approach, and are described in detail in [15]. We discuss in detail two of these tests that are representative of the set. We start by discussing the *Coordinates* test, which includes a common error where a global update is divided in two transactions. Then, we present the *Account* test, which has a datarace with just one single variable.

4.1.1 Coordinates Test

This test was adapted from [2] and is represented in Figure 6. In this test, the pair of coordinates coord.x and coord.y is meant to always be accessed atomically. There are two possible operations, swap() that exchanges the values of both coordinates in a single transaction, and reset() that mistakenly resets their values in two distinct transactions.

When a thread t_1 runs swap() between the two atomic blocks of the reset() method, which is being executed by another thread t_2, then it results in an inconsistent final state, since both reset operations are made to the same coordinate leaving the other intact, which was not the intended behavior.

All the views of threads t_1 and t_2 are represented in Table 1. Both write views of thread t_2 ($\{x\}$ and $\{y\}$) have non-empty intersections with t_1's read and write maximal view ($\{x, y\}$), and do not form a chain. Thus, the high-level datarace is correctly detected by the *View Consistency Sensor*, without generating false positives.

4.1.2 Account Test

This test was adapted from [17], and is presented in Figure 7. In this test, the balance of a bank account can be accessed and updated by more than one thread. The update operation is composed by two atomic sub-operations, read() and sumTmp(), that read and update the account's balance. However, between reading and updating the account's balance, its value might have changed due to the interleaving with another thread running the same code.

This situation could generate a lost update situation, representing a datarace generated by accesses to a single variable, thus not detectable with the concept of view consistency. Our algorithm has successfully detected this datarace with the *Single Variable Sensor* described Section in 2.3.2.

Table 1: Coordinates Views

	t_1 (swap)	t_2 (reset)
V_r	$\{x,y\}$	-
V_w	$\{x,y\}$	$\{x\},\{y\}$
M_r	$\{x,y\}$	-
M_w	$\{x,y\}$	$\{x\},\{y\}$

```
int balance;
void update (int a) {
  int tmp = read();
  //tmp can have a obsolete value
  sumTmp(tmp,a);
}
@Atomic
int read () {
  return balance;
}
@Atomic
private void sumTmp(int tmp,int a){
  balance = tmp + a;
}
```

Figure 7: Code snippet for Account test

4.2 Results

According to the classification of *soundness* and *completeness* by Flanagan et al. [8] and likewise Artho's approach, our approach is both unsound and incomplete, i.e., it can yield both false positives and negatives. In a total of 15 dataraces present in these programs, 13 were correctly pointed out (87% of total dataraces). Moreover, only 6 false positives were yielded, all by the *Single Variable Sensor*.

The majority of the warnings raised by our tool correspond to real (potential) conflicts that cannot occur in the execution of this specific program, and could be excluded with a *may-happens-in-parallel* analysis. As an example, consider a thread that execute two atomic methods, one that reads and another that writes the value of a variable *var*. One could think that, if it first reads *var* in one transaction, and then based on this result it updates *var* in a different one, we would have a datarace occurrence. However, if the thread always executes the write before the read operation, its executions are datarace free. In this scenario, despite the fact that these methods can generate a datarace, no warnings should be yield since it does not occur in this specific program execution.

Some other warnings could be excluded refer to dataraces that are related to accesses made to different instances of the same class. We are currently unable to distinguished

Table 2: Test results summary

Test Name	Known Anomalies	False Negatives			False Positives		
		$\mathcal{M}$oTh	Artho [1]	Teixeira [15]	$\mathcal{M}$oTh	Artho [1]	Teixeira [15]
Connection [3]	2	1	1	1	1	0	1
Coord03 [1]	1	0	0	0	0	0	3
Local [1]	1	0	1	0	0	0	1
NASA [1]	1	0	0	0	0	0	0
Coord04 [2]	1	0	0	0	0	0	3
Buffer [2]	0	0	0	0	1	0	7
DoubleCheck [2]	0	0	0	0	1	0	2
StringBuffer [7]	1	1	1	1	0	0	0
Account [17]	1	0	1	0	0	0	0
Jigsaw [17]	1	0	0	0	2	0	1
OverReporting [17]	0	0	0	0	1	0	2
UnderReporting [17]	1	0	1	0	0	0	0
Allocate Vector [11]	1	0	1	0	0	0	1
Knight [15]	1	0	1	0	0	0	2
Arithmetic Database [15]	3	0	3	1	0	0	0
Total	**15**	**2**	**10**	**3**	**6**	**0**	**23**

these different class instances and assume the accesses to be on the same variable. In some cases this assumption could be dropped with a *points-to* analysis.

The results of our validation/benchmarking procedure are summarized in Table 2. Note that our approach detects a series of dataraces generated by accesses to a single variable, missed by Artho's approach and reported as false negatives in the Tabble, despite the few extra false positives introduced by the corresponding *sensor*. Furthermore, we have significantly reduced Teixeira's number of false positives from 67% to only 26%.

4.3 Limitations of our Approach

Although partially discussed before, we identify below some of the limitations of our approach, not always visible with the set of tests used.

By assuming that all accesses to a field of objects of the same class are made to the same object, i.e., there is no distinction between multiple instances of the same class, we can generate a considerably high number of false positives. A *points-to* analysis will help tackling this issue.

Moreover, not knowing if there is a well defined execution ordering among methods, and thus assuming that every method execution ordering is possible, including concurrent execution, will also yield some false positives. A *may-happens-in-parallel* analysis will help tackling this issue.

Finally, since we are using static analysis, we are collecting all the possible variable accesses in a given method, rather than their real accesses in a program execution. Therefore, in the presence of an `if-then-else` condition, we are merging the accesses made in each distinct branch. This over conservative approximation can also generate false positives, since we are implicitly stating that the union of the accesses of all branches have to be accessed atomically. Further investigation is clearly required, but we believe that more precise static analysis techniques may help tackling this issue.

5. RELATED WORK

There are other works that try to address the issues discussed in this paper. Several of these works target the detection of high-level dataraces in concurrent programs, both static [3, 15, 16], dynamic [1, 18], and hybrid [6]. The detection of *stale-value errors* is also largely covered in the literature [2, 5, 7, 9, 15, 17, 18]. We will further discuss some of these works due to their relevance or similarity to ours.

Undoubtedly, the work presented in [1] is the nearest to our approach since it represents the base and motivation to this work. The authors state that a high-level datarace occurs when a set of related variables should be accessed together, but there is some thread that does not access them atomically. Despite the possibility of both false positives and false negatives, this work has an interesting approach analyzing the relations between variables rather than the interactions between transactions.

The same authors present a data-flow-based technique to detect stale-value errors in [2]. This algorithm makes no strong assumptions and do not depend on annotations, and was implemented using static analysis.

Praun and Gross [17] introduce the concept of *method consistency*, an extension of *view consistency* distinguishing reads and writes. Based on the intuition that the variables that should be accessed atomically in a given method are all the variables accessed inside a synchronized block, the authors define the concept of *method views* that relates to Artho's maximal views. Similarly to our approach, their tool points out all dataraces detected with the concept of view consistency, while missing some others.

The concept of partial order between accesses to the same variable, as introduced in this paper, allows $\mathcal{M}$oTh to detect dataraces missed by the *method consistency* algorithm. One example of this scenario occurs in the UnderReporting test. Moreover, this information also allows us to exclude some of the false positives yielded by *method consistency* algorithm.

However, it is still possible to find a datarace missed by our tool that is detected by [17]. We expect to exclude these cases by developing and plugging new Sensors to our approach.

Wang and Stoller [18] use the concept of thread atomicity to detect and prevent dataraces. Notice that this *atomicity* has a different meaning than the one stated in the ACID properties. Here, atomicity is related to the concept of transaction serialization which guarantees that all concurrent executions of a set of processes is equivalent to a sequential execution of those processes.

An attempt to reduce the number of false positives yield by [18] was made by Teixeira et al. [15]. Motivated by the intuition that the majority of bugs come from two consecutive atomic segments in the same thread, which should be merged into a single one, the authors detect dataraces through the creation and detection of some anomalous access patterns. Our work relates to this one since it also identifies some dataraces caused by accesses to one single variable, that would not be detected with if using only the concept of *View Consistency*. The implementation of the Sensor described in Section 2.3.2 is based on the RwW pattern defined in this work.

Another approach that also detect the same dataraces tackled by the *Single Variable Sensor* is described in [9], where the authors present a type system that verifies the *atomicity* of code blocks.

Beckman et al. [3] present an intra-procedural static analysis, formalized as a type system, using the concept of *access permissions* to detect dataraces. Contrarily to our approach, this work demands that the programmer explicitly declares what are the invariants and the access permissions of the object references of the program. Despite of this observation, this work is still related to ours since it detects the same kind of dataraces in transactional programs written in Java.

Another approach based on statical pattern matching is the work of Vaziri et al. [16]. The authors create a new definition of the concept of datarace, through the theoretical assemblage of all possible anomalous access patterns, embracing both low- and high-level dataraces. Since this approach is based on sets of variables that should be accessed atomically, the user must explicitly declare which variables are related. This work is related to ours since it also tries to statically detect high-level dataraces, despite its incapacity to automatically find out which variables are related.

Finally, Elmas et al. [6] have designed and implemented a Java runtime system that monitors Java program executions, and throws a *DataRaceException* when a datarace is about to occur. Their system supports multiple synchronization idioms, allowing for example the combination of transactions with blocks synchronized with locks.

6. CONCLUDING REMARKS

In this paper we present MoTH, a practical and extensible tool to detect dataraces in transactional memory programs. MoTH is based in an extension of the formalism of view consistency to include distinction between read and write accesses, together with a partial order relation between accesses to the same variable.

We also propose the design a extensible infrastructure that is able to analyze Java Bytecode programs and build a basic knowledge base, which is then used by Datarace Sensors that detect specific dataraces in the program. Two Sensors were designed, one based on the concept of view consistency, and another in the RwW pattern described in [15].

Our approach was validated with a series of known tests taken from the literature. The observed results were significantly better than the ones available up to now by those referred in the related work.

The developed framework can be improved by adding new analysis modules, namely *may-happens-before* and *points-to* analysis, that would reduce the number of false positives currently yielded.

Moreover, more Sensors could also be integrated into MoTH in order to detect dataraces that are not detected by the two existing Sensors described in this paper. These new sensors could exclude the false negatives currently yielded in the validation of our approach.

Finally, the implemented static analysis could be complemented with a dynamic analysis, which would probably provide useful information that would improve the precision of our tool.

Acknowledgments

This work was partially supported by Sun Microsystems under the Sun Worldwide Marketing Loaner Agreement #11497, by the Centro de Informática e Tecnologias da Informação (CITI), and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the research projects PTDC/EIA/74325/2006, PTDC/EIA-EIA/108963/2008, and PTDC/EIA-EIA/113613/2009, and research grant SFRH/BD/41765/2007. We would like to thank the anonymous reviewers for their constructive comments, and express our gratitude to our colleague José Alferes for the diligent review of this paper and for his valuable suggestions.

7. REFERENCES

[1] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, December 2003.

[2] Cyrille Artho, Klaus Havelund, and Armin Biere. Using block-local atomicity to detect stale-value concurrency errors. In Farn Wang, editor, *ATVA*, volume 3299 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2004.

[3] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10):227–244, 2008.

[4] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[5] Michael Burrows and K. Rustan M. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience*, 16(12):1161–1172, October 2004.

[6] Tayfun Elmas and S Qadeer. Goldilocks: a race and transaction-aware java runtime. *ACM SIGPLAN Notices*, pages 245–255, 2007.

[7] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of*

programming languages, pages 256–267, New York, NY, USA, 2004. ACM.

[8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234, May 2002.

[9] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM.

[10] Maurice Herlihy, Victor Luchangco, Mark Moir, and III 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.

[11] IBM's Concurrency Testing Repository.

[12] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[13] Laurie Hendren Vijay Sundaresan Patrick Lam Etienne Gagnon Raja Vallée-Rai and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[14] Nir Shavit and Dan 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.

[15] Bruno Teixeira, João Lourenço, Eitan Farchi, Ricardo Dias, and Diogo Sousa. Detection of transactional memory anomalies using static analysis. In *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '10, pages 26–36, New York, NY, USA, 2010. ACM.

[16] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *ACM SIGPLAN Notices*, 41(1):334–345, January 2006.

[17] Christoph von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology*, page 2004, 2003.

[18] L Wang and S Stoller. Run-Time Analysis for Atomicity. *Electronic Notes in Theoretical Computer Science*, 89(2):191–209, October 2003.