A sound and scalable method for static data race detection in Java programs

Ali Ghanbari Mehran S. Fallah

August 25, 2018

Abstract

A scalable, yet sound and precise, static data race detection method is still a need. In this paper, we achieve such a method through what we devise for concurrency and points-to analyses. Our concurrency analysis of a program results in an abstract execution graph. Each node of such a graph abstracts away the events that may be raised by the corresponding instruction at run-time. The edges of the graph represent a relation on abstract events making a conservative approximation of the happened-before relation on concrete events. In this way, we can model program executions in a precise manner. Such an abstraction also allows parametric events which are useful in constructing parametric summaries for methods, or functions, in a given program. This results in a scalable method for data race analysis. We prove that our method is sound. As part of the proof, it is proven that the way we treat recursive functions gives the same results when every chain of the functions calling each other is analyzed as a whole. The proposed data race detection method is implemented and applied to a number of Java programs. Experimental results show that its precision is comparable to the state-of-the-art, sound data race detection methods for Java. At the same time, unlike existing solutions, it is scalable and can be used for analyzing large practical programs.

1 Introduction

A data race occurs when two concurrent threads access the same location of memory without enough ordering constraints enforced on them. This error is subtle and may have disastrous consequences justifying the need for automating its prevention and detection. In general, there are two approaches to confront data races: *dynamic* and *static* [39]. By a dynamic analysis, one detects data races through monitoring the events raised during the runs of the program. For a static analysis, on the contrary, there is no need to run the program and the aim is to find possible data races on the basis of the code of the program. The earlier research in this area has mainly focused on dynamic data race detection methods where the inherent problem is the impossibility of tracking all executions of the given program. It is for this reason that static analyses—in isolation or along with dynamic ones—are still required in eliminating data races [6].

There are several requirements for an effective method for static data race detection: soundness, precision, and scalability [33, 5]. The requirements refer to the ability of the algorithm in detecting all possible data races, recognizing

genuine data races and avoiding false positives, and handling large programs, respectively. There may also be other requirements such as the ability to handle a variety of synchronization idioms and open programs such as libraries. Despite considerable advances in data race detection, there remain shortcomings in the proposed solutions due to the inherent tradeoff among the above requirements. The annotations needed to make a sound and scalable data race detection method acceptably precise hinder its practical application [20, 8]. Automated inference of the required annotations is not tractable either [1]. At the same time, soundness may be sacrificed in a scalable and precise algorithm [16, 4]. Moreover, sound and precise methods do not scale [32, 37]. Thus, there is a need for a sound, precise, and scalable data race detection algorithm with a tolerable amount of annotations.

In this paper, we present a data race detection method that achieves soundness and scalability and arrives at an acceptable level of precision. Such a method typically involves concurrency[1] and points-to analyses. To be more precise, one may also need a thread-escape analysis [3]. We give such algorithms for the Java programming language. The algorithms bring scalability through a modular analysis at the level of Java methods. In comparison to the similar attempts made in the literature, our method is sound, scalable, and has the potential of delivering useful information about possible races in open programs. The proposed method can potentially support a wide range of synchronization idioms. The experiments conducted as part of this research show that our method is precise enough for practical purposes.

Our points-to analysis keeps track of all interprocedural pointer-information flows, thereby preserving soundness. The analysis is context-sensitive and, thus, precise [31]. This, in turn, results in a precise data race detection method. It is worth noting that our points-to analysis indeed integrates the good features of context-sensitive and context-insensitive analyses. In fact, library classes are analyzed context-insensitively, whereas application classes are analyzed in a context-sensitive manner. This leads to a remarkable performance gain at the cost of a negligible imprecision. Our concurrency analysis involves deriving so-called abstract execution graphs by which one can infer the concurrency relation among pairs of instructions. A node of the graph is a compile-time object which is called an abstract event and reflects the concurrency-critical run-time events that may be raised by the execution of the corresponding instruction. The abstraction is such that we can handle a wide variety of synchronization idioms such as locks, semaphores, and messages.

We prove that the proposed data race detection method is sound, something that is addressed informally in similar works. More precisely, we first make the observation that in a language that allows recursion, the subject reduction proof [36], which states that properties are preserved under state transitions, is intriguing in case the proof builds on the results of modular analyses. On the other hand, the same proof could be conducted much easier in the case of a non-modular analysis—by a non-modular analysis, we mean a summary-based analysis in which the functions (methods) calling each other in a chain are analyzed as a whole. Thus, if we show that the soundness of the modular analysis is reduced to that of the corresponding non-modular analysis, it will be

[1]It is worth noting that by a "concurrency analysis" we mean an algorithm that determines, for each pair of instructions in a program, whether they may execute concurrently or not.

enough to give a subject reduction proof for the latter.

We implement a prototype and apply it to a set of multi-threaded Java programs of different sizes. The results indicate that the proposed method is practical, in that it scales to large industrial programs, and its precision is close to tools such as Chord [32]. It is worth noting that the scalability of a summary-based analysis has also been corroborated in several studies [55, 27, 53]. In fact, in-lining method summaries at method call sites is computationally simpler than solving a system of data flow equations involving several methods. We also observe that by a judicious use of lazy data structures, one may circumvent the problem of augmented method summaries.

In the following section, we set the scene for solving the problem. Section 3 presents two methods for concurrency analysis, one is modular and the other is non-modular. In Section 4, we prove that the two analysis methods are equivalent. Section 5 reports experimental results. The related work is discussed in Section 6. Section 7 concludes the paper. The proof of soundness for the non-modular concurrency analysis, the syntax and semantics of our object language, and a somewhat ameliorated summary-based points-to analysis appear in the online appendix of this paper [21].

2 Basic Concepts

2.1 Program Text and Control flow Information

In order to avoid repeating basic concepts, we have adopted the terminology and notations of [34]. For data flow analysis of the programs of our object language ConcurrentJava$^+$ whose syntax is given in [21], we label the subexpressions of a given expression in a recursive manner. As with [34], we call a substring of a program text, which is represented by a node in the control flow graph, an elementary block. Such a block may be a labeled expression or another labeled entity denoting part of the program text. For any labeled expression of the form $[\texttt{synchronized}\ e_1\ \texttt{in}\ e_2]^{\ell_n}_{\ell_x}$, we define the two elementary blocks $[enterMonitor(e_1; e_2)]^{\ell_n}$ and $[exitMonitor(e_1; e_2)]^{\ell_x}$ which correspond to the entry and exit points of the block, respectively. Similarly, we define the two elementary blocks $[begin(var = e_1; e_2)]^{\ell_n}$ and $[end(var = e_1; e_2)]^{\ell_x}$ for a labeled expression of the form $[\texttt{let}\ var = e_1\ \texttt{in}\ e_2]^{\ell_n}_{\ell_x}$. We also consider the two elementary blocks $[begin(v_1, \ldots, v_k)]^{\ell_n}$ and $[end(v_1, \ldots, v_k)]^{\ell_x}$ for any labeled method declaration $t\ mn(t_1\ v_1, \ldots, t_k\ v_k)\ [\{]^{\ell_n}\ e\ [\}]^{\ell_x}$.

We represent the set of labels with $Labels$ and that of the labels occurring in a particular program P with $Labels_P$. Similar to [34], we also use the functions $init$, $final$, and $labels$ to obtain the initial label, the final label, and the set of labels occurring in an elementary block. Furthermore, we use the functions $blocks$ and $flows$ to obtain the set of elementary blocks associated with a labeled substring of the program text and the set of flows in a control flow graph corresponding to a labeled substring of the program text. We shall use $Flows$ and $Blocks$ to represent the sets of all flows and elementary blocks, respectively. The sets $Flows_P$, $Blocks_P$ denote the set of flows and elementary blocks associated with a given program P.

The set of valid identifiers is represented by $Identifiers$ (see the syntax given in [21]). The sets of valid field names, variable names, and method names

are noted *Fields*, *Variables*, and *Methods*, respectively. The first two sets are disjoint and subsumed by the set $Identifiers$. The set *Methods* is defined to be $\{c.mn \mid c, mn \in Identifiers\}$. Similarly, the sets of field names, variable names, and method names of a given program P are denoted by $Fields_P$, $Variables_P$, and $Methods_P$. We use the function *declOf* to obtain the declaration of a method in the program under consideration.

In order to obtain nontrivial control flow information at method call sites and thread creation sites, it is assumed that, for a given program P, we have done a rapid type analysis (RTA), based on what has been presented in [50], to obtain a program call graph[2]. Thus, we feel free to use $callees(\ell)$ to obtain the set of method names that may be called at a method call site, or a thread creation site, with label ℓ. Furthermore, we define the function *mayCall* from $Methods_P$ to the set of all subsets of $Methods_P$. This function returns the set of method that may be directly called by a given method. Going one step further, we define the function $mayCall^*$ from $Methods_P$ to the set of all subsets of $Methods_P$, which is supposed to return the set of all method that may be called, directly or indirectly, by a given method.

With a call graph at hand, we may identify the set of strongly connected components in the graph. This set is represented by SCC. Roughly speaking, SCC denotes the set of pairs of methods that are recursively defined in terms of each other. By using the relation SCC, we define the predicate $isRecursive$ to identify those methods that are defined recursively.

Given a labeled program P, a method m in P, and the function $flows$, we can construct a control flow graph for the method. To do so, the set of the nodes of the graph is defined to be $labels(declOf(m))$ and that of its edges to be $flows(declOf(m))$. Since such a control flow graph is used in our modular analyses, we call it a *modular control flow graph*. The graph is represented by $MCFG_m^P$, which equals (B, F) with B and F as its sets of nodes and edges, respectively. Furthermore, by using the function *callees*, we may construct an *interprocedural control flow graph* for that method. This graph is noted $ICFG_m^P = (B, F)$ and equals $MCFG_m^P$ if $isRecursive(m)$ does not hold. Otherwise, it includes interprocedural flows and the sets B and F of its nodes and edges are obtained from

$$B = \bigcup_{m' \in mayCall^*(m)} labels(m'),$$

$$\begin{aligned} F = &\Big(\bigcup_{m' \in mayCall^*(m)} flows(m')\Big) \\ &\cup \{(init(b); init(declOf(m'))) \mid b \in B \ \wedge \ b \text{ is a call site} \\ &\qquad \wedge\ m' \in callees(init(b))\} \\ &\cup \{(final(declOf(m')); final(b)) \mid b \in B \ \wedge \ b \text{ is a call site} \\ &\qquad \wedge\ m' \in callees(init(b))\}. \end{aligned}$$

Note that we represent an intraprocedural flow by a pair of labels separated by a comma and an interprocedural flow by a pair of labels separated by a semicolon.

[2]Note that this is just an assumption for our theoretical purposes. For our implementation, a much more precise call graph backed by a points-to analysis is computed.

$\otimes$	1	*	?
1	1	*	1
*	*	*	*
?	1	*	?

Figure 1: The operator $\otimes$.

2.2 Points-to and Multiplicity Information

In the online appendix of this paper [21], we give a context-sensitive points-to analysis method the ultimate goal of which is to compute a points-to graph. We call the nodes of a points-to graph *abstract locations*. An abstract location corresponds to an object creation site or a formal parameter of a method. Throughout the paper, we will use $mayPointsTo(e)$ to obtain the set of abstract locations the expression e may point to. Note that the points-to analysis, supposed to be constructed along with the concurrency analysis, results in a *mayPointsTo* function that does not need any context information.

Multiplicities represent the cardinality of the set of concrete entities corresponding to an abstract entity. Compile-time abstractions, such as abstract events, abstract locations, and abstract threads denote classes of concrete (run-time) entities. According to the placement of an expression in a program, the abstract entity associated with the expression may denote a single concrete entity or a non-sigleton set of entities. For example, since a thread creation site at the entry-point expression of a program (see Appendix C of [21]) will not be executed repeatedly, it denotes a single run-time thread. Thus, the corresponding abstract thread is of multiplicity 1. On the other hand, since a thread creation site at the body of a recursively defined method may be executed more than once, we give it the multiplicity * representing a number which is possibly greater than one. Since we analyze each method of a program in isolation, however, we are unsure about the number of times a non-recursive method may be called. Therefore, a thread creation site located at the body of a non-recursive method will receive an unknown multiplicity, which is denoted by a question mark. In this paper, we use the function *labMult* which, given a label, returns the multiplicity of the location in a program with that label.

It should be noted that multiplicities are not limited to (syntactic) places in a program. The following are some observations regarding multiplicities.

- Any abstract location or an abstract thread that is created by an abstract thread of multiplicity *, or in the body of a recursive method, is of multiplicity $*$.
- If an abstract location, corresponding to a set of thread objects, is of multiplicity *, then so is the abstract thread associated with the objects.
- The multiplicity of the special thread `main`, running the entry-point expression of a program, is 1.

These observations together with the syntactic nature of multiplicities lead to the definition of the operator $\otimes$ described in Figure 1. We shall use this operator to determine the multiplicity of the abstract entities introduced at call and thread creation sites.

3 Concurrency Analysis

3.1 Abstract Events and Executions

A program can be regarded as a set of *executions*. Each execution is defined to be a sequence of *events*. An event is an execution instance of an instruction such as read, write, lock, unlock, etc. Two events are said to be *conflicting* if they both access the same memory location, at least one of them is a write, and the accesses are accomplished by different threads of execution.

Lamport's happened-before relation [24], which is denoted by $\rightarrow$, is an irreflexive partial ordering on the set of events in an execution. For the sake of simplicity, we take locking as the only synchronization mechanism. Later, in Section 3, we shall discuss other mechanisms as well. Let $\langle e_k \rangle$ be a possibly finite execution. The relation $\rightarrow$ is the smallest relation satisfying the following conditions: (1) if e_i and e_j are two events of a thread with $i < j$, then $e_i \rightarrow e_j$, (2) if e_i is the event of exiting from a monitor (unlock), e_j is the event of entering into the same monitor (lock), and $i < j$, then $e_i \rightarrow e_j$, and (3) the relation is transitive. Now, we are able to give a precise definition of a data race error and an ill-behaved program.

Definition 1. *Given an execution E, there is a data race error in E if there exist two conflicting events in the execution with no happened-before relation between them. A program $P = \{E_0, E_1, \dots\}$ is ill-behaved, or has a data race, if there exists an execution $E_t \in P$ with a data race error in E_t.*

In order to detect data races statically, we need to introduce compile-time abstractions of events and the happened-before relation. The notion of abstract events is introduced in this section, while we defer the definition of a compile-time approximation of the happened-before relation until Section 4. An abstract event represents a set of run-time, i.e., concrete, events that correspond to different execution instances of an instruction. Thus, in order to effectively identify data races in a program, an abstract event should contain all the information about the set of concrete events it represents. We define the set of abstract events as

$$\mathit{AE} = \{\kappa(\mathit{id}, \tau, \rho) \mid \kappa \in \mathit{Kinds} \ \wedge \ \mathit{id} \in \mathit{EID} \ \wedge \ \tau \in \mathit{TID} \ \wedge \ \rho \in \mathit{eventLoci}(\kappa)\},$$

where *Kinds* is the set of event kinds. In this paper, without loss of generality, we suppose that there are only four kinds of basic instructions, namely derefence, assignment, enter monitor (lock), and exit monitor (unlock), that correspond to event kinds R, W, N, and X, respectively. Let L be a finite subset of *Labels*, and S_L be the set of all non-empty strings made up of labels in L. The set *EID* of abstract event identifiers and the set *TID* of abstract thread identifiers are defined by

$$\begin{aligned} &\mathit{EID} = S_L, \\ &\mathit{TID} = \{\tau^m \mid \tau \in (S_L \cup \{\Lambda\}) \ \wedge \ m \in \mathit{Multiplicities}\}, \end{aligned}$$

where Λ denotes a null-string. The set of locations that may be affected by an abstract event is a function of its kind. We define $\mathit{eventLoci}(\kappa)$ to be the set of all pairs of abstract locations and field names if $\kappa \in \{R, W\}$ and to be the set of all abstract locations for $\kappa \in \{N, X\}$.

$meth$	::=	$t\ mn(par^*)\ q\ \{e\}$	Method declaration
par	::=	$t\ var$	Parameter
t	::=	$q\ c$	Type
q	::=	$(\texttt{norace})^?$	Qualifier

Figure 2: Extended abstract syntax, where c is a class identifier (see Appendix C of [21].)

At the first place, we need to explain the information space extracted by the analysis. Abstract executions are compile-time approximations of the set of executions of a program. The set of all abstract executions is defined to be

$$AX = \{(E, \mapsto) \mid E \subseteq AE \ \wedge \ \mapsto \subseteq E \times E\}.$$

Given two abstract executions $\eta_1 = (E_1, \mapsto_1)$ and $\eta_2 = (E_2, \mapsto_2)$, we define the partial ordering $\sqsubseteq$ on AX by

$$\eta_1 \sqsubseteq \eta_2 \iff E_1 \subseteq E_2 \ \wedge \ \mapsto_1 \subseteq \mapsto_2 .$$

Once the partial ordering is defined, the least element would be $\bot = (\emptyset, \emptyset)$ and the least upper bound of every subset Y of AX may be defined in the obvious way

$$\bigsqcup Y = \left(\bigcup\{E \mid (E, \mapsto) \in Y\}, \bigcup\{\mapsto \mid (E, \mapsto) \in Y\}\right).$$

Note that $\bigcup$ when precedes a collection of sets is used to obtain the union of all of its elements. Evidently, $\mathcal{AX} = (AX, \sqsubseteq, \bigsqcup, \bot)$ is a complete lattice.

3.2 The `norace` Type Qualifier

One major source of inaccuracy in prior data race detection methods is the false positives arising from benign data races. We may prune such false reports by marking only those fields that are supposed to be data race free. We shall, of course, devise typing rules that prevent a regularly-typed identifier from being an alias for an object of `norace`-qualified type.

In doing so, we extend the abstract syntax of our object language as shown in Figure 2. Note that a type qualifier located at method headers, i.e., before its body, qualifies the type of the identifier `this`. We also add the following typing rules to the type system given in Appendix C of [21]. The rule at the left guarantees the well-formedness of qualified types and the one at the right defines subtyping relation on such types. Evidently, a qualified type is compatible only with qualified types.

$$\frac{P; E \vdash t \qquad \forall t'.\ t \neq \texttt{norace}\ t'}{P; E \vdash \texttt{norace}\ t}, \qquad \frac{P; E \vdash t_1 <: t_2}{P; E \vdash \texttt{norace}\ t_1 <: \texttt{norace}\ t_2}.$$

3.3 Non-modular Analysis

In this subsection, we give a non-modular version of our concurrency analysis. Without loss of generality, we can assume that every ConcurrentJava$^+$ program has a collection of methods declared in classes. Let $\mathit{ICFG}^P_{c.mn} = (B, F)$ be the interprocedural control flow graph for an arbitrary method $c.mn$ of program

P. Our analysis is an instance of the *monotone framework* [34] and could be identified as a sextuple $\mathsf{CA}^{\mathsf{int}} = (\mathcal{AX}, \mathcal{F}, F, E, \iota, f_{\_})$, where $\mathcal{F}$ is a set of one-place and two-place monotone functions on AX that is closed under function composition and includes identity functions, $E = \{init(declOf(c.mn))\}$, and $\iota = \bot$. The last component is a function mapping $Labels_P$ to appropriate functions in $\mathcal{F}$. For $\ell \in Labels_P$, this function chooses an appropriate transition function according to the syntactic structure of the expression labeled by ℓ.

Our concurrency analysis is a may, forward, flow-sensitive, and context-sensitive data flow analysis. Given a program P, for each method in $Methods_P$ chosen in a special order (given in Appendix XXX), the interprocedural analysis constructs the system of equations

$$CA^{\mathsf{int}}_{\circ}(\ell) = \bigsqcup\{CA^{\mathsf{int}}_{\bullet}(\ell') \mid (\ell', \ell) \in F \vee (\ell'; \ell) \in F\}; \qquad \ell \in B, \tag{1}$$

where $CA^{\mathsf{int}}_{\circ}(\ell)$ and $CA^{\mathsf{int}}_{\bullet}(\ell)$ are variables. For each $\ell \in B$ that does not label a call site, we define

$$CA^{\mathsf{int}}_{\bullet}(\ell) = f_{\ell}(CA^{\mathsf{int}}_{\circ}(\ell)). \tag{2}$$

For call sites, the data flow equations are defined by

$$CA^{\mathsf{int}}_{\bullet}(\ell_c) = f^{call}_{\ell_c}(CA^{\mathsf{int}}_{\circ}(\ell)), \tag{3}$$

where $\ell_c \in B$ and there is a label $\ell_r \in B$ such that

$$CA^{\mathsf{int}}_{\bullet}(\ell_r) = f^{ret}_{\ell_c,\ell_r}(CA^{\mathsf{int}}_{\circ}(\ell_c), CA^{\mathsf{int}}_{\circ}(\ell_r)), \tag{4}$$

where $\ell_c \in B$ and $[e.mn(e^*)]^{\ell_c}_{\ell_r} \in Blocks_P$.

In (2)–(4), f_{ℓ}, $f^{call}_{\ell_c}$, and $f^{ret}_{\ell_c,\ell_r}$ are transition functions. The general form of a transition function for an elementary block, except for those relating to method call sites, is

$$f_{\ell}(\eta) = \eta \sqcup gen(\eta)([e]^{\ell}); \qquad \ell \in B \ \wedge \ [e]^{\ell} \in Blocks_P.$$

Next, we shall present our definition for the function *gen* and the two other transition functions. Before defining the transition functions, we need to define our notion of *method summary*. Roughly speaking, method summary is a data structure for storing the results obtained from analyzing the methods of a program until a particular moment during the analysis. In fact, the method summary for a given program is updated, with new summaries added to it, whenever new methods are analyzed.

Definition 2 (The result of the non-modular analysis)**.** *A pair of functions* $(CA^{\mathsf{int}}_{\circ}, CA^{\mathsf{int}}_{\bullet})$ *that satisfies equations (1)-(4) for a method c.mn in the program* P *with the method summary* ms *is called the result of the concurrency analysis of the method. We use the following function for the sake of a simpler notation.*

$$IntAnalyze(P, c.mn, ms) = \{(\circ, CA^{\mathsf{int}}_{\circ}), (\bullet, CA^{\mathsf{int}}_{\bullet})\}.$$

Definition 3 (Method summary)**.** *A method summary is a function mapping a method name to a set of abstract executions. If the set is nonempty, then it*

contains the value of the result of the concurrency analysis of the method on its final block. In other words, this mapping could be defined as

$$ms : Methods \to \mathcal{P}(AX),$$

where $ms(c.mn) = \emptyset$ *means that the method in question has not been analyzed yet, hence no summary is available for it. Otherwise, for a method summary* ms'*, such that for each* $m \in mayCall(c.mn)$ *we have* $ms'(m) \neq \emptyset$*, we define the method summary for* $c.mn$ *by*

$$ms(c.mn) = \{IntAnalyze(P, c.mn, ms')(\bullet)(final(declOf(c.mn)))\}.$$

The function *gen*, in the case of elementary blocks of the form $[(t)e]^{\ell}$, $[var]^{\ell}$, $[\texttt{new}\ t]^{\ell}$, $[\texttt{null}]^{\ell}$, $[begin(var = e_1; e_2)]^{\ell_n}$, $[end(var = e_1; e_2)]^{\ell_x}$, $[begin(var_{1\ldots k})]^{\ell_n}$, and $[end(var_{1\ldots k})]^{\ell_x}$ does nothing interesting and simply yields the value $\perp$. In what follows, we shall define the function, and the two other transition functions, for the remaining syntactic forms.

3.3.1 Transition function for $[e_0.fn]^{\ell}$

This kind of statement, when executed, produces one event of kind R. Accordingly, we define

$$gen(\eta)([e_0.fn]^{\ell}) = \begin{cases} (e, \mapsto_g) & ; e_0 \text{ is of a } \texttt{norace} - \text{qualified type,} \\ \perp & ; o.w. \end{cases}$$

where e and $\mapsto_g$ are defined by

$$\begin{aligned} &e = R(\ell, \Lambda^?, mayPointsTo(e_0) \times \{fn\}), \\ &\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\ldots, \Lambda^?, \ldots) \wedge \eta = (E, \mapsto)\}. \end{aligned}$$

3.3.2 Transition function for $[e_1.fn = e_2]^{\ell}$

Such a statement produces one event of kind W. Thus, we define

$$gen(\eta)([e_1.fn = e_2]^{\ell}) = \begin{cases} (e, \mapsto_g) & ; e_1 \text{ is of a } \texttt{norace} - \text{qualified type,} \\ \perp & ; o.w. \end{cases}$$

where e and $\mapsto_g$ are

$$\begin{aligned} &e = W(\ell, \Lambda^?, mayPointsTo(e_1) \times \{fn\}), \\ &\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\ldots, \Lambda^?, \ldots) \wedge \eta = (E, \mapsto)\}. \end{aligned}$$

3.3.3 Transition function for $[enterMonitor(e_1; e_2)]^{\ell_n}$

It produces one event of kind N. Thus, we define

$$gen(\eta)([enterMonitor(e_1; e_2)]^{\ell_n}) = (\{e\}, \mapsto_g),$$

where e and $\mapsto_g$ are defined as

$$\begin{aligned} &e = N(\ell_n, \Lambda^?, L), \\ &\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\ldots, \Lambda^?, \ldots) \wedge \eta = (E, \mapsto)\} \end{aligned}$$

in which L is assumed to be

$$L = \begin{cases} \emptyset & ; |mayPointsTo(e_1)| > 1 \\ & \quad \vee \exists n \in mayPointsTo(e_1).\ n = \texttt{i}(\ldots, *) \\ mayPointsTo(e_1) & ; o.w. \end{cases}$$

3.3.4 Transition function for $[exitMonitor(e_1; e_2)]^{\ell_x}$

This statement produces one event of kind X. Therefore, we define

$$gen(\eta)([enterMonitor(e_1; e_2)]^{\ell_n}) = (\{e\}, \mapsto_g)$$

with e and $\mapsto_g$ defined as follows:

$$\begin{array}{l} e = X(\ell_x, \Lambda^?, L), \\ \mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\ldots, \Lambda^?, \ldots) \wedge \eta = (E, \mapsto)\} \end{array}$$

in which L is

$$L = \begin{cases} \emptyset & ; |mayPointsTo(e_1)| > 1 \\ & \quad \vee \exists n \in mayPointsTo(e_1).\ n = \texttt{i}(\ldots, *) \\ mayPointsTo(e_1) & ; o.w. \end{cases}$$

3.3.5 Transition function for $[e.\texttt{start}()]^{\ell}$

We assume that for each method called (as thread body at a thread creation site) which do not have SCC relation with the method being analyzed, there exists a method summary. More precisely,

$$\forall m \in callees(\ell).\ (m, c.mn) \notin SCC \implies ms(m) \neq \emptyset.$$

Thus, the function gen for this kind of statement is defined by

$$gen(\eta)([e.\texttt{start}()]^{\ell}) = (E_I, \mapsto_g),$$

where $(E_I, \mapsto_I) = inst(\ell, \bigsqcup\{s \mid ms(c'.\texttt{run}) = \{s\} \wedge c'.\texttt{run} \in callees(\ell) \wedge (c.mn, c'.\texttt{run}) \notin SCC\})$ and we have

$$\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\ldots, \Lambda^?, \ldots) \wedge e \in E_I \wedge \eta = (E, \mapsto)\} \cup \mapsto_I .$$

In the above definition, the function $inst$ is defined by

$$inst(\ell, \eta) = i_3(\ell, i_2(\ell, i_1(\ell, \eta))).$$

As discussed in the online appendix of this paper [21], in order to construct a modular points-to analysis, we introduce the notion of external nodes. In the above definition, i_1 is responsible for interacting with our points-to analysis through instantiating external nodes. Assuming that the points-to analysis is correct, we can ignore this function when we are proving the correctness of the (non-modular) concurrency analysis. Here, for simplicity, we have omitted the definition of the function. The function i_2 instantiates event identifiers and is defined in the following way.

$$i_2(\ell, (E, \mapsto)) = (E', \mapsto'),$$

where $E' = E\,[\kappa(\ell s, \dots)/\kappa(s, \dots)]$ and $\mapsto' = \mapsto\,[\kappa(\ell s, \dots)/\kappa(s, \dots)]$. Moreover, the function i_3 instantiates identifiers and multiplicities of threads for those events that may be raised at the thread creation site. Let $m = \mathit{labMult}(\ell)$. The function is defined to be $i_3(\ell, (E, \mapsto)) = (E', \mapsto')$ in which E' and $\mapsto'$ are defined by

$$E' = E\left[\kappa(\dots, \ell s^{m \otimes m'}, \dots)/\kappa(\dots, s^{m'}, \dots)\right],$$
$$\mapsto' = \mapsto\left[\kappa(\dots, \ell s^{m \otimes m'}, \dots)/\kappa(\dots, s^{m'}, \dots)\right].$$

Note that there is no constraint on s.

3.3.6 Transition functions for $[e_0.mn'(e_1, \dots, e_k)]^{\ell_c}_{\ell_r}$

In order to guarantee the existence of summaries for those methods that may be called at the method call site ℓ_c and that do not have SCC relation with the method containing the call site, i.e., the method $c.mn$, we give the following condition.

$$\forall m \in \mathit{callees}(\ell_c).\ (m, c.mn) \notin SCC \implies ms(m) \neq \emptyset.$$

We define two transition functions for method calls, one for the call and the other for the return.

$$f^{call}_{\ell_c}(\eta) = \bot,$$
$$f^{ret}_{\ell_c, \ell_r}(\eta, \eta') = \eta \sqcup (E_I, \mapsto_g),$$

where, assuming $(E_I, \mapsto_I) = \mathit{inst}'(\ell_c, \eta' \sqcup R)$, we have

$$\mapsto_g = \{(e', e) \mid e' \in E \wedge e' = \kappa(\dots, \Lambda^?, \dots) \wedge e \in E_I \wedge \eta = (E, \mapsto)\} \cup \mapsto_I,$$
$$R = \bigsqcup\{s \mid ms(m) = \{s\} \wedge m \in \mathit{callees}(\ell_c) \wedge (m, c.mn) \notin SCC\}.$$

Now, we define inst' through

$$(E_I, \mapsto_I) = i'_3(\ell_c, i_2(\ell_c, i_1(\eta'))),$$

where i_1, i_2 are as before and i'_3 is responsible for instantiating identifiers and multiplicities for those call sites that may spawn some thread. Let $m = \mathit{labMult}(\ell)$. The function is defined to be $i'_3(\ell, (E, \mapsto)) = (E', \mapsto')$, where E' and $\mapsto'$ are defined as

$$E' = E\left[\kappa(\dots, \ell s^{m \otimes m'}, \dots)/\kappa(\dots, s^{m'}, \dots)\right],$$
$$\mapsto' = \mapsto\left[\kappa(\dots, \ell s^{m \otimes m'}, \dots)/\kappa(\dots, s^{m'}, \dots)\right].$$

with an extra constraint $s \neq \Lambda$.

3.4 Modular Analysis

In order to simplify the proof of equivalence of modular and non-modular analyses, we designed the analysis method of Section 3.3 as a summary-based analysis, which makes the two analyses as similar as possible. Unfortunately, because of unlimited growth of the number of events in abstract executions, there would be no algorithm doing the non-modular analysis. Therefore, we need a computable, yet fully modular, analysis.

All constituent parts of the modular analysis is the same as its non-modular version, except for the transition function for method call sites. To avoid an unlimited number of events, the modular analysis avoids recursive analysis, and in-lining the effect, of called methods at recursive call sites. Indeed, a method which has already been analyzed will not be analyzed again and the value $\bot$ will be considered as the result of the new analysis. As we shall see, this can be easily accomplished with the aid of an extra data structure which we name the call stack and represent by cs.

The call stack is a set of names of the methods for which the analysis is finding a solution. During the analysis of a method in the set, it may be needed to analyze another method. In such a case, we add the name of the callee to the call stack, interrupt the analysis of the current method, and start analyzing the body of the callee. If the callee is already a member of the set, we use $\bot$ as the result of the analysis of its body. As we shall see in the rest of this section, using $\bot$ as explained above will not deteriorate the usefulness of the information obtained through the modular analysis. In effect, the information obtained through modular and non-modular analyses are equivalent from a data race detection point of view. This indicates that the soundness of the non-modular analysis implies the soundness of the modular analysis.

In what follows, we present our modular analysis method for $c.mn$, a method in the given program P. Here, we use the control flow graph $MCFG^P_{c.mn} = (B, F)$, which does not contain interprocedural flows. The analysis is an instance of the monotone framework, and can be represented by the sextuple $\mathsf{CA}^{\mathsf{mod}} = (\mathcal{AX}, \mathcal{F}, F, E, \iota, f_.)$ in which all of the components are defined as in the non-modular case except for $\mathcal{F}$ which is defined to be the set of all monotone functions on AX closed under function composition.

Variables of the system of data flow equations are $CA^{\mathsf{mod}}_\circ(\ell)$, $CA^{\mathsf{mod}}_\bullet(\ell)$ in which $\ell \in Labels_P$. The system of equations is, then, defined by

$$CA^{\mathsf{mod}}_\circ(\ell) = \bigsqcup\{CA^{\mathsf{mod}}_\bullet(\ell') \mid (\ell', \ell) \in F\}; \qquad \ell \in B. \tag{5}$$

Furthermore, for each label $\ell \in B$, including the labels of call sites, we have

$$CA^{\mathsf{mod}}_\bullet(\ell) = f_\ell(cs)(CA^{\mathsf{mod}}_\circ(\ell)), \tag{6}$$

where f_ℓ is a transition function of the form $f_\ell(cs)(\eta) = \eta \sqcup gen'(cs, \eta)([B]^\ell)$ with $[B]^\ell$ a member of $Blocks_P$.

Definition 4 (The result of the modular analysis)**.** *Given a method c.mn in program P, a call stack cs containing c.mn, and the method summary ms, the pair of functions* $(CA^{\mathsf{mod}}_\circ, CA^{\mathsf{mod}}_\bullet)$ *satisfying equations (5) and (6) is the result of the modular concurrency analysis. We pack the result by using the function*

$$ModAnalyze(P, c.mn, ms, cs) = \{(\circ, CA^{\mathsf{mod}}_\circ), (\bullet, CA^{\mathsf{mod}}_\bullet)\}; \qquad c.mn \in cs.$$

Note that we need to introduce a new definition for a method summary. In fact, we define it in terms of *ModAnalyze* instead of *IntAnalyze*. Now, by defining transition functions, we complete the formulation of the modular

analysis. It should also be stated that the function gen' builds on the function gen of Section 3.3 as follows:

$$gen'(cs,\eta)([B]^{\ell}) = \begin{cases} gen''(cs,\eta)([B]^{\ell}) & ;\ell \text{ labels a method call site} \\ gen(\eta)([B]^{\ell}) & ;o.w. \end{cases}$$

3.4.1 Transition function for $[e_0.mn(e_1,\ldots,e_k)]^{\ell_c}_{\ell_r}$

We assume that there exists a summary for any method that may be called at the site if it does not call (recursively) any method in cs. More formally,

$$\forall m \in callees(\ell_c).\ (\forall m' \in cs.\ (m,m') \notin SCC) \implies ms(m) \neq \emptyset. \tag{7}$$

Furthermore, for each method that may be called at the site, we assume that its body, i.e., the text of its body, is available if it may call (recursively) some methods in cs. That is,

$$\forall m \in callees(\ell_c).\ (\exists m' \in cs.\ (m,m') \in SCC) \implies declOf(m) \neq \Lambda. \tag{8}$$

Now, we define the function gen'' as follows:

$$\begin{array}{l} gen''(cs,\eta)([e_0.mn(e_1,\ldots,e_k)]^{\ell_c}_{\ell_r}) = \\ \quad inst'(\bigsqcup\{s \mid ms(m) = \{s\} \wedge m \in callees(\ell) \wedge \forall m' \in cs.\ (m,m') \notin SCC\} \\ \qquad \sqcup(\bigsqcup\{ModAnalyze(P,m,ms,cs \cup \{m\})(\bullet)(final(declOf(m))) \mid \\ \qquad\quad m \in callees(\ell_c) \wedge m \notin cs \wedge \exists m' \in cs.\ (m,m') \in SCC\})), \end{array}$$

where $inst'$ is the instantiation function defined in Section 3.3. The value of a method contained in the call stack is also defined to be $\bot$. Note that, for an interprocedural analysis, we should analyze methods in reverse pseudo-topological order of the call graph of the program. In Appendix F of [21], we present an algorithm that enforces such an ordering in the analysis of methods of a given program and guarantees that (7) will always hold.

4 Equivalence of Modular and Non-modular Analyses

Abstract executions contain comprehensive information about events and their ordering. In this section, we first define a number of predicates on the concurrency information contained in abstract executions. We also formalize the notion of *safety*, i.e., being race-free, for abstract executions. Then, we prove that the proposed modular and non-modular analyses of concurrency are equivalent in the sense that they lead to the same results when employed in a data race detection method.

In order to judge the concurrency relation among two given events of an abstract execution, we should ascertain if the events are produced by a single thread or by the threads that make use of synchronization mechanisms to place an ordering on the events. Here, we only consider the locking discipline; as we shall see in Section 4.1, it would not be a challenge to take into account other synchronization mechanisms. Two events from two distinct threads are ordered by some lock if and only if the events, in the sequence of events associated

with each thread, are surrounded by monitor operations affecting the lock. In order to recognize monitor events surrounding a given event, we need to identify *enterMonitor* events before, and *exitMonitor* events after the event. To do so, we should define the notion of *matching monitor events*.

Definition 5. *Two monitor events of kinds N and X are said to be matching monitor events if they are of different kinds and operate on the same object. More formally, two abstract events $e_1 = \kappa_1(\dots, L_1)$ and $e_2 = \kappa_2(\dots, L_2)$ are matching monitor events, denoted by $match(e_1, e_2)$, iff*

$$\kappa_1, \kappa_2 \in \{N, X\} \ \wedge \ \kappa_1 \neq \kappa_2 \ \wedge \ L_1 = L_2.$$

According to the notion of lock acquisition and release, matching monitor events do indeed neutralize the effect of each other. Therefore, they are akin to matching parentheses around an event e if enter events before e and exit events after e do not match any other monitor events before and after e. We define the sets of enter and exit events before an event e in an abstract execution η as follows:

$$\begin{aligned} N_\eta^{\leftarrow}(e) = \{e_N \mid\ & e_N = N(\dots, \tau, \dots) \ \wedge \ e_N \mapsto e \ \wedge \ e_N, e \in E \\ & \wedge\ e = \kappa(\dots, \tau, \dots) \ \wedge \ \eta = (E, \mapsto)\}, \end{aligned}$$

$$\begin{aligned} X_\eta^{\leftarrow}(e) = \{e_X \mid\ & e_X = X(\dots, \tau, \dots) \ \wedge \ e_X \mapsto e \ \wedge \ e_X, e \in E \\ & \wedge\ e = \kappa(\dots, \tau, \dots) \ \wedge \ \eta = (E, \mapsto)\}. \end{aligned}$$

Similarly, the following defines the sets of enter and exit events after an event e in an abstract execution η.

$$\begin{aligned} N_\eta^{\rightarrow}(e) = \{e_N \mid\ & e_N = N(\dots, \tau, \dots) \ \wedge \ e \mapsto e_N \ \wedge \ e_N, e \in E \\ & \wedge\ e = \kappa(\dots, \tau, \dots) \ \wedge \ \eta = (E, \mapsto)\}, \end{aligned}$$

$$\begin{aligned} X_\eta^{\rightarrow}(e) = \{e_X \mid\ & e_X = X(\dots, \tau, \dots) \ \wedge \ e \mapsto e_X \ \wedge \ e_X, e \in E \\ & \wedge\ e = \kappa(\dots, \tau, \dots) \ \wedge \ \eta = (E, \mapsto)\}. \end{aligned}$$

Now, for any two sets A and B of monitor events, we define the set $F(A, B)$ to be

$$\begin{aligned} F(A, B) = \{f \mid\ & f : A \to B \ \wedge \ f \text{ is injective} \\ & \wedge\ \forall a \in A.\ \forall b \in B.\ f(a) = b \implies match(a, b) \\ & \wedge\ \forall a \in A.\ \forall b \in B.\ match(a, b) \implies [f(a) = b \vee \ (\exists b' \in B.\ f(a) = b')]\}. \end{aligned}$$

In fact, $F(A, B)$ is comprised of those injective functions from A to B that map each event of A to a matching event of B, if there is any. It can easily be shown that for nonempty sets A and B such that there is at least one $a \in A$ and one $b \in B$ with $match(a, b)$, we have $F(A, B) \neq \emptyset$.

We may now define the set $NetN_\eta^{\leftarrow}(e)$ of enter events before a given event e in an abstract execution η that have not been matched with any exit event before e.

$$NetN_\eta^{\leftarrow}(e) = \bigcup_{f \in F(X_\eta^{\leftarrow}(e), N_\eta^{\leftarrow}(e))} \big(N_\eta^{\leftarrow}(e) - ran(f)\big),$$

where $ran(f)$ is the range of f. Similarly, the set $NetX^{\rightarrow}_{\eta}(e)$ of exit events after e in the abstract execution η that have not been matched with any enter event after e is defined by

$$NetX^{\rightarrow}_{\eta}(e) = \bigcup_{f \in F(N^{\rightarrow}_{\eta}(e), X^{\rightarrow}_{\eta}(e))} \big(X^{\rightarrow}_{\eta}(e) - ran(f)\big).$$

Given an abstract execution η, we define the causal relation ord_{η} on the events of η as

$$\begin{aligned} ord_{\eta}(e_1, e_2) \iff & ord'_{\eta}(e_1, e_2) \\ & \vee \exists (e_{N_1}, e_{X_1}, e_{N_2}, e_{X_2}) \in \mathcal{N}.\ (ord'_{\eta}(e_{N_1}, e_{X_2}) \wedge ord'_{\eta}(e_{N_2}, e_{X_1})), \end{aligned}$$

where $\mathcal{N} = NetN^{\leftarrow}_{\eta}(e_1) \times NetX^{\rightarrow}_{\eta}(e_1) \times NetN^{\leftarrow}_{\eta}(e_2) \times NetX^{\rightarrow}_{\eta}(e_2)$ and ord'_{η} is defined by

$$ord'_{\eta}(e_1, e_2) \iff \tau_1 = \tau_2 \ \vee \ (match(e_1, e_2) \ \wedge \ L_1 \neq \emptyset)$$

in which $\eta = (E, \mapsto)$, $e_1, e_2 \in E$, $e_1 = \kappa_1(id_1, \tau_1, L_1)$, and $e_2 = \kappa_2(id_2, \tau_2, L_2)$. As the definition suggests, the relation ord_{η} is defined among events of a thread and also among events of two different threads that are surrounded by some matching monitor events.

Before stating our *no data race* requirement in a formal manner, we need to present a couple of auxiliary definitions.

Definition 6. *Let $\eta = (E, \mapsto)$ be an abstract execution and $e_1, e_2 \in E$ such that $e_1 = \kappa_1(id_1, \tau_1, L_1)$ and $e_2 = \kappa_2(id_2, \tau_2, L_2)$. We say that e_1 and e_2 are conflicting and write $conf(e_1, e_2)$ if at least one of the events denotes a write access, the events are issued by different threads, and the events relate to the same memory location. That is,*

$$conf(e_1, e_2) \iff \kappa_1 = W \ \wedge \ (\kappa_2 = W \vee \kappa_2 = R) \ \wedge \ \tau_1 \neq \tau_2 \ \wedge \ L_1 \cap L_2 \neq \emptyset.$$

Definition 7. *An event $e = \kappa(id, \tau^m, L)$ is a multiple write, written $mwr(e)$, if it denotes a write access and the multiplicity of the thread creating e is $*$. More formally,*

$$mwr(e) \iff \kappa = W \wedge m = *.$$

Now, we define the no data race requirement for an abstract execution.

Definition 8. *An abstract execution $\eta = (E, \mapsto)$ is said to satisfy no data race requirement, written NORACE(η), iff*

$$(\forall e \in E.\ \neg mwr(e)) \ \wedge \ (\forall e_1, e_2 \in E.\ conf(e_1, e_2) \implies ord_{\eta}(e_1, e_2)).$$

It is easy to check that all the transition functions presented in Section 3 are monotone. Thus, there exist solutions to both systems of data flow equations. The aim of this section is to show that both of the solutions are equivalent from a data race detection point of view. First, we prove that a solution computed for the modular analysis is contained within the solution for the non-modular analysis. Next, we show that the result computed for modular analysis at final label of the method being analyzed is *safe*, i.e., is *NORACE*, if and only if the corresponding value associated with the non-modular analysis is safe. Note that, in what follows, η^{mod}_m, η^{int}_m denote the value of the solution of the modular and non-modular analyses at the final label of a method m, respectively.

Lemma 1. *Assume that c.mn be a method of a program P and that* $\eta_m^{\mathsf{mod}} \sqsubseteq \eta_m^{\mathsf{int}}$ *for all* $m \in mayCall(c.mn)$ *with* $(m, c.mn) \notin SCC$. *Then, we have* $\eta_{c.mn}^{\mathsf{mod}} \sqsubseteq \eta_{c.mn}^{\mathsf{int}}$.

Proof. A modular control flow graph for *c.mn* is a subgraph of the non-modular control flow graph for *c.mn*—the graph for modular analysis contains only elementary blocks corresponding to the body of *c.mn* and does not contain interprocedural flows. Furthermore, transition functions for all kinds of elementary blocks, except for method call sites, are equal in the two analyses. Hence, given a modular control flow graph (B, F) for the method in question, we have $CA_\bullet^{\mathsf{mod}}(\ell) = CA_\bullet^{\mathsf{int}}(\ell)$ for any $\ell \in B$ which is not a label of a method call site. In addition, the initial values in the two analyses are the same.

Now, we should prove that the set of events produced at call sites by the modular analysis as well as the order relation on those events are contained within the corresponding sets obtained from the non-modular analysis. First, let $m \in mayCall(c.mn)$ with $(m, c.mn) \notin SCC$. For m, the both analyses, instead of analyzing the body of the method, make use of the method summary associated with m, i.e., η_m^{mod} or η_m^{int}. For this case, the result follows directly from the hypothesis and the definition of method summaries. Second, suppose that $m \in mayCall(c.mn)$ with $(m, c.mn) \in SCC$. By the definition of the transition function for method call sites in the modular analysis, $\bot$ is chosen as the result of the analysis of m. As a result, the analysis of such a method does not lead to any event or event ordering. This completes the proof. □ □

By induction on the size of the set of methods of program P, i.e., $Methods_P$, similar to the proof of Theorem 3 at Appendix F of [21], and by using Lemma 1, we obtain the following result.

Lemma 2. *Let c.mn be a method of a program P. Then, we have* $\eta_{c.mn}^{\mathsf{mod}} \sqsubseteq \eta_{c.mn}^{\mathsf{int}}$.

Before presenting our next result, we need to point out a fact regarding the analyses. Let $\eta_M = (E_M, \mapsto_M)$ and $\eta_I = (E_I, \mapsto_I)$ be method summaries obtained through modular and non-modular analyses, respectively. According to Lemma 2, we have $\eta_M \sqsubseteq \eta_I$. Let $E = E_I - E_M$ be the set of all events that can be obtained from the non-modular analysis but not from the modular analysis. By a careful inspection of transition functions in the two analyses, it follows that

$$\forall e \in E.\ \exists e' \in E_M.\ \exists s \in L_{call}^+.\ \mathit{getEID}(e) = s \cdot \mathit{getEID}(e'), \tag{9}$$

and

$$\forall e \in E.\ \exists e' \in E_M.\ \exists s \in L_{call}^+.\ \mathit{getTID}(e) = s \cdot \mathit{getTID}(e'). \tag{10}$$

In these equations, centered dot denotes the string concatenation operator and $+$ denotes the closure operator that forms the set of all possible non-empty finite strings out of the alphabet L_{call}. Furthermore, *getEID* and *getTID* are functions used to obtain the identifier and the thread identifier of an abstract event. Moreover, L_{call} represents the set of all labels associated with a method call site in *c.mn* and in the set of methods that are, directly or indirectly, called by *c.mn*.

Theorem 1. *Let $\eta_M = (E_M, \mapsto_M)$ and $\eta_I = (E_I, \mapsto_I)$ be summaries for a method which are obtained form the modular and the non-modular analyses, respectively. Then, we have NORACE$(\eta_M) \iff$ NORACE(η_I).*

Proof. By Lemma 2, we have $E_M \subseteq E_I$ and $\mapsto_M \subseteq \mapsto_I$. This immediately implies one part of the desired result, i.e., NORACE$(\eta_I) \implies$ NORACE(η_M). For the opposite direction, assume that $e \in E_I$, $e' \in E_M$, and $\exists s \in L^+_{call}$. *getEID*$(e) = s{\cdot}$*getEID*(e'). Since the function *mayPointsTo* used in the two analyses is the same, we conclude that the third components of e and e' are equal. Furthermore, according to the definition of label multiplicity, that is the same in both analyses, and transition functions corresponding to thread creation sites, assignment and dereference statements, and method call sites, that are either equal in both analyses or behave in the same way, in determining the event kinds and thread multiplicities, we know that e and e' have the same multiplicity and kind. Hence, we have

$$(\forall e \in E_M.\ \neg mwr(e)) \implies (\forall e \in E_I.\ \neg mwr(e)). \tag{11}$$

Finally, by inspecting the definition of the transition functions of the two analyses, we conclude that the order of events in a thread determined by both analyses is the same. Because the third components of e and e' are equal, the monitor events obtained from the two analyses affect the same lock sets. Therefore, according to the definition of the relation ord_η, the statement

$$\forall e_1, e_2 \in E_M.\ conf(e_1, e_2) \implies ord_{\eta_M}(e_1, e_2)$$

implies

$$\forall e_1, e_2 \in E_I.\ conf(e_1, e_2) \implies ord_{\eta_I}(e_1, e_2).$$

This result, in conjunction with (11), yields our desired result. □ □

4.1 Another Synchronization Mechanism

Using lock sets to determine temporal relations among accesses made by different threads limits synchronization mechanisms to those based on locking and to those that can be simulated by locking [52, 41]. Our concurrency analyses, however, associate with each synchronization instruction an abstract event, thereby being able to handle a wide variety of synchronization idioms. In this section, we demonstrate this fact through an example that shows how the effects of semaphores can be incorporated into our analysis method.

Let `r` be a resource that at most two threads are allowed to use it at the same time. Suppose that the resource is to be used by four concurrent threads, as shown is Figure 3. Four instances of the thread `T` is created by using `parbegin` statement. The threads are synchronized by the semaphore `s`, so that at most two threads may access `r` in parallel. This means that the program is well-behaved.

For this resource, a data race is defined by three conflicting events that are not ordered through the happened-before relation. We shall not dwell upon what we mean by three conflicting events, for it depends on the nature of the resource. Figure 4 illustrates an abstract execution of the program, namely sequences of events for each thread. Note that the abstract event kind P denotes a `wait` action, while the kind V denotes a `signal` action. Sequences of events

```
semaphore s = 2;
resource r;

thread T = {
    wait(s);
    /*use the resource r*/
    signal(s);
}

parbegin T1, T2, T3, T4;
```

Figure 3: An example program making use of semaphores for synchronization.

$$
\begin{array}{ll}
\mathtt{T}_1: & e_1 = P(\mathtt{T}_1, s), \ldots, e_2 = V(\mathtt{T}_1, s), \\
\mathtt{T}_2: & e_3 = P(\mathtt{T}_2, s), \ldots, e_4 = V(\mathtt{T}_2, s), \\
\mathtt{T}_3: & e_5 = P(\mathtt{T}_3, s), \ldots, e_6 = V(\mathtt{T}_3, s), \\
\mathtt{T}_4: & e_7 = P(\mathtt{T}_4, s), \ldots, e_8 = V(\mathtt{T}_4, s).
\end{array}
$$

Figure 4: Abstract events corresponding to each thread in the program of Figure 3.

corresponding to manipulations of `r` are denoted by three consecutive dots in each trail of events. Now, in order to verify the behavior of the program, the only thing that is left to do is to define what is meant by a safe abstract execution. For this case, according to the semantics of a semaphore with initial value 2, we may informally argue that since both e_2 and e_4 are "paired" with e_5 and e_7 (similarly, e_6 and e_8 match e_1 and e_3) and since the events of a thread are ordered according to the order guaranteed by the program, the events corresponding to accesses to the resource by $\mathtt{T}_1$ and $\mathtt{T}_2$ match those reflecting the accesses to the same resource by the threads $\mathtt{T}_3$ and $\mathtt{T}_4$. Because there do not exist three unordered conflicting events, the abstract execution is safe, i.e., the program is free of data races.

5 Experimental Results

We have implemented our ideas by using the Soot framework [43]. The Soot framework can be used to parse standard Java programs. Since it provides an off-the-shelf call graph analysis and also a fast points-to analysis, it serves as a good choice for quickly implementing a prototype demonstrating the effectiveness of our concurrency analysis. In this section, we present the experimental results obtained from applying our prototype tool on a suit of Java programs. As we shall see, the results demonstrate that the tool is scalable and its precision is comparable to that of leading precise data race detection tools.

The implemented tool performs the analysis in four stages: (1) extracting information needed for subsequent analyses, (2) points-to analysis, (3) thread-escape analysis, and (4) concurrency analysis. In order to speed up the development process, we have implemented our points-to and thread-escape analyses using Datalog [2]. To solve the resulting equations, we also make use of bddbddb [53]. The details of our implementation of the summary-based points-to analysis can be found in the online appendix of this paper [21], which also contains

guidelines for making the analysis more precise when it is applied to methods in isolation.

In the first stage, we compute input relations for the second and third Datalog-based stages. In the next stage, we run our points-to analysis. Through several experiments on the implemented points-to analysis, we realize that the only feasible choice for context sensitivity is 1-object sensitivity. Even in 1-object sensitivity, it takes a while to analyze large programs. A little thought will show that, for a programmer, precision in data race detection matters for the objects that are created in the application code rather than in the library code. Thus, we do not need to analyze the whole program (application + library) using a context sensitive analysis method. As a result, we decide on a 2-stage (hybrid) points-to analysis.

In the first stage of our points-to analysis, we analyze the whole program by using a fast, context-insensitive points-to analysis. To do so, we employ Spark, which comes with Soot as an off-the-shelf analysis. In the subsequent stage, we drop the points-to information computed for the application code and recompute it using our 1-object-sensitive points-to analysis. In this stage, in the case of any call to a method from a library class, we ignore parameters passed to the method and query previous stage for possible returned objects or thrown exceptions. In other words, instead of analyzing each called library method, we ask the context-insensitive analysis to give us possible objects returned by the method. Note that ignoring parameters passed to library methods does not compromise soundness, for we have already taken into account the effect of passing the same parameters to the methods. Our experiments show that this technique results in a several times faster analysis. For instance, an analysis which takes 4 hours and 30 minutes turns into an analysis taking 40 minutes. It is worth noting that, feeding the results of the hybrid points-to analysis to the subsequent data race detection algorithm does not result in a perceptible loss of precision—the number of false positives for some of benchmark programs increased by just 2 or 3. Finally, it should be noted that using the hybrid points-to analysis halves the number of false positives issued by the tool in comparison to when it is accompanied by a context-insensitive points-to analysis like Spark.

As further technical details of our points-to analysis, we may mention that it is a "may," "flow-insensitive," and "inclusion-based" (a.k.a. Andersen-style) analysis. Furthermore, the call graph is computed on-the-fly (making the analysis more precise) and no type-filter is employed (making the analysis less precise but more faster). This stage is also able to identify multiplicity of the objects. In general, by assuming that the subexpressions of the input program are uniquely labeled, it is able to determine the multiplicity of program labels. That is, we may identify whether an object/thread is created by a thread of multiplicity of m, inside a loop, or by a non-recursive method (See Section 2).

Our escape analysis (stage 3) is based on what has been presented in [10]; for the sake of brevity, we avoid describing the analysis.

Finally, the tool performs concurrency analysis at the forth stage. It starts with constructing the abstract execution graph of the input program and ends with categorizing and pretty-printing warning messages resulted from traversing the abstract execution for finding multiple-write events (write events caused by threads with multiplicity of *) or conflicting access pairs. We query points-to information from points-to analysis while constructing abstract execution graph and query escape information from stage 3 while traversing the graph.

In this paper, by a lazy data structure, we mean a data structure such as a map, a list, or a set backed by some secondary storage. There are a number of free, open-source libraries such as MapDB [23] to bring functionality to the idea of lazy data structures. Nevertheless, we have our own implementations that perform even better in some cases. As it is expected, we observe that when lazy data structures are used everywhere, i.e., in computing abstract execution graphs, in storing computed method summaries, and in the implementation of the concurrency analysis, we need only a *constant* amount of heap space to complete the analysis regardless of the size and shape of the call graph of the input program. However, due to frequent accesses to the secondary storage, which is usually far slower than main memory, the tool suffers from significant performance penalties.

We devised a solution to this problem by (1) parallelizing the analysis of each method, (2) caching data read from the secondary memory, and more importantly (3) caching and delaying writes to the secondary memory until a certain limit has been reached[3]. Unfortunately, since Soot does not support concurrent iteration over control flow graphs for methods[4], we could not fully take advantage of this technique. We also tried to use compressed MemoryDBs (a map which is stored in the main memory and which serializes and compresses its keys and values) as offered by MapDB to implement abstract execution graph internals and to use a FileDB (FileDBs are reminiscent of MemoryDBs but are stored in a file) to store computed method summaries. This makes the analysis faster and uses less than 3 GB of heap space to complete the analysis of our largest benchmark program. These two cases almost completely alleviate the problem of augmented method summaries, i.e., the problem which arises due to composition of huge method summaries that result from successive in-lining of summaries of callee methods. Finally, we tried to use standard data-structures of Java library to implement abstract execution graphs and to use a FileDB just to store computed method summaries. In this case, we have the problem of augmented method summaries, but it is much more faster than the other three configurations.

Our current implementation supports neither Java's native methods nor reflection. In the case of native methods, we could implement our analysis algorithm for some particular machine codes. Hence, we can analyze shared libraries and save computed summaries in a database and upon call from the client code we may pick the appropriate method summary from the database. For Java reflection, we may use tools such as Tamiflex [7] that is designed to work along with Soot. Implementing most recent reflection analysis algorithms, such as [25], also seems promising.

We have conducted our experiments on a computer running Linux 3.13 with 2.5 GHz of CPU clock frequency and 8 GB of RAM. Furthermore, the allocated memory space for Java Virtual Machine is 4 GB. The first column of Table 1 lists the set of benchmark programs used in our experiments. The benchmark programs are gathered among those that are tested by different researchers before and cover a wide variety of (real-world) synchronization idioms. Details of the experiments are also given in the table. In general, data race warnings reported by our tool may be divided into two categories: (1) data races caused

[3]Due to the structure of some secondary memories, such as hard disks, writing say a 4-KB block is accomplished faster than writing four 1-KB smaller blocks.

[4]Eric Bodden, personal communication.

by threads with multiplicity of * that are called *multiple-write* warnings and (2) data races caused by two statically distinguishable concurrent threads or *conflicting-access* warnings.

Program	**Size (LOC)**	**Warnings**	
		Multiple-Write	*Conflicting-Access*
Example	30	0	0
Philo	84	0	2
Sor	265	0	2
Elevator	531	3	4
TSP	706	3	8
Weblech	1903	2	3
Clanbomber	2630	0	19
hedc	24936	1	6
jEdit	178218	0	0
jFreeChart	311119	0	0
MegaMek	411172	0	0

Table 1: The size of each benchmark program together with the number of warnings issued by our tool.

The tool reports no warning for large programs such as jEdit, jfreechart, and MegaMek. It is for this reason that these programs use simple concurrency patterns, i.e., threads which access only thread-local data, or simple inter-thread communication patterns. We have included these programs in our list so as to show that our tool is able to scale to large programs. The higher number of warnings in the case of Clanbomber is due to the fact that the program is a multi-threaded program, altering shared data, without any synchronization mechanism employed. The programmer of that has been a novice and completely unaware of data races.

In order to compare the precision of our tool with two precise, well-known systems, let us call the tool presented in [41] ETH-Tool—a static analysis tool that aims at finding access conflicts so as to instrument subject programs and reduce run-time overhead and the number of false positives issued by its dynamic part [40]. In this comparison, we should note that ETH-Tool is not sound, and thus, neglects a number of suspicious cases. Chord [32] is another tool that we are going to compare with our tool. It is the leading sound and precise data race detection tool for Java. Table 2 lists the number of warnings issued by ETH-Tool and Chord for each of the benchmark programs.

	Example	Philo	Sor	Elevator	TSP	Weblech	hedc
Chord	?	0	0	0	4	2	4
ETH-Tool	0	1	1	4	3	?	36

Table 2: The number of warning messages issued by ETH-Tool and Chord for each of the benchmark programs; a question mark represents lack of information.

A distinguishing feature of our tool is its modularity. This means that the most complex computation during an analysis, which is the formation and solving data flow equations, is done separately for each method of a program. This makes the tool much more scalable in comparison with tools like Chord—an

experiment shows that the tool is unable to analyze large programs in a reasonable time and space limits [30]. Our prototype tool is fully implemented in Java and, in most cases, we have used naive data structures. The tool would perform even better if we tune the performance of the tool through implementing a number of frequently called methods in a native language and reducing the accesses to disk through more sophisticated caching techniques.

6 Related Work

The two main classes of methods for data race detection are *dynamic* and *static* analyses. In this section, we give a brief overview of a number of proposed solutions in each class and compare them with the method presented in this paper.

6.1 Dynamic Race Detection

Dynamic data race detection methods usually build on the notions of so-called happened-before relation and lock sets [46, 40]. They may also be hybrid making use of the both of the techniques [35]. A wide variety of related techniques can also be found in recent patents. For example, Erickson and Musuvathi [17] have proposed a dynamic data race detection technique which randomly pairs a number of memory access instructions of the program with particular instructions called breakpoints. Breakpoints, once triggered, pause the execution of the current thread to monitor accesses made by other threads. They may also compare the values of accessed memory locations before and after the pause. In this way, the effects of conflicting instructions on current accesses can be ascertained.

The branch of research concerning dynamic methods has been advanced substantially in recent years. There exist algorithms based on an extended version of the happened-before relation that, at the cost of a negligible number of false positives, can detect all possible data races in program executions [47]. There are also generalized tools that detect data races and atomicity violations [26] and other types of race conditions that may occur at the level of library interfaces [**?**]. Managing run-time overhead of dynamic data race detection techniques, e.g., efficient construction of vector clocks, is a primary concern which has constantly been concerned in recent years [40, 22, 12, **?**]. Unfortunately, dynamic methods are unable to detect all data races in a program. That is, such methods are generally unsound [39].

6.2 Static Race Detection

Static data race detection methods do not need to run the program. Instead, they check the program text for possible data races. Following [39], we divide static data race detection methods into three main classes: pragmatic methods, type-based methods, and methods based on data flow analysis.

6.2.1 Pragmatic Methods

A pragmatic data race detection method uses a heuristic algorithm to find bug patterns in a given program text. Such patterns are indeed violations of pre-

scribed programming practice. Pragmatic data race detection tools are neither sound nor complete, i.e., a reported case is not always a genuine data race [39], and are usually application specific. Since a fast and statistically effective tool is required in most industrial applications, pragmatic data race detection tools have become very popular. The tool FindBugs [4] is an example general-purpose bug finder tool for Java programming language which has been successfully applied to a number of industrial applications. RacerX [16] is another example which is a fast and surprisingly precise data race detection tool for large system programs written in C.

6.2.2 Type-based Methods

From a data race detection point of view, the most desirable characteristic of type-based methods [9] is their modularity [5]. The idea of using type systems to prevent data races goes back to attempts made in devising type-safe concurrent object calculi [19, 18]. Later, the idea extended to the analysis of real-life Java programs [20]. Since the resulting Java language required programmers to annotate their programs with appropriate "guarded-by" annotations, Flanagan and Fruend proposed the use of inference algorithms. They later realized that such an inference problem for Java and similar programming languages is NP-Complete [1]. As a research in this line of research, Boyapati and Rinard [8] employed generic language constructs and proposed a light-weight annotation inference algorithm, thereby reducing the annotation burden of the Java language of [20]. Their work served as a milestone on which future tools were based. In particular, Sasturkar et al. [45] introduced a type-system ensuring atomicity of methods in a Java program. Despite the endeavors made, the annotation burden and intractability of annotation inference algorithms are still two main obstacles in type-based approaches to data race detection. Locksmith [37] is a data race prevention system for C. The tool consists of several types and an effect system. The tool is precise and does not bring about any annotation burden. But since sets of constraints associated with type and effect systems involve the whole program, the tool is not scalable.

Atomicity is a criterion for the correctness of concurrent programs. However, it is a strong requirement when the problem is to shun data races [38]. Autolocker [29] is a compiler for an extended version of the C programming language that automatically produces safe (deadlock-free) sequences of synchronization operations to guard atomic sections. Although the resulting programming language incurs a high amount of annotations, the work shares similarities with a contemporary work which has mitigated the problem. Vaziri et al. [51] have identified a complete set of scenarios the absence of which guarantee not only data race freedom in a traditional sense, but also avoids high-level data races such as stale-value errors and inconsistent views.

The notion of atomic sets, which first introduced in [51], led to the introduction of a data-centric approach to synchronization that flourished by the development of a dialect of Java called AJ [14]. The compiler for AJ minimizes the annotation burden of synchronization-related tasks by automatically adding synchronization operations according to data-centric annotations. The automated inference of needed synchronization operations was not perfect thus far and the compilers usually incurred perceptible performance penalty. Nevertheless, these works instigated an active line of research on inference of atomic

set annotations [13] and extending data-centric synchronization approach so as to prevent other synchronization defects such as deadlocks [28]. Atomicity and high-level data race freedom have also been attacked by a combination of concurrency and typestate analyses [49] in a recent work [56].

6.2.3 Methods Based on Data Flow Analysis

Here, we discuss data race detection methods based on data flow analysis [34]. By using a combination of concurrency and alias analyses, one can implement a data race detection algorithm [39]. The idea was first stated in [15] where the proposed concurrency analysis was modular and the method was able to analyze recursively defined Ada procedures. The same idea formed the basis for an unsound data race detection method for Java [41]. Because concurrency analyses are language-specific, theoretical results appeared in [15] are not applicable to other languages. Furthermore, there is no formal proof of soundness for the proposed method.

Choi et al. [11] present the notion of inter-thread control flow graph (*ICFG*) which is similar to our notion of abstract executions. In fact, the set of events and the set of pairs in the second component of an abstract execution correspond to the set of nodes and edges in the *ICFG*, respectively. Abstract events in an abstract execution, however, are determined in such a way that contain much more information than the nodes of *ICFG*. Our concurrency analysis is context-sensitive and distinguishes different call sites of a method, so abstract events contain information about call sites. It is worth noting that such an additional information lets a programmer to distinguish those call sites of a method that may cause a data race from those call sites that are not suspicious. Hence, the programmer can use synchronization instructions, computational complexity of which is not negligible for most applications, only when they are actually needed. The solution proposed in [11] does not scale and lacks a formal proof of soundness.

The method presented by von Praun and Gross [41] is the closest work to ours. Their algorithm leverages on the notions of event and event ordering. Indeed, abstract executions are similar to the object use graphs (OUGs) except that abstract executions allow us to take into account a wide variety of synchronization idioms and does not suffer from unnecessary edges. Their race detection technique is not sound for several reasons. We explain the most important reason. Events in OUGs are attributed to abstract threads and each abstract thread is represented by a node of the heap shape graph (HSG) constructed for the target program. Since HSGs are constructed through an algorithm based on equivalence sets [48, 42], there is a possibility of merging two or more nodes yielding another node. Hence, two or more completely unrelated concrete threads could be mistakenly considered as an abstract thread. In this paper, instead of using nodes in HSGs, we use thread creation sites to determine abstract threads. By using multiplicity information, we also prevent the coalescence of the effects of the concrete threads created inside loops or recursions from being crossed over. The system presented in [41] is scalable, but the employed points-to analysis is not precise enough so that the system will produce false negatives. In this paper, instead of a points-to analysis based on equivalence sets, we have used a more precise context-sensitive analysis similar to that presented in [54, 44].

Voung et al. present RELAY [52], a scalable static data race detection system for C. In order to achieve scalability, the tool employs function summaries inspired by Sharir and Pnueli's functional approach to interprocedural analysis, taking different names in the literature [54, 42, 16, 41]. The notion of method summaries, used in this paper, is similar to function summaries. Method summaries, however, contain temporal information about conflicting events, and hence, yield more precise information. In this paper, we have shown how to achieve scalability along with soundness and precision. The use of semaphores in a target program is the primary source of imprecision in RELAY. We have observed that by attributing distinct events to each synchronization construct, the effect of semaphores could be precisely taken into account. The points-to analysis proposed in RELAY does not deal with interprocedural flows, and thus, it is neither sound nor precise [37]. In this paper, we have shown how interprocedural flows can be taken into account through a modular points-to analysis.

In [32], Naik and Aiken presented CHORD, a sound, precise, and remarkably fast data race detection tool for Java. The notion of statement in CHORD corresponds to the notion of events in our work. Since CHORD analyzes a given program whole, it does not scale to large programs [39, 30]. In contrast to CHORD, to analyze open programs, our algorithms do not need to synthesize a client program. It is for this reason that method summaries contain sufficient information about access patterns of methods, thereby enabling the tool to provide useful information.

7 Conclusion

The goal is to design a sound, scalable, and precise data race detection system. In order to attain soundness, we keep track of references created interprocedurally. Furthermore, we use thread creation sites to identify abstract threads. By using thread multiplicities, we can also avoid the coalescence of the effects of those threads created multiple times at a thread creation site. We bring scalability through modularity. To achieve modularity, we analyze each method in isolation, obtain a parametric summary for the method, and instantiate the resulting summary at each call site of the method. It is proven that the proposed system is sound, something absent in earlier solutions. In doing so, we prove that the soundness of the proposed non-modular analysis implies the soundness of the modular analysis.

By introducing some annotations at method declaration points, we also devise an improved points-to analysis which is able to deliver more precise information when methods are analyzed in isolation. Our experimental results reveal details about the behavior of our modular analysis based on method summaries and show that by using a combination of context-sensitive and context-insensitive points-to analyses we may achieve both precision and good performance together. It is also observed that through an appropriate use of lazy data structures, we can implement a summary-based analysis algorithm that consumes only a constant amount of heap space regardless of the size of programs.

References

[1] Abadi, M., Flanagan, C., Freund, S.N.: Types for Safe Locking: Static Race Detection for Java. ACM Transactions on Programming Languages and Systems **28**(2), 207–255 (2006). DOI 10.1145/1119479.1119480. URL http://doi.acm.org/10.1145/1119479.1119480

[2] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)

[3] Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. Addison-Wesley Professional (2005). 4th Edition

[4] Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Using Static Analysis to Find Bugs. IEEE Software **25**(5), 22–29 (2008). DOI http://doi.ieeecomputersociety.org/10.1109/MS.2008.130

[5] Beckman, N.E.: A Survey of Methods for Preventing Race Conditions (2006). Technical Report

[6] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. Communications of the ACM **53**(2), 66–75 (2010). DOI 10.1145/1646353.1646374. URL http://doi.acm.org/10.1145/1646353.1646374

[7] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 241–250. ACM, New York, NY, USA (2011). DOI 10.1145/1985793.1985827. URL http://doi.acm.org/10.1145/1985793.1985827

[8] Boyapati, C., Rinard, M.: A Parameterized Type System for Race-free Java Programs. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '01, pp. 56–69. ACM, New York, NY, USA (2001). DOI 10.1145/504282.504287. URL http://doi.acm.org/10.1145/504282.504287

[9] Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys **17**(4), 471–523 (1985). DOI 10.1145/6041.6042. URL http://doi.acm.org/10.1145/6041.6042

[10] Choi, J.D., Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P.: Stack allocation and synchronization optimizations for java using escape analysis. ACM Trans. Program. Lang. Syst. **25**(6), 876–910 (2003). DOI 10.1145/945885.945892. URL http://doi.acm.org/10.1145/945885.945892

[11] Choi, J.d., Loginov, A., Sarkar, V.: Static datarace analysis for multi-threaded object-oriented programs (2001). Technical Report

[12] Dimitrov, D., Vechev, M., Sarkar, V.: Race Detection in Two Dimensions. In: Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA '15, pp. 101–110. ACM, New York, NY, USA (2015). DOI 10.1145/2755573.2755601. URL http://doi.acm.org/10.1145/2755573.2755601

[13] Dinges, P., Charalambides, M., Agha, G.: Automated Inference of Atomic Sets for Safe Concurrent Execution. In: Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '13, pp. 1–8. ACM, New York, NY, USA (2013). DOI 10.1145/2462029.2462030. URL http://doi.acm.org/10.1145/2462029.2462030

[14] Dolby, J., Hammer, C., Marino, D., Tip, F., Vaziri, M., Vitek, J.: A Data-centric Approach to Synchronization. ACM Trans. Program. Lang. Syst. **34**(1), 4:1–4:48 (2012). DOI 10.1145/2160910.2160913. URL http://doi.acm.org/10.1145/2160910.2160913

[15] Duesterwald, E., Soffa, M.L.: Concurrency Analysis in the Presence of Procedures Using a Data-flow Framework. In: Proceedings of the Symposium on Testing, Analysis, and Verification, TAV4, pp. 36–48. ACM, New York, NY, USA (1991). DOI 10.1145/120807.120811. URL http://doi.acm.org/10.1145/120807.120811

[16] Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pp. 237–252. ACM, New York, NY, USA (2003). DOI 10.1145/945445.945468. URL http://doi.acm.org/10.1145/945445.945468

[17] Erickson, J., Musuvathi, M.: Data Race Detection (2014). US Patent 8,813,038

[18] Flanagan, C., Abadi, M.: Object Types against Races. In: J. Baeten, S. Mauw (eds.) CONCUR'99 Concurrency Theory, *Lecture Notes in Computer Science*, vol. 1664, pp. 288–303. Springer Berlin Heidelberg (1999). DOI 10.1007/3-540-48320-9_21. URL http://dx.doi.org/10.1007/3-540-48320-9_21

[19] Flanagan, C., Abadi, M.: Types for Safe Locking. In: S. Swierstra (ed.) Programming Languages and Systems, *Lecture Notes in Computer Science*, vol. 1576, pp. 91–108. Springer Berlin Heidelberg (1999). DOI 10.1007/3-540-49099-X_7. URL http://dx.doi.org/10.1007/3-540-49099-X_7

[20] Flanagan, C., Freund, S.N.: Type-based Race Detection for Java. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, pp. 219–232. ACM, New York, NY, USA (2000). DOI 10.1145/349299.349328. URL http://doi.acm.org/10.1145/349299.349328

[21] Ghanbari, A., Fallah, M.S.: A Sound and Scalable Method for Static Data Race Detection in Java Programs. online (2015). http://ceit.aut.ac.ir/formalsecurity/autjava/

[22] Kasikci, B., Zamfir, C., Candea, G.: RaceMob: Crowdsourced Data Race Detection. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, pp. 406–422. ACM, New York, NY, USA (2013). DOI 10.1145/2517349.2522736. URL http://doi.acm.org/10.1145/2517349.2522736

[23] Kotek, J.: Mapdb: Database engine. online (2016). http://www.mapdb.org/

[24] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM **21**(7), 558–565 (1978). DOI 10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563

[25] Li, Y., Tan, T., Xue, J.: Effective soundness-guided reflection analysis. In: S. Blazy, T. Jensen (eds.) Static Analysis, *Lecture Notes in Computer Science*, vol. 9291, pp. 162–180. Springer Berlin Heidelberg (2015). DOI 10.1007/978-3-662-48288-9_10. URL http://dx.doi.org/10.1007/978-3-662-48288-9_10

[26] Lu, S., Park, S., Zhou, Y.: Detecting Concurrency Bugs from the Perspectives of Synchronization Intentions. Parallel and Distributed Systems, IEEE Transactions on **23**(6), 1060–1072 (2012). DOI 10.1109/TPDS.2011.254

[27] Mangal, R., Naik, M., Yang, H.: A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join. In: Z. Shao (ed.) Programming Languages and Systems, *Lecture Notes in Computer Science*, vol. 8410, pp. 513–533. Springer Berlin Heidelberg (2014). DOI 10.1007/978-3-642-54833-8_27. URL http://dx.doi.org/10.1007/978-3-642-54833-8_27

[28] Marino, D., Hammer, C., Dolby, J., Vaziri, M., Tip, F., Vitek, J.: Detecting Deadlock in Programs with Data-centric Synchronization. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 322–331. IEEE Press, Piscataway, NJ, USA (2013). URL http://dl.acm.org/citation.cfm?id=2486788.2486831

[29] McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: Synchronization Inference for Atomic Sections. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, pp. 346–358. ACM, New York, NY, USA (2006). DOI 10.1145/1111037.1111068. URL http://doi.acm.org/10.1145/1111037.1111068

[30] Milanova, A., Huang, W.: Static Object Race Detection. In: Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings, pp. 255–271 (2011). DOI 10.1007/978-3-642-25318-8_20. URL http://dx.doi.org/10.1007/978-3-642-25318-8_20

[31] Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. **14**(1), 1–41 (2005). DOI 10.1145/1044834.1044835. URL http://doi.acm.org/10.1145/1044834.1044835

[32] Naik, M., Aiken, A.: Conditional Must Not Aliasing for Static Race Detection. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '07, pp. 327–338. ACM, New York, NY, USA (2007). DOI 10.1145/1190216.1190265. URL http://doi.acm.org/10.1145/1190216.1190265

[33] Naik, M., Aiken, A., Whaley, J.: Effective Static Race Detection for Java. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06, pp. 308–319. ACM, New York, NY, USA (2006). DOI 10.1145/1133981.1134018. URL http://doi.acm.org/10.1145/1133981.1134018

[34] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)

[35] O'Callahan, R., Choi, J.D.: Hybrid Dynamic Data Race Detection. In: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '03, pp. 167–178. ACM, New York, NY, USA (2003). DOI 10.1145/781498.781528. URL http://doi.acm.org/10.1145/781498.781528

[36] Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)

[37] Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Practical Static Race Detection for C. ACM Transactions on Programming Languages and Systems **33**(1), 3 (2011). DOI 10.1145/1889997.1890000. URL http://doi.acm.org/10.1145/1889997.1890000

[38] von Praun, C.: Race Conditions. In: D.A. Padua (ed.) Encyclopedia of Parallel Computing, pp. 1691–1697. Springer US (2011)

[39] von Praun, C.: Race Detection Techniques. In: D.A. Padua (ed.) Encyclopedia of Parallel Computing, pp. 1697–1706. Springer US (2011). DOI 10.1007/978-0-387-09766-4_38. URL http://dx.doi.org/10.1007/978-0-387-09766-4_38

[40] von Praun, C., Gross, T.R.: Object Race Detection. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '01, pp. 70–82. ACM, New York, NY, USA (2001). DOI 10.1145/504282.504288. URL http://doi.acm.org/10.1145/504282.504288

[41] von Praun, C., Gross, T.R.: Static Conflict Analysis for Multi-threaded Object-oriented Programs. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03, pp. 115–128. ACM, New York, NY, USA (2003). DOI 10.1145/781131.781145. URL http://doi.acm.org/10.1145/781131.781145

[42] Ruf, E.: Effective Synchronization Removal for Java. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, pp. 208–218. ACM, New York, NY, USA (2000). DOI 10.1145/349299.349327. URL http://doi.acm.org/10.1145/349299.349327

[43] Sable: Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/, [accessed 18-Sep-2014]

[44] Sagiv, M., Reps, T., Wilhelm, R.: Solving Shape-Analysis Problems in Languages with Destructive Updates. ACM Transactions on Programming Languages and Systems **20**(1), 1–50 (1998)

[45] Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.D.: Automated Type-based Analysis of Data Races and Atomicity. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '05, pp. 83–94. ACM, New York, NY, USA (2005). DOI 10.1145/1065944.1065956. URL http://doi.acm.org/10.1145/1065944.1065956

[46] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems **15**(4), 391–411 (1997). DOI 10.1145/265924.265927. URL http://doi.acm.org/10.1145/265924.265927

[47] Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., Flanagan, C.: Sound Predictive Race Detection in Polynomial Time. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, pp. 387–400. ACM, New York, NY, USA (2012). DOI 10.1145/2103656.2103702. URL http://doi.acm.org/10.1145/2103656.2103702

[48] Steensgaard, B.: Points-to Analysis in Almost Linear Time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96, pp. 32–41. ACM, New York, NY, USA (1996). DOI 10.1145/237721.237727. URL http://doi.acm.org/10.1145/237721.237727

[49] Strom, R.E., Yemini, S.: Typestate: A Programming Language Concept for Enhancing Software Reliability. IEEE Transactions on Software Engineering **12**(1), 157–171 (1986). DOI 10.1109/TSE.1986.6312929. URL http://dx.doi.org/10.1109/TSE.1986.6312929

[50] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallèe-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical Virtual Method Call Resolution for Java. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00, pp. 264–280. ACM, New York, NY, USA (2000)

[51] Vaziri, M., Tip, F., Dolby, J.: Associating Synchronization Constraints with Data in an Object-oriented Language. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, pp. 334–345. ACM, New York, NY, USA (2006). DOI 10.1145/1111037.1111067. URL http://doi.acm.org/10.1145/1111037.1111067

[52] Voung, J.W., Jhala, R., Lerner, S.: RELAY: Static Race Detection on Millions of Lines of Code. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07,

pp. 205–214. ACM, New York, NY, USA (2007). DOI 10.1145/1287624.1287654. URL http://doi.acm.org/10.1145/1287624.1287654

[53] Whaley, J.: Context-Sensitive Pointer Analysis Using Binary Decision Diagrams. Ph.D. thesis, Stanford University (2007)

[54] Whaley, J., Rinard, M.: Compositional Pointer and Escape Analysis for Java Programs. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '99, pp. 187–206. ACM, New York, NY, USA (1999). DOI 10.1145/320384.320400. URL http://doi.acm.org/10.1145/320384.320400

[55] Yan, D., Xu, G., Rountev, A.: Rethinking soot for summary-based whole-program analysis. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP '12, pp. 9–14. ACM, New York, NY, USA (2012). DOI 10.1145/2259051.2259053. URL http://doi.acm.org/10.1145/2259051.2259053

[56] Yang, Y., Gringauze, A., Wu, D., Rohde, H.: Detecting Data Race and Atomicity Violation via Typestate-Guided Static Analysis (2013). US Patent 8,510,722