Precise Detection of Atomicity Violations

Ricardo J. Dias, Vasco Pessanha, and João M. Lourenço*

Departamento de Informática and CITI
Universidade Nova de Lisboa, Portugal
{ricardo.dias,v.pessanha}@campus.fct.unl.pt joao.lourenco@fct.unl.pt

Abstract. Concurrent programs that are free of unsynchronized accesses to shared data may still exhibit unpredictable concurrency errors, called *atomicity violations*, which include both high-level data races and *stale-value* errors. Atomicity violations occur when programmers make wrong assumptions about the atomicity scope of a code block, incorrectly splitting it in two or more atomic blocks and allowing them to be interleaved with other atomic blocks. In this paper we propose a novel static analysis algorithm that works on a dependency graph of program variables and detects both high-level data races and *stale-value* errors. The algorithm was implemented for a Java Bytecode analyzer and its effectiveness was evaluated with well known faulty programs. The results obtained show that our algorithm performs better than previous approaches, achieving higher precision for small and medium sized programs, making it a good basis for a practical tool.

1 Introduction

The absence or misspecification of the scope of atomic blocks in a concurrent program may trigger atomicity violations and lead to runtime misbehaviors.

Low-level data races occur when the program includes unsynchronized accesses to a shared variable, and at least one of those accesses is a write, i.e., it changes the value of the variable. Although low-level data races are still a common source of errors and malfunctions in concurrent programs, they have been addressed by others in the past and are out of the scope of this paper. We will consider herein that the concurrent programs under analysis are free from low-level data races.

High-level data races results from the misspecification of the scope of an atomic block, by splitting it in two or more atomic blocks with other (possibly empty) non-atomic block between them. This anomaly is often referred as a high-level data race, and is illustrated in Fig. 1(a). A thread uses the method areEqual() to check if the fields 'a' and 'b' are equal. This method reads both fields in separate atomic blocks, storing their values in local variables, which are then compared. However, due to an interleaving with another thread running

* This work was partially supported by the Euro-TM EU COST Action IC1001, and by the Portuguese National Science Foundation (FCT) in the research project Synergy-VM (PTDC/EIA-EIA/113613/2009) and the research grant SFRH/BD/41765/2007.

the method setPair(), between lines 12 and 13 the value of the pair may have changed. In this scenario the first thread observes an inconsistent pair, composed by the old value of 'a' and the new value of 'b'.

```
1  atomic void getA() {
2    return pair.a;
3  }
4  atomic void getB() {
5    return pair.b;
6  }
7  atomic void setPair(int a, int b){
8    pair.a = a;
9    pair.b = b;
10 }
11 boolean areEqual(){
12   int a = getA();
13   int b = getB();
14   return a == b;
15 }
```

(a) A high-level data race.

```
1  atomic int getX() {
2    return x;
3  }
4  atomic void setX(int p0) {
5    x = p0;
6  }
7  void incX(int val) {
8    int tmp = getX();
9    tmp = tmp + val;
10   setX(tmp);
11 }
```

(b) A stale value error.

Fig. 1: Example of atomicity violations.

Figure 1(b) illustrates a stale value error, another source of atomicity violations in concurrent programs. The non-atomic method incX() is implemented by resorting to two atomic methods, getX() (at line 1) and setX() (at line 4). During the execution of line 9, if the current thread is suspended and another thread is scheduled to execute setX(), the value of 'x' changes, and when the execution of the initial thread is resumed it overwrites the value in 'x' at line 10, causing a lost update. This program fails due to a stale-value error, as at line 8 the value of 'x' escapes the scope of the atomic method getX() and is reused indirectly (by way of its private copy 'tmp') at line 10, when updating the value of 'x' in setX().

In this paper we propose a novel approach for the detection of high-level data races and stale-value errors in concurrent programs. As our proposal only depends on the concept of atomic regions and is neutral concerning the mechanisms used for their identification, the atomic regions are not delimited using locks but rather using an **@Atomic** annotation. Our approach is based on a novel notion of variable dependencies, which we designate as *causal* dependencies. There is a *causal* dependency between two variables if the value of one of them influences the writing of the other. We also extended previous work from Artho et al. [2] by reflecting the read/write nature of accesses to shared variables inside atomic regions and additionally use the dependencies information to detect both high-level data races and stale-value errors. We formally describe the static analysis algorithms to compute the set of *causal* dependencies of a program and define safety conditions for both high-level data races and stale-value errors.

Our approach can yield both false positives and false negatives. However, the experimental results demonstrate that it still achieves high precision when detecting atomicity violations in well know examples from the literature, suggesting its usefulness for software development tools.

In the next Section of this paper we introduce the previous relevant work on detections of high-level data races and stale-value errors; in Section 3 we define a core language and introduce some definitions that support the remainder of the paper, namely Sections 4 and 5, where we propose algorithms for defining *causal dependencies* between variables and for detecting atomicity violations (data races). In Section 6 we briefly describe a tool that applies the proposed algorithms with static analysis techniques for Java Bytecode programs, and compare and discuss the results obtained. We terminate in Section 7 with some final concluding remarks.

2 Background and Related Work

Several past works have addressed the detection of the same class of atomicity violations in concurrent programs as addressed in this paper.

The work from Artho et al. [2] introduces the concept of *view consistency*, to detect high-level data races. A *view* of an atomic block is a set containing all the shared variables accessed (both for reading and writing) within that block. The *maximal views* of a process are those views that are not a subset of any other view. Intuitively, a maximal view defines a set of variables that should always be accessed atomically (inside the same atomic block). A program is free from high-level data races if all the views of one thread that are a subset of the maximal views from another thread form an inclusion chain among themselves.

Our work builds on the proposal from Artho et al. [2], but we extend it by incorporating the type of memory access (read or write) into the views, and refine the rules for detecting high-level data races to consider this additional information and the information given by the *causal* dependencies, with considerable positive impact in the precision of the algorithm, as demonstrated in Section 6.

Praun and Gross [9] introduce *method consistency* as an extension of view consistency. 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 et al's maximal views, which aggregates all the shared variables accessed in a method and also differentiates between read and write memory accesses. Similarly to ours, this approach is more precise than Artho et al's because it also detects stale-value errors. Our algorithm however has higher precision than Praun's and give less false positives, as we use *maximal views* rather than *method views*.

Wang and Stoller [10] use the concept of *thread atomicity* to detect and prevent data races, where thread atomicity guarantees that all concurrent executions of a set of threads is equivalent to a sequential execution of those threads. In an attempt to reduce the number of false positives yield by [10], Teixeira et al. [7] proposed a variant of this algorithm based in the intuition that the majority of the atomicity violations come from two consecutive atomic blocks that should be merged into a single one. The authors detect data races by defining and detecting some anomalous memory access patterns for both high-level data races and stale-value errors. Our approach may be seen as a generalization of

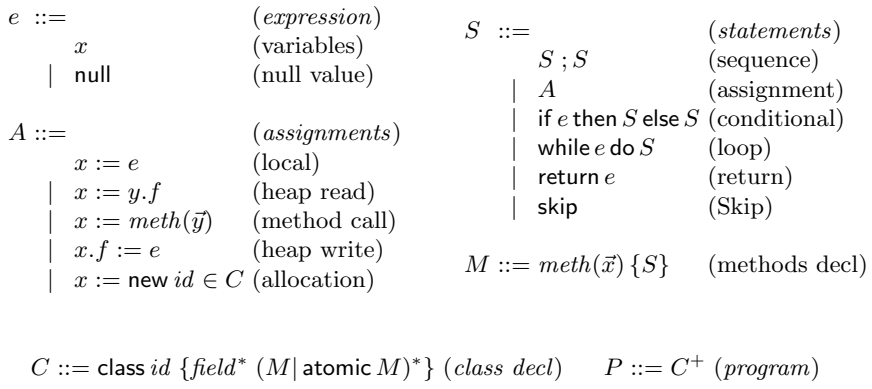

Fig. 2: Core language syntax

this concept of memory access patterns, but in our case supported by the notion of *causal* dependencies between variables, which allow to reduce considerably the number of both false negatives and false positives.

3 Core Language

We start by defining a core language that captures essential features of a subset of the Java programming language, namely class declaration (class $id\{...\}$), object creation (new), field dereferencing ($x.f$), assignment ($x := e$), and method invocation ($meth(\vec{x})$). The syntax of the language is defined by the grammar in Fig. 2.

A program in this language is composed by a set of class declarations. Atomic blocks correspond to methods that are declared using the atomic keyword. We require the restriction of not allowing nesting of atomic blocks i.e., we do not allow to call an atomic method inside another atomic method. Variables can hold integers or object references and boolean values are encoded as integers using the value 1 for true and value 0 for false. We also do not support exception handling as normally found in typical object-oriented languages.

We now define some sets that are necessary to the understanding of the following sections:

- Classes: is the set of all class identifiers of all classes declared in the program.
- Fields: is the set of all class fields defined in the program.
- Methods: is the the set of all methods defined in the program.
- Atomics $\subseteq$ Methods: is the subset of methods that were declared as atomic.

We define a local (stack) variable as a pair of the form (x, m) where x is the variable identifier and $m \in$ Methods is the method where this variable is

declared. For the sake of simplicity we write the pair (x, m) as only x whenever is not ambiguous to do so. The set of all local variables of a program is denoted as $\mathsf{LocalVars}$.

We define a global variable as an object field and we represent it as the pair (c, f) where $c \in \mathsf{Classes}$ represents the class where field $f \in \mathsf{Fields}$ is declared. The set of all global variables is denoted as $\mathsf{GlobalVars}$. These global variables appear in the code when dereferencing an object reference. For instance, in the statement $x.f := 4$, the expression $x.f$ represents a global variable of the form (c, f) where c is the class of the object reference pointed by local variable x. We define a function $\mathsf{typeof} : \mathsf{LocalVars} \rightarrow \mathsf{Classes}$, which given a local variable returns the class of the object reference that it holds. So, in the example above $c = \mathsf{typeof}(x)$.

Please note that by deciding to represent an access to a field of an object as a pair with the class of the object reference and the field accessed, we are not able to differentiate between different object instances of the same class, and hence we may consider that there is always at most one object instance of each declared class in the program. This allows us to avoid pointer analysis at the cost of losing precision and becoming unsound in some cases but, as the results in Section 6 show, this design choice has proven to be very effective.

Finally we define the set $\mathsf{Vars} \equiv \mathsf{LocalVars} + \mathsf{GlobalVars}$, which corresponds to all variables used in the program, both local and global variables.

4 Causal Dependencies

There is a *Causal* dependency, which we will designate herein only as dependency, between two program variables (local or global) if the value read from one variable influences the value written into the other. For instance, the following expression

```
y := x
```

generates a dependency between variable x and y because the value that is written into variable y was read from variable x. As another example, consider the following code:

```
if (x == 0) { y := 4 }
```

In this example, the variable y is written only if the condition $x = 0$ is true, thus it depends on the current value of variable x and therefore there is also a dependency between variables x and y. We represent a dependency between two variables x and y as $x \hookrightarrow y$ where $x \in \mathsf{Vars}$ is the variable read and $y \in \mathsf{Vars}$ is the variable written.

For each program we can compute a directed graph of *causal* dependencies. The information provided by this graph plays an important role in finding correlations between variables, which can be used to detect atomicity violations. We can define two kinds of correlations between variables.

Definition 1 (Direct Correlation). *There is a* direct *correlation between a read variable x and a written variable y if there is a path from x to y, in a dependency graph $\mathcal{D}$.*

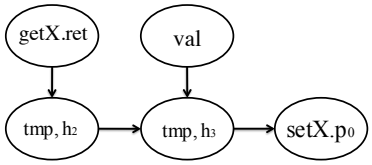

Fig. 3: Dependency graph example

Definition 2 (Common Correlation). *There is a* common *correlation between a read variable x and a read variable y if there is a written variable z, where $z \neq x$ and $z \neq y$, for which there is a path from x to z and another path from y to z, in a dependency graph $\mathcal{D}$.*

In the following section we describe how to compute the graph of dependencies using symbolic execution.

4.1 Dependency Analysis

The construction of the dependency graph is done in two steps. In the first step we only detect data dependencies between variables. In the second step we detect control dependencies between variables. In the end we merge all dependencies in a single graph.

Data Dependencies The accurate detection of data dependencies relies on the precise localisation of where the variables are defined. SSA (Single Static Assignment) [1] could be used, because each variable would only have one definition site, but this only works for local variables, and we still need to track each definition site for global variables. Therefore we did not use SSA as internal representation and we solve the problem by defining a new variable version whenever the variable is updated.

A variable version is defined as a triple of the form (x, h, m) where $x \in \mathsf{Vars}$ is a variable (local or global), h is a unique identifier, and $m \in \mathsf{Atomics} \cup \{\bot\}$ indicates if this variable is used inside an atomic method or not ($\bot$). The set of all variable versions is denoted as $\mathsf{Versions}$.

The unique identifier h is a hash value based on the line of code of the respective definition site. If the version of the variable is not known in the current context, as in the case of method arguments, a special hash value is used. We denote this special hash value as $h_?$.

Figure 3 depicts the dependency graph for the method '`incX()`' from Fig. 1(b). For the sake of simplicity, we omitted the method (m) part of the version representation. We denote $getX.\,ret$ as the return value of method `getX()`, and $setX.\,p0$ as the parameter of method `setX(int p0)`. Both the return value and the parameter do not need to have an hash value associated, and thus we omitted it from their representation.

In method incX(**int** val), the value returned by the method getX() is written into a temporary variable *tmp*, which is then incremented using parameter *val* and is then used as a parameter on the invocation of method setX(**int** p0).

While analyzing this method, we first start by creating the dependency $getX.\,ret \hookrightarrow (tmp, h_2)$ between the return value of getX() method and variable *tmp* with an hash value h_2. In the next statement variable *tmp* is redefined with a value resulting from the sum of the previous *tmp* variable and the *val* parameter, and hence we create two dependencies $(tmp, h_2) \hookrightarrow (tmp, h_3)$ and $val \hookrightarrow (tmp, h_3)$, where the new version of *tmp* variable has the hash value h_3. Finally, we invoke method setX(**int** p0) with the value of *tmp* as parameter and therefore we create the dependency $(tmp, h_3) \hookrightarrow setX.\,p0$.

The symbolic execution rules are defined as a transition system ($\langle \mathcal{D}, \mathcal{H}, S\rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}'\rangle$) over a state composed by a dependency graph $\mathcal{D}$ and a set of versions, denoted has $\mathcal{H} \subseteq \mathsf{Versions}$, which holds the current versions of each program variable. In a single program point, we may find different versions of the same variable because our analysis over-approximates the run-time state of a program. The rules can be depicted in Figure 4, and we always omit the method (m) parameter from the representation of a variable version.

Function $\mathsf{ver}_{\mathcal{H}}$ is used to retrieve the set of current versions of a variable, and is defined as follows:

Definition 3 (Version Retrieval). *Given a set of versions $\mathcal{H}$ and a variable $v \in \mathsf{Vars}$:*

$$\mathsf{ver} : \mathcal{P}(\mathsf{Versions}) \times \mathsf{Vars} \to \mathcal{P}(\mathsf{Versions})$$

$$\mathsf{ver}_{\mathcal{H}}(v) \triangleq \begin{cases} \{(v,h,m) \mid (v,h,m) \in \mathcal{H}\} & \text{if } \exists (v,h,m) \in \mathcal{H} \\ \{(v, h_?, m)\} & \text{otherwise} \end{cases}$$

If a variable version cannot be found in $\mathcal{H}$, a version with the special hash value $h_?$ is returned.

Every time that a variable is written, it is created a new version for such variable and all other existing current versions are replaced by the new one. We define an helper function $\mathsf{subs}_{\mathcal{H}}$ for this purpose as:

Definition 4 (Version Substitution). *Given a set of versions $\mathcal{H}$ and a variable version $(v,h,m) \in \mathsf{Versions}$:*

$$\mathsf{subs} : \mathcal{P}(\mathsf{Versions}) \times \mathsf{Versions} \to \mathcal{P}(\mathsf{Versions})$$

$$\mathsf{subs}_{\mathcal{H}}((v,h,m)) \triangleq (\mathcal{H} \setminus \{(v,h',m') \mid (v,h',m') \in \mathcal{H}\}) \cup \{(v,h,m)\}$$

Each hash value is generated using the function nhash, which given a statement S generates a new and unique hash value based in the line number of that statement. This function is deterministic in the sense that for any statement S the same hash value is always returned.

At the beginning of the analysis, the sets $\mathcal{D}$ and $\mathcal{H}$ are empty. We represent the parameters of methods as $meth.\,p_i$, and the return value of a method

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

$$\frac{\begin{array}{c} h = \mathsf{nhash}(x := y) \\ \mathcal{H}' = \mathsf{subs}_{\mathcal{H}}((x,h)) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x,h) \mid v \in \mathsf{ver}_{\mathcal{H}}(y)\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x := y \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle} (\textsc{Assign})$$

$$\frac{\begin{array}{c} c = \mathsf{typeof}(y) \qquad h = \mathsf{nhash}(x := y.f) \qquad \mathcal{H}' = \mathsf{subs}_{\mathcal{H}}((x,h)) \\ \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x,h) \mid v \in \mathsf{ver}_{\mathcal{H}}((c,f))\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x := y.f \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle} (\textsc{Heap Read})$$

$$\frac{\begin{array}{c} c = \mathsf{typeof}(x) \qquad h = \mathsf{nhash}(x.f := y) \qquad \mathcal{H}' = \mathsf{subs}_{\mathcal{H}}(((c,f),h)) \\ \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow ((c,f),h) \mid v \in \mathsf{ver}_{\mathcal{H}}(y)\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x.f := y \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle} (\textsc{Heap Write})$$

$$\frac{h = \mathsf{nhash}(x := \mathsf{new}\, C()) \qquad \mathcal{H}' = \mathsf{subs}_{\mathcal{H}}((x,h))}{\langle \mathcal{D}, \mathcal{H}, x := \mathsf{new}\, C() \rangle \Longrightarrow \langle \mathcal{D}, \mathcal{H}' \rangle} (\textsc{Allocation})$$

$$\frac{\begin{array}{c} h = \mathsf{nhash}(x := meth(\vec{y})) \qquad \mathsf{spec}(meth) = \langle \mathcal{D}_f, \mathcal{H}_f \rangle \qquad \mathcal{D}' = \mathcal{D}_f \cup \mathcal{D} \\ \mathcal{D}'' = \mathcal{D}' \cup \{v_i \hookrightarrow meth.\, p_i \mid y_i \in \vec{y} \wedge v_i \in \mathsf{ver}_{\mathcal{H}}(y_i)\} \cup \{meth.\, ret \hookrightarrow (x,h)\} \\ \mathcal{H}' = \{(v,h) \mid (v,h) \in \mathcal{H} \wedge ((v,h_?) \in \mathcal{H}_f \vee (v,h) \notin \mathcal{H}_f)\} \\ \mathcal{H}'' = \{(v,h) \mid (v,h) \in \mathcal{H}_f \wedge h \neq h_?\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x := meth(\vec{y}) \rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}' \cup \mathcal{H}'' \rangle} (\textsc{Meth Call})$$

$$\frac{\begin{array}{c} \langle \mathcal{D}, \mathcal{H}, S_1 \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle \qquad \langle \mathcal{D}, \mathcal{H}, S_2 \rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}'' \rangle \\ \mathcal{H}''' = \mathcal{H}' \cup \mathcal{H}'' \cup \{(v,h_?) \mid (v,h_1) \in \mathcal{H}' \wedge (v,h_2) \notin \mathcal{H}''\} \\ \cup \{(v,h_?) \mid (v,h_1) \in \mathcal{H}'' \wedge (v,h_2) \notin \mathcal{H}'\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{if}\, b\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{D}' \cup \mathcal{D}'', \mathcal{H}''' \rangle} (\textsc{Conditional})$$

$$\frac{\begin{array}{c} \langle \mathcal{D}, \mathcal{H}, S \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle \qquad \mathcal{H}'' = \mathcal{H} \cup \mathcal{H}' \cup \{(v,h_?) \mid (v,h_1) \in \mathcal{H} \wedge (v,h_2) \notin \mathcal{H}'\} \\ \cup \{(v,h_?) \mid (v,h_1) \in \mathcal{H}' \wedge (v,h_2) \notin \mathcal{H}\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{while}\, b\, \mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{D} \cup \mathcal{D}', \mathcal{H}'' \rangle} (\textsc{Loop})$$

$$\frac{\mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow \mathsf{retVar} \mid v \in \mathsf{ver}_{\mathcal{H}}(x)\}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{return}\, x \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H} \rangle} (\textsc{Return}) \qquad \frac{}{\langle \mathcal{D}, \mathcal{H}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{D}, \mathcal{H} \rangle} (\textsc{Skip})$$

Fig. 4: Symbolic execution rules of data dependencies analysis

as *meth. ret*. When evaluating the Return statement, the return value of the method is denoted as retVar.

All assignment operations, namely Assign, Heap Read, and Heap Write, create dependencies between all versions of the variables used in the right side of the assignment and the new version of the assigned variable. The newly generated version is then used to replace all existing versions of that same variable.

In the rule Meth Call, the function spec returns the result, denoted as $\langle \mathcal{D}_p, \mathcal{H}_p \rangle$, of the analysis of method *meth*. The dependencies in $\mathcal{D}_p$ are merged with the current dependencies and we create a dependency between each value that is passed as an argument to *meth* and the respective declared parameter $meth.\, p_i$. We also need to update the variables' versions that are generated inside

the method. If a variable was redefined ($h \neq h_?$) inside *meth* then we replace the existing versions with the new version, otherwise we keep the current versions. Finally, we add one more dependency between the return value of method *meth* and the assigned value.

In the rule CONDITIONAL, the dependencies are generated in both branches and are merged with the initial $\mathcal{D}$. We also generate the versions for each branch, and if a variable x has a version $h \neq h_?$ in one branch but there is no version for the same variable in the other branch, then we generate a special version $h_?$ for variable x and we join it to all the other versions. The intuition behind this operation is that if a variable is written only in one of the branches then we also need to add the case that the variable might not have been written. The rule LOOP is similar to the CONDITIONAL rule. The remaining rules should be self-explanatory.

After analyzing all methods of the program we get a dependency graph for the whole program, based on data-flow information. Next, we have to add the remaining dependencies based on the control flow information.

Control Dependencies If an assignment or return statement is guarded by some condition then that assignment or return statement depends on the variables used in the condition. This situation may occur with every conditional statement such as an if then else, or a while loop.

The analysis of control dependencies traverses the control flow graph and keeps the set of variables that the assignments may depend on. When an assignment or return statement is found we create a dependency between the current variables, that it may depend on, and the respective assigned variable.

The symbolic execution rules are shown in Figure 5 as a transition system ($\langle \mathcal{IS}, \mathcal{D}, S \rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}' \rangle$). The state is composed by a set of conditional variables $\mathcal{IS} \subseteq \mathsf{Versions}$, which correspond to the variable versions that the current statement depends on, and a dependency graph $\mathcal{D}$. In the beginning of the analysis the dependency graph is empty, and the set of conditional variables has the union of all conditional variables that are present at all calling contexts of the method that is going to be analyzed. For instance, given the program methods m_1, m_2 and m_3 where method m_1 calls method m_2 with the current conditional variables set $\mathcal{IS} = \{c_1, c_2\}$, and m_3 calls method m_2 with the current conditional variables set $\mathcal{IS} = \{c_3, c_4\}$, then the initial set of conditional variables when analyzing method m_2 is $\mathcal{IS} = \{c_1, c_2, c_3, c_4\}$.

In the end of this analysis the resulting graph of dependencies is merged with the one that resulted from the data dependencies analysis, described in the previous section, thus forming the complete graph of *causal* dependencies.

For every kind of assignment we create a dependency between the current conditional variables and the assigned variable. This situation may occur in the rules ASSIGN, HEAP READ, HEAP WRITE, ALLOCATION and METH CALL. In the case of a return statement, as in rule RETURN, we create a dependency with the special variable retVar.

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

$$\frac{h = \mathsf{nhash}(x := y) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x, h) \mid v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := y \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Assign})$$

$$\frac{h = \mathsf{nhash}(x := y.f) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x, h) \mid v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := y.f \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Heap Read})$$

$$\frac{\begin{array}{c} c = \mathsf{typeof}(x) \\ h = \mathsf{nhash}(x.f := y) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow ((c, f), h) \mid v \in \mathcal{IS}\} \end{array}}{\langle \mathcal{IS}, \mathcal{D}, x.f := y \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Heap Write})$$

$$\frac{h = \mathsf{nhash}(x := \mathsf{new}\, C()) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x, h) \mid v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := \mathsf{new}\, C() \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Allocation})$$

$$\frac{\begin{array}{c} h = \mathsf{nhash}(x := meth(\vec{y})) \\ \mathsf{spec}(meth) = \langle \mathcal{IS}_f, \mathcal{D}_f \rangle \qquad \mathcal{D}' = \mathcal{D} \cup \mathcal{D}_f \cup \{v \hookrightarrow (x, h) \mid v \in \mathcal{IS}\} \end{array}}{\langle \mathcal{IS}, \mathcal{D}, x := meth(\vec{y}) \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Meth Call})$$

$$\frac{\begin{array}{c} \mathcal{IS}' = \mathcal{IS} \cup \{b\} \\ \langle \mathcal{IS}', \mathcal{D}, S_1 \rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}' \rangle \qquad \langle \mathcal{IS}', \mathcal{D}, S_2 \rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}'' \rangle \end{array}}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{if}\, b\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \cup \mathcal{D}'' \rangle} (\textsc{Conditional})$$

$$\frac{\mathcal{IS}' = \mathcal{IS} \cup \{b\} \qquad \langle \mathcal{IS}', \mathcal{D}, S \rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}' \rangle}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{while}\, b\, \mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D} \cup \mathcal{D}' \rangle} (\textsc{Loop})$$

$$\frac{\mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow \mathsf{retVar} \mid v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{return}\, x \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Return}) \qquad \frac{}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D} \rangle} (\textsc{Skip})$$

Fig. 5: Symbolic execution rules of control dependencies analysis

In the rules Conditional and Loop, we analyze each branch with a new set of conditional variables, which include the current conditional variables plus the variable of the condition. Each variable is actually a variable version with an unique hash value. When we exit the scope of the condition we remove the condition variable and proceed with the analysis. The remaining rules are self-explanatory.

The result of these two analysis generate the graph of *causal* dependencies that is used to detect the existence of atomicity violations in a concurrent program, as we will show in the following sections.

5 Atomicity Violations

The purpose of our work is to detect two kinds of atomicity errors, the high-level data race and the stale-value error, that may occur during the execution of concurrent programs that use atomic blocks to guarantee mutual exclusion in the access to shared data.

The definition of both errors assume that the concurrent program has no low-level data races, meaning that all accesses to shared variables are done inside atomic blocks.

5.1 High Level data races

A *view*, as described by Artho et al. in [2], expresses what variables are accessed inside a given atomic code block. We extend this definition by also keeping the kind of access (read or write) that was made for each variable in the *view*.

Please note that a *view* only stores global variables. Local variables are not shared between threads and thus do not require synchronized accesses.

We denote as $\mathsf{Accesses}$ the set of memory accesses made inside an atomic block. An access $a \in \mathsf{Accesses}$ is a pair of the form (α, v) where $\alpha \in \{r, w\}$ represents the kind of access (r-read or w-write) and $v \in \mathsf{GlobalVars}$ is a global variable[1]. A *view* is a subset of $\mathsf{Accesses}$ and the set of all views in a program is denoted as Views. A *view* is always associated with one atomic method, and we define the bijective function Γ that given a *view* returns the associated atomic method as:

$$\Gamma : \mathsf{Views} \rightarrow \mathsf{Atomics}$$

The inverse function, denoted as Γ^{-1}, returns the *view* associated with a given atomic method. The set of *generated views* of a process p, denoted as $V(p)$, corresponds to the atomic blocks executed by one process, and is defined as:

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

The predicate $\mathsf{executes}$ asserts if a method m may be executed by process p, and is defined by an auxiliary static analysis that computes the set of processes and the atomic methods that are called in each process.

We can refine the previous definition of $V(p)$ with a parameter α, where $\alpha \in \{r, w\}$, to get only the views of a process with read (V_r) or write accesses (V_w).

Definition 5 (Procedure Views).

$$V_\alpha(p) = \{v_2 \mid v_1 \in V(p) \wedge v_2 = \{(\alpha, x) \mid (\alpha, x) \in v_1\}\} \quad \textit{where } \alpha \in \{r, w\}$$

We defined a static analysis to compute a *view* of an atomic method. Every time a global variable is read or written, the corresponding read or write access is created and added to the *view*. The *view* resulting from a method call is merged with the current *view* that is being computed. In the case of conditional and loop statements we perform an over-approximation union of the *views* of each branch. In the end of the analysis we have the set of *views* corresponding to the atomic methods present in the program code.

[1] Please remember that global variables are represented as a pair with a class identifier and the field accessed.

The *maximal views* of a process, denoted as M_α, are all the views of the process that are not a subset of any other view in that same process. A maximal view is defined as follows:

Definition 6 (Maximal Views). *Given a process p, a* maximal view v_m *is defined as:*

$$v_m \in M_\alpha(p) \Leftrightarrow v_m \in V_\alpha(p) \wedge (\forall v \in V_\alpha(p) : v_m \subseteq v \Rightarrow v = v_m) \quad \textit{where } \alpha \in \{r, w\}$$

Each *maximal view* represent the set of variables that should be accessed atomically, i.e., should always be accessed in the same atomic block.

Given a set of views of a process p and a *maximal view* v_m of another process, we define the read or write *overlapping views* of process p with *view* v_m as all the non empty intersection views between v_m and the views of process p.

Definition 7 (Overlapping Views). *Given a process p and* maximal view v_m*:*

$$\mathsf{overlap}_\alpha(p, v_m) \triangleq \{v_m \cap v \mid v \in V_\alpha(p) \wedge v_m \cap v \neq \emptyset\} \quad \textit{where } \alpha \in \{r, w\}$$

The notion of compatibility between a process p and a *view* v_m, defined in [2], states that a process p and a *view* v_m are *compatible* if all their *overlapping views* form a chain. We extended this definition with the information given by the *causal* dependencies graph, and we additionally require that, even if the read overlapping views do not form a chain, there may not exist a *common correlation* (Definition 2) between the variables in the read overlapping views.

Definition 8 (Process Compatibility). *Given a process p and* maximal view v_m*:*

$$\begin{aligned} \mathsf{comp}_w(p, v_m) &\Leftrightarrow \forall v_1, v_2 \in \mathsf{overlap}_w(p, v_m) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1 \\ \mathsf{comp}_r(p, v_m) &\Leftrightarrow \forall v_1, v_2 \in \mathsf{overlap}_r(p, v_m) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1 \\ &\qquad \vee \neg\, \mathsf{CommonCorrelation}(v_1, v_2) \end{aligned}$$

The intuition behind this additional condition is that, even if two shared variables that belong to a *maximal view* were read in different atomic blocks, we will only consider that there is an incompatibility if both variables are used in a common write operation.

We can now define the *view consistency* safety property in terms of the compatibility between all pairs of processes of a program. A process may only have *views* that are compatible with all *maximal views* of another process. A program is free from high-level data races if the following condition holds:

Definition 9 (View Consistency).

$$\forall p_1, p_2 \in \mathcal{P}_\mathcal{S}, m_r \in M_r(p_1), m_w \in M_w(p_1) : \mathsf{comp}_w(p_2, m_r) \wedge \mathsf{comp}_r(p_2, m_w)$$

where $\mathcal{P}_\mathcal{S}$ *is the set of processes.*

5.2 Stale-Value Error

Stale-value errors are a class of atomicity violations that are not detected by the *view consistency* property. Our approach to detect this kind of errors uses the graph of *causal* dependencies to detect values that escape the scope of an atomic block (e.g., by assigning a shared variable to a local variable) and are later used inside another atomic block (e.g., by assigning the previous local variable to a shared variable).

First we define the set $\mathsf{IVersions} \subseteq \mathsf{Versions}$, which stores all global variable versions that were accessed inside an atomic block. Each variable version has a parameter m that indicates in which atomic method it was defined, or has the value $\perp$ if it was not used inside an atomic method.

Definition 10 (Atomic Variable Version). *A global variable version* (x, h, m) *is an atomic variable if:*

$$(x, h, m) \in \mathsf{IVersions} \Leftrightarrow (x, h, m) \in \mathsf{Versions} \wedge x \in \mathsf{GlobalVars} \wedge m \neq \perp$$

Now we define a new graph, denoted as $\mathcal{D}_\mathcal{V}$, which represent the dependencies between views. A labeled edge of this graph $\mathcal{D}_\mathcal{V}$ is represented as (m_1, x, m_2) where $m_1, m_2 \in \mathsf{Atomics}$ and $x \in \mathsf{GlobalVars}$, and can be interpreted as atomic method m_2 depends on atomic method m_1 through global variable x. Intuitively, this means that the value of variable x exited the scope of method m_1 and entered the scope of method m_2, and while it was out of the atomic scopes it might have become outdated.

Each edge (m_1, x_1, m_2) of a *view* dependency graph $\mathcal{D}_\mathcal{V}$, is created when, given two version variables $a_1 = (x_1, h_1, m_1) \in \mathsf{IVersions}$ and $a_2 = (x_2, h_2, m_2) \in \mathsf{IVersions}$, and a *causal* dependency graph $\mathcal{D}$, the following conditions hold:

$$\begin{aligned}(&\mathsf{DirectCorrelation}(\mathcal{D}, a_1, a_2) \wedge m_1 \neq m_2) \vee (m_1 = m_2 \\ &\quad \wedge \mathsf{DirectCorrelation}(\mathcal{D}, a_1, m_1\,.\,ret) \wedge \mathsf{DirectCorrelation}(\mathcal{D}, m_1\,.\,ret, m_1\,.\,p_i) \\ &\quad \wedge \mathsf{DirectCorrelation}(\mathcal{D}, m_1\,.\,p_i, a_2))\end{aligned}$$

The predicate $\mathsf{DirectCorrelation}$ asserts if two variables are *directly correlated* according to Definition 1. These conditions state that there is a dependency between m_1 and m_2 through variable x_1, if the variable version a_1 is directly correlated with a_2 when m_1 and m_2 are two different atomic methods, or if the two methods m_1 and m_2 are the same, then we must be sure that the value of x_1 left out the scope of the method and then entered it again.

A process p writes in a variable $x \in \mathsf{Vars}$ if there is a write access on variable x in one of the views of process p:

$$\mathsf{writes}(x, p) \Leftrightarrow \exists v \in V_w(p) : (w, x) \in v$$

The safety property for *stale-value* errors can be defined as the case where no process writes to a global variable that leaves, and then enters, the scope of an atomic method of another process.

Table 1: Results for benchmarks — *Set 1*

Tests	AV	False Negatives			False Positives			Acc. Vars	LOC	Time (sec.)
		MoTH	Artho	Teix.	*MoTH*	Artho	Teix.			
Connection [4]	2	0	1	1	0	0	1	34	112	45
Coord03 [2]	1	0	0	0	0	0	3	13	170	43
Local [2]	1	0	1	0	0	0	1	3	33	42
NASA [2]	1	0	0	0	0	0	0	7	121	43
Coord04 [3]	1	0	0	0	0	0	3	7	47	40
Buffer [3]	0	0	0	0	1	0	7	8	64	41
DoubleCheck [3]	0	0	0	0	1	0	2	7	51	41
StringBuffer [5]	1	0	1	1	0	0	0	12	52	44
Account [9]	1	0	1	0	0	0	0	3	65	40
Jigsaw [9]	1	0	0	0	0	0	1	33	145	40
OverReporting [9]	0	0	0	0	0	0	2	6	52	42
UnderReporting [9]	1	0	1	0	0	0	0	3	31	39
Allocate Vector [6]	1	0	1	0	0	0	1	24	304	41
Knight [7]	1	0	1	0	0	0	2	10	223	41
Arithmetic Database [7]	3	0	3	1	1	0	0	24	416	54
Total	**15**	**0**	**10**	**3**	**3**	**0**	**23**	–	–	–

Definition 11 (Stale-Value Safety).

$$\forall p \in \mathcal{P}_{\mathcal{S}}, (m_1, x, m_2) \in \mathcal{D}_{\mathcal{V}} : \neg\, \mathsf{writes}(x, p) \text{ where } \mathcal{P}_{\mathcal{S}} \text{ is the set of processes}$$

If there is a *view* dependency for variable x and there is a process p that writes on that variable then a *stale-value* error is detected.

6 Evaluation

To evaluate the accuracy of our algorithms and techniques, we adapted and implemented the theoretical framework described in the previous sections to the Java Bytecode language, where the atomic methods are defined using the **@Atomic** method annotation. We used the data-flow analysis infrastructure of the Soot framework [8] to implement all the described analysis.

Our tool starts by parsing a Java bytecode program and computing a set of analysis, namely: *process analysis* to identify which threads may exist when executing the program; *instance type analysis* to handle Java interfaces and dynamic dispatching; *views analysis*, to compute the *views* of each atomic method; inter-procedural *causal dependency analysis*, to compute dependencies between variables used in assignments and conditional code blocks. Once all these analysis are concluded, the tool creates the *causal dependency graph*. Another analysis is then ran over this dependency graph to identify atomic blocks that break the atomicity violation safety properties.

Besides comparing our results with those reported on the literature for individual benchmarks, we did an exhaustive comparison with two other approaches: the work of Artho et al [2], because our approach is an extension of this work; and the work of Teixeira et al [7], because their results are currently a reference

Table 2: Results for benchmarks — *Set 2*

Tests	AV	False Negatives		False Positives		Acc. Vars	LOC	Time (sec.)
		MoTH	Artho	MoTH	Artho			
Elevator [9]	16	0	16	6	4	39	558	46
Philo [9]	0	0	0	2	0	9/594	96	45/612
Tsp [9]	0	0	0	2	0	635	795	869
Store	2	0	1	0	1	44/608	901	149/1763
Total	**18**	**0**	**17**	**10**	**5**	–	–	–

for the field. The results presented were obtained by running our tool with the algorithms described in this paper; by using Artho et al's algorithm implemented with static analysis techniques (rather than the dynamic analysis reported in [2]); and by running Teixeira's tool on the Java source (instead of the Bytecode).

Tables 1 and 2 summarize the results achieved by applying our tool to a set of benchmarking programs, most of them well known from related works and compares them with the two works cited above. Teixeira's tool was unable to process some of the benchmarks, so they are reported in a separate second set. Columns *AV* indicate the number of known atomicity violations, *false negatives* indicate the number of known program atomicity violations that were missed by the approach[2], *false positives* indicate the number of reported but non-existing atomicity violations, *Acc. Vars* indicate the number of variables accessed inside atomic regions and is an indication of the problem size, together with the number of *LOC*, and how long it took for our analysis to run.

In the case of Table 2, the benchmarks Philo and Store have two different values for *accessed variables* and *time*. The second values report on the original benchmarks, which includes some (non-essential) calls to I/O methods in the JDK library. The first values report on a tailored version of the benchmarks where those calls to the JDK library were commented.

For the benchmarks listed in Table 1, our approach revealed a very high accuracy by reporting no false negatives and only three false positives. The false positive in the Buffer benchmark is due to an assumption claim from its authors that is not implemented in the actual code. The information collected by the Causal Dependency Analysis is incomplete and imprecise and originates false positives in the Double Check and Arithmetic Database benchmarks while checking for stale-value errors, which are not detected by Artho et al's approach.

For the benchmarks listed in Table 2, our appropriate again reveled very high accuracy, as although it reported 10 false positives (vs. only 5 from Artho et al's), it reported zero false negatives (vs. 17 from Artho et al's). These benchmarks also indicate that our algorithms scale well with the the size of the problem, both in the number of accessed variables inside the atomic blocks and the number of lines of code.

[2] The identification of false negative is only possible because the sets of atomicity violations in the benchmarking programs are well known.

7 Conclusions

In this paper we presented a novel approach to detect high-level data races and stale-value errors in concurrent programs. The proposed approach relies on the notion of *causal* dependencies to improve the precision of previous detection techniques. The high-level data races are detected using an algorithm based on a previous work by Artho et al. refined to distinguish between read and write accesses and extended with the information given by the *causal* dependencies. The stale-value errors are detected using the information given by the *causal* dependencies, which exposes the values of variables that escaped an atomic block and entered into another atomic block.

Our detection analysis still remains unsound mainly due to the absence of pointer analysis and to the way that *views* are computed. But these design decisions allowed us to maintain the scalability of our approach without incurring in a strong precision loss, as our experimental results confirm.

We evaluated our analysis techniques with well known examples from the literature and compared them to previous works. Our results show that we are able to detect all atomicity violations present in the examples, while reporting a low number of false positives.

References

1. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 1–11. POPL '88, ACM, San Diego, CA, USA (1988)
2. Artho, C., Havelund, K., Biere, A.: High-level data races. Software Testing, Verification and Reliability 13(4), 207–227 (Dec 2003)
3. Artho, C., Havelund, K., Biere, A.: Using block-local atomicity to detect stale-value concurrency errors. In: Wang, F. (ed.) Automated Technology for Verification and Analysis, LNCS, vol. 3299, pp. 150–164. Springer Berlin Heidelberg (2004)
4. Beckman, N.E., Bierhoff, K., Aldrich, J.: Verifying correct usage of atomic blocks and typestate. SIGPLAN Not. 43(10), 227–244 (2008)
5. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proc. of the 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 256–267. POPL '04, Venice, Italy (2004)
6. IBM HRL — Concurrency Testing Repository
7. Teixeira, B., Lourenço, J.M., Farchi, E., Dias, R.J., Sousa, D.G.: Detection of transactional memory anomalies using static analysis. In: Proc. of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. pp. 26–36. PADTAD '10, ACM, New York, NY, USA (2010)
8. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: Proc. of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. pp. 125–135. CASCON '99, IBM Press (1999)
9. von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. In: Journal of Object Technology. p. 2004 (2003)
10. Wang, L., Stoller, S.: Run-Time Analysis for Atomicity. Electronic Notes in Theoretical Computer Science 89(2), 191–209 (Oct 2003)