Fall 2003 CS263 Course Project

Survey: Race Detection and Atomicity Checking

Feng Zhou (zf@cs.berkeley.edu)

December 16, 2003

1 Introduction

Concurrent programs are hard to write and debug because of the inherent concurrency and indeterminism. Bugs in these programs are often hard to find in the testing process. When they do show up later in production systems, they are not easy to pinpoint because a lot of them are hard to reproduce. Therefore, automated tools that are able to find these bugs are of great value in relieving programmers from identifying these bugs manually. In this paper we survey some of the recent work in this area, focusing on static language-based tools that do not actually run the application. Static tools have the benefit of considering different execution paths more exhaustively and can potentially be sound, i.e. being able to prove the bug-freeness of programs.

Apart from static tools, other kinds of concurrency bug finding tools include dynamic tools and post-mortem tools. Dynamic tools, e.g. Eraser [8], tracks program execution and report potential concurrency bugs if the program does not follow certain concurrency control disciplines (e.g. locking). These tools often produce relevant results with low false positive rate. But they cannot possibly cover all execution paths and therefore cannot be sound. Post mortem tools detect bugs by analyzing execution traces collected during execution. Although they introduces less overhead than dynamic tools, they still suffer the same limitations.

Concurrency bugs can be of several forms. The most extensively studied one is race conditions. Races occur when shared variables are accessed concurrently by multiple threads of control, at least one of them doing a write, without proper synchronization. The final values of the variables will depend on the interleaving of the threads and thus normal assumptions of serial programming, essentially that a variable will not change value if the current thread does not change it, do not hold any more. If the programmer make use of these assumptions, insidious bugs are introduced. However, it is important to note that not all races cause bugs. Races can be benign if, e.g. only reads of a single variable are done without synchronization, or if only increment and decrement are done to a particular shared variable and hardware guarantees atomicity of these operations.

Races are actually only part of the reasons for so-called atomicity bugs [4]. In general, atomicity bugs happen when the serial assumptions that the correctness of the program is based on is broken. These assumptions can be broken even if no races exist. For example, `lock(l); t = s; unlock(l); t++; lock(l); s = t; unlock(l)` will not be correct if the intended functionality is to increment shared variable `s`, protected by lock `l`. It is easy to see that the result may be wrong if more than one of these operations are interleaved.

Static tools for finding concurrency bugs include at least the following kinds, ordered roughly from lightweight ones to heavyweight ones in terms of analysis complexity. 1) **Type systems**, e.g. rccjava [3] and Java atomicity types[4]. They work by extending the type system of the language with atomicity-related properties like thread-local, shared, "protected by lock l", etc. Code blocks or functions/methods also have concurrency properties defined so that the tools can

do type check to find out any inconsistency among them. The extended types can either be provided by the programmer using annotations or inferred by the tools. 2) **Program analysis tools**, e.g. Warlock[9] and RacerX[7]. These two tools do inter-procedural analysis and tracks the locking behavior of programs to find inconsistencies. Compared to type systems, program analysis tools often requires less annotation because the tools do more inference. 3) **Model checking tools**, e.g. Java Pathfinder [10] and Bandera [1]. Model checking uses a simplified model of the program and exhaustively tests it on all inputs. 4) **Extended static checking** [2] [5]. ESC uses a theorem prover to find bugs in programs, including concurrency bugs. It can check stronger properties than races.

Ways to ensure atomicity is studied extensively in the database community as the problem of concurrency control . By introducing the notion of transactions, databases delimitate the operations that it should keep atomic. Concurrency control mechanism ensure the isolation and consistency [1] of transactions either through pessimistic locking, optimistic concurrency control or timestamp based concurrency control . These mechanisms all work at runtime. They require that the database engine has full knowledge of the data structures of the database, the operations in each transactions and has full control of scheduling. In contrast, language-based detection of races and atomicity bugs need to work on general programs without knowledge of how the program works beforehand.

The rest of this survey is organized as follows. Section 2 discusses Race Condition Checker for Java. Section 3 discusses the type and effect system for atomicity, the successor to rccjava. Section 4 discusses RacerX [7], the program analysis tools for race conditions and deadlocks. And discussion of methodologies and possible future work is in section 5.

2 Type-based Race Detection

2.1 Rccjava Overview

The race condition checker for Java (rccjava) supports the locking style of programming to prevent race conditions, which are defined as unprotected accesses to shared variables. In order to verify that the program is lock free, the type system,

1. associates a protecting lock with each field
2. tracks the set of locks held at each program point

Note that in general the locks held at a particular program point can be arbitrarily different each time the program counter is at the point. However, for the Java programming language, because locking is integrated into the language, locking and unlocking must come in pairs and each pair forms a block. Therefore the locksets at each program point will be the same if they are the same at the entrance of methods. Thus the lockset tracking problem is simplified.

The type system relies on the programmer to aid the type checker by providing annotations about the additional type information, as well as by some common default values to lessen the annotation burden on programmers. The following is a simple program written in ConcurrentJava[3], a multithreaded Java subset, to illustrate the use of annotations.

```
class Account {
  int balance guarded_by this = 0
  int deposit1(int x) {
```

[1] Note that the word "atomicity" is used to refer to another property in databases, i.e. the all-or-nothing property of actions in case of program or system crashes. "Atomicity" in this surveys and [4] refers to *isolation* in database term.

```
    synchronized this in {
      this.balance = this.balance + x
    }
  }
  int deposit2(int x) requires this {
    this.balance = this.balance + x
  }
}

let Account a = new Account in {
  fork { a.deposit1(10) }
  fork { synchronized a in a.deposit2(10) }
}
```

The meaning of the annotations should be straight-forward from the example. The `guarded_by` annotation declares a field to be protected by the lock of an object (just as in Java, every object has an associated lock with it in ConcurrentJava. In this example, every access to the field `balance` must be protected by the lock of `this`. The `requires` annotation declares a method to be called with at least the declared locks to be held. In this example, the `deposit2` method should be called with lock `this` held but `deposit1` requires no lock.

2.2 Types for Race Detection

The type check in rccjava is done on a per-method basis. Every method is checked independently and the type checker only needs to check every function once.

We use the same extension to ConcurrentJava, RaceFreeJava, as in [3] to illustrate the idea of a type system for race detecting. The extension to ConcurrentJava for race detection is the following,

$$
\begin{aligned}
\mathit{field} &::= \texttt{[final]}_{\texttt{opt}}\ t\ \mathit{fd}\ \texttt{guarded\_by}\ \ l = e \\
\mathit{meth} &::= t\ \mathit{mn}(\mathit{arg}^*)\ \texttt{requires}\ \ \mathit{ls}\ \{\ e\ \} \\
\mathit{ls} &::= l* \\
l &::= e
\end{aligned}
$$

Note ls is a lock set and l is a single lock. Built on top of these annotations, the core of the type system is the typing rules. Figure 1 shows a set of most important rules.

These rules all reason about the type judgement,

$$P; E; \mathit{ls} \vdash e : t$$

where P is the program, E is the environment and ls is the current lock set, i.e. the locks held at the current program point. The rules check for correct type of every expression e of the program. The rule `EXP FORK` states that `fork` does not inherit parent's lock set. The rule `EXP SYNC` means the `synchronized` statement adds one item to the current lock set. Similarly, `EXP REF` and `EXP ASSIGN` requires that access to a shared field should be done with the required lock held. Note all `this` instances in the required lock is replaced with the current object e ($[e/\texttt{this}]l$) before looking it up in the current lock set. This enables the use of the keyword `this` in the `guarded_by` clause. Apart from this simple alias resolution, the expression in `guarded_by` should be final, i.e. a fixed object. This limitation is addressed by an extension discussed later. `EXP INVOKE` verifies that a method is invoked with the required locks held.

[EXP FORK]

$$\frac{P; E; \emptyset \vdash e : t}{P; E; ls \vdash \texttt{fork}\ e : \texttt{int}}$$

[EXP SYNC]

$$\frac{P; E \vdash_{\text{final}} e_1 : c \quad P; E; ls \cup \{e_1\} \vdash e_2 : t}{P; E; ls \vdash \texttt{synchronized}\ e_1\ \texttt{in}\ e_2 : t}$$

[EXP REF]

$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash ([\texttt{final}]_{\texttt{opt}}\ t\ fd\ \texttt{guarded\_by}\ l = e') \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \\ P; E \vdash [e/\texttt{this}]t \end{array}}{P; E; ls \vdash e.fd : [e/\texttt{this}]t}$$

[EXP ASSIGN]

$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E; \vdash (t\ fd\ \texttt{guarded\_by}\ l = e'') \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \\ P; E; ls \vdash e' : [e/\texttt{this}]t \end{array}}{P; E; ls \vdash e.fd = e' : [e/\texttt{this}]t}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} P; E; ls_1 \vdash e : c \\ P; E \vdash (t\ mn(s_j\ y_j^{j \in 1 \cdots n})\ \texttt{requires}\ ls_2\{e'\}) \in c \\ P; E; ls_j \vdash e_j : [e/\texttt{this}]s_j \\ P; E \vdash [e/\texttt{this}]ls_2 \subseteq ls_1 \\ P; E \vdash [e/\texttt{this}]t \end{array}}{P; E; ls_1 \vdash e.mn(e_{1 \cdots n}) : [e/\texttt{this}]t}$$

Figure 1: Type Rules of RaceFreeJava

2.3 External Locks

RaceFreeJava requires all lock expressions to be either final or final fields of `this`. This will not work for complex locking schemes. For example, data structures like trees often has a single lock for the whole tree while every node is a separate object. In rccjava, this is handled by introducing *ghost parameters* to class definitions. Class declaration and instantiation with ghost parameters have the following form.

$$\texttt{class}\ \ cn < \texttt{ghost}\ \ t_1\ x_1, \cdots, \texttt{ghost}\ \ t_n\ x_n >\ \ body$$

$$\texttt{class}\ \ cn < l_1, \cdots, l_n > [l_1/x_1, \cdots, l_n/x_n]body$$

The type check for classes with ghost parameters are done by substitution. Thus this mechanism works a little bit like templates. Note also that the actual ghost parameters must be final. It can also refer to `this` which is the object containing the field.

For an example of the usage of using ghost parameters to specify external locks, see [3].

2.4 Thread-local Classes

One problem with RaceFreeJava is every field needs to be annotated because the type checker assumes all fields can be accessed by multiple threads. In real programs, apart from local(on-stack) variables, a lot of object referenced by fields are also accessed by one only thread and thus requires no locking. By extending the type system to support these thread-local classes, not only number of annotations will be reduced, but also thread-local access can be enforced by the type checker. The latter is achieved in rccjava using a form of escape analysis. Basically thread-local objects should not be accessible from any thread-shared objects. Therefore thread-shared classes must only have thread-shared super-class and must only contain shareable fields. However, thread-local classes is allowed to have thread-shared super-classes. Anyway the `Object` class is thread-shared.

In order to preserved the single-thread guarantee of thread-local classes, limitations need to be made to up and down casts of thread-local classes. In rccjava, thread-local classes are not allowed to override methods of any shared super-class. And downcasts from shared super classes to thread-local sub-classes are forbidden. Note that the downcast limit may be too much for a lot of applications. This basically forbids downcasts from `Object` to any thread-local classes. This means thread-local classes cannot be put into collection classes like `ArrayList` and `HashMap`.

2.5 Implementation Issues

Implementing these typing rules will result in a working type checker. But it will still require too many annotations and produce too many spurious warnings to be useful. Therefore, rccjava employees two techniques to alleviate these problems. One is to support a number of escape pragmas to suppress certain kinds of warnings at certain places. This makes the type system unsound. But practically, it does not prevent the tools from being a useful bug-finding tool. The other technique is to support a number of default annotation rules, e.g., unguarded non-final instance fields in thread-shared classes are guarded by `this`. These rules do not change the type system at all, but makes a big difference for the usability of the tool.

Rccjava has been reported [3] to be applied to Java library classes and several small to medium scale applications. It is reported that fewer than 20 annotations are required per 1000 lines of code. And the annotations tend to be clustered on some classes manipulated from different threads. One hour per 1000 thousand lines is the reported average annotation time, although it should depends highly on the locking style of the program. Several races have been found in Java library classes using rccjava.

3 Type-based Atomicity Checking

We already mentioned that race conditions are only part of the reasons for concurrency bugs and we have seen an example of such a bug in the introduction part. We may be able to solve this problem if we focus on the *behavior* of functions/methods when executed concurrently, instead of how specific variables are accessed. The simplest case is memory read/write of a single word is atomic on most hardware, and a series of accesses to variables protected by certain locks while holding the locks are atomic. Thus operations and functions/methods can be classified as atomic or non-atomic. Programmers normally have a pretty clear idea of what functions/methods are intended to be atomic and what are not. So they can annotate functions/methods as atomic or not.

This approach is taken in [4], which adds atomicity types to Java. The concurrent behavior of operations and methods are called *effects* in this work. Therefore the type system is extended to

include both data types and effects of operations. The type checker verifies the declared atomicity of methods are actually satisfied by the operations in the bodies of the methods.

3.1 The Theory of Movers

In order to reason about atomicity types, it is essential to introduce more levels of atomicity other than *atomic* and *non-atomic*. The theory of movers by Lipton [6] is useful here. An action a is a *right mover* is for any execution where the action a performed by one thread is immediately followed by action b of another thread, these two actions can be swapped without affecting the resulting state. Similarly we define a *left mover* to be an action a which can be swapped with any action b immediate before it by another thread without affecting the resulting state. Concretely, a locking operation is a right mover, because any operation by another thread immediately after the locking operation cannot change any state protected by the lock and thus can be swapped with the locking operation. Similarly, a unlocking operation is a left mover. An access to a shared variable protected by lock l is both a right mover and a left mover because the lock is guaranteed to be held by the type system.

Classifying certain operations as movers enables us to reason about atomicity properties of methods. In particular, the following theorem holds,

Theorem. A method is atomic if it contains a sequence of right movers followed by a single atomic action followed by a series of left movers.

The correctness of the theorem can be seen as follows. For any interleaved execution of the method with any other actions of other threads, we can move the right movies of the method to the right until they reach another right mover or the atomic action. And the left movers to the left. In the end, this method will be executed without interleaving with actions from other threads. These can be done without affecting the resulting state. Therefore, any interleaved execution of this method always results in the same program state.

3.2 Types for Atomicity

The following atomicity types are defined in [4], ordered from the strongest atomicity to the weakest.

- const: expressions whose evaluation does not depend on any state and does not change state.
- mover: as defined above except that all movers are both left and right in Java.
- atomic: yields same resulting state independent of the interleaving with other threads
- cmpd: non of the above
- error: typing error

The atomicity of *iterative closure* $\alpha*$ is the same with that of α except for `atomic`, where a series of atomic actions have atomicity `cmpd`. And *sequential composition* of α_1 and α_2 has the weaker atomicity of the two, exception that two `atomic` expressions form a `cmpd` expression.

Atomicity of some operations depends on the locks held, e.g. accessing a shared field field is a `mover` if the required lock is held or `error` otherwise. This is written as a conditional atomicity,

$$l?\texttt{mover} : \texttt{error}$$

3.3 Typing Rules

The atomicity type system is an extension of that of rccjava. Here we use the typing rules of AtomicJava[4], an extension to ConcurrentJava to illustrate the type system. Some of the interesting rules are listed below. They are all about the typing judgement, $P; E \vdash e : t\&a$, where t is the normal data type of e and a is the atomicity of e.

[EXP WHILE]

$$\frac{P; E \vdash e_1 : \texttt{int}\&a_1 \quad P; E \vdash e_2 : t\&a_2}{P; E \vdash \texttt{while}\ e_1\ e_2 : \texttt{int}\&(a_1; (a_2; a_1)*)}$$

[EXP REF GUARD]

$$\frac{\begin{array}{c} P; E \vdash e : c\&a \\ P; E \vdash (t\ fd\ \texttt{guarded\_by}\ l = e') \in c \\ b \equiv (l[\texttt{this} := e]?\texttt{mover}) \quad P; E \vdash b \end{array}}{P; E \vdash e.fd : t\&(a; b)}$$

[EXP CALL]

$$\frac{\begin{array}{c} P; E \vdash e_1 : t_1\&a_1 \quad t_0 = c \quad P; E \vdash b[\texttt{this} := e_0] \\ P; E \vdash (b\ s\ mn(t_1\ y_1, \cdots, t_n\ y_n)\{e\}) \in c \end{array}}{P; E \vdash e_0.mn(e_1, \cdots, e_n) : s\&(a_0; a_1; \cdots; a_n; b[\texttt{this} := e_0])}$$

[EXP SYNC]

$$\frac{P; E \vdash l : c\&\texttt{const} \quad P : E \vdash e : t\&a}{P; E \vdash \texttt{synchronized}\ l\ e : t\&S(l, a)}$$

`[EXP WHILE]` uses sequential composition and iterative closure to calculate the atomicity of a `while` loop. Note that the loop condition e_1 is evaluate first.

`[EXP REF GUARD]` checks accesses to lock guarded fields. The field access itself is a `mover` if the lock is held and `error` otherwise. The atomicity of the whole expression ($e.fd$) is the sequential composition of the atomicity a of expression e and that of the access itself b.

`[EXP CALL]` checks method calls. The atomicity of a method call is the sequential composition of the evaluation of the object, the parameters and the method invocation itself. The atomicity of the invocation itself is got by replace `this` with the real object in the formal atomicity annotation, which can be a conditional atomicity.

`[EXP SYNC]` deals with locking operations. It is a little more complicated. The $S(l, a)$ function evaluates the atomicity of the statement when l is locked or not. For example, $S(l, \texttt{const}) = l?\texttt{const} : \texttt{atomic}$ and $S(l, \texttt{mover}) = l?\texttt{mover} : \texttt{atomic}$. This means when lock l is already held, the `const` and `mover` of the expression is kept for the whole statement. Otherwise, the statement is atomic. For conditional atomicities, $S(l, (l?b_1 : b_2)) = S(l, b_1)$ and $(S(l, (l'?b_1 : b_2)) = l'?S(l, b_1) : S(l, b_2)$ if $l \neq l'$. This should be easy to understand. The type checker basically resolves the conditional atomicity if the lock in `synchronized` is the one in the conditional atomicity. Otherwise, it recurses.

There are other important rules not discussed here. See [4] for a complete discussion.

3.4 Implementation and Applications

The authors of [4] implemented the atomicity checker based on rccjava for the full Java language. Similar relaxing techniques as discussed in section 2.5 are used to suppress spurious reports and reduce annotation numbers.

The type checker is applied to several JDK 1.4 library classes. A concurrency bug in `StringBuffer` is found. The conditional atomicity was proved to be useful in type checking `java.util.Vector` class.

4 RacerX: Static Detection of Race Conditions and Deadlocks

Type system based techniques are useful but they normally requires slightly heavy annotations and are sometimes too restrictive in terms of locking scheme. The fact that type checking is normally done intra-procedural also limits the "cleverness" of the checker and mandates more annotations. More sophisticated program analysis can potentially do clever inference and thus reduce annotation amount and check for invariants not easily expressed as type systems. For large existing systems, the amount of annotations needed is an important factor in whether the application will be successful. RacerX [7] is a static program analysis tool to detect both race conditions and deadlocks. It is intended to be used on complex systems like the Linux and FreeBSD kernel. Here we only discuss its race detection part.

4.1 Analysis Overview

RacerX uses flow-sensitive, depth-first and inter-procedural analysis to detect races in C programs. It is flow-sensitive because every function is analyzed separately at each call-site, because different locks can be held at different call sites. Note that rccjava mandates each function to be called all holding the required locks. It is inter-procedural meaning that analysis follows function calls and thus can be exponential. At a high level, using RacerX to check a system involves the following phases, (1) retargeting it to system-specific locking functions, in which the programmer provide a list of locking/unlocking function and specify their specific functions (2) extracting a control flow graph from the system, (3) detecting races and deadlocks from the program, (4) post-processing and ranking the results, (5) inspection.

The control flow graph (CFG) is a simplified model of the whole system. It contains information like all function calls, access to shared variables, any concurrency operations like lock/unlocking and interrupt disabling/enabling, etc. The CFG facilitates later processing by providing a linked, in-memory representation of the relevant information in the whole system.

The analysis then starts from each root function in the CFG and follows all the function calls in the function recursively. Lockset tracking is the basic form of the analysis. At function level, the analysis produces for each function f a list of mappings, each having the form $l \rightarrow (l_1, \cdots, l_n)$, meaning calling f with lockset l will possibly produce any one of $l_1, \cdots, l_n$. At the statement level, each path is taken to exhaustively to generate the list of resulting locksets for each function. Cache can be done at both the function level and statement level to reduce the amount of analysis significantly. This is analysis is certainly exponential. However, most functions should acquire and lock locks consistently if they are meant to be easily understood. Therefore the number of locksets in the resulting list for each input lock should be small.

With the lockset information available for each statement after the analysis is done, in an overly simplistic way, race conditions can be detected by finding those statements that access global variables without holding locks. A backtrace of how the race happens can also be obtained by tracing back the analysis path and print out all the function calls and branches taken.

4.2 Post Processing and Result Ranking

The analysis described above is overly simplistic and for real program a lot can go wrong. Unlike rccjava, where users annotate the program to provide information like thread-local classes, RacerX

uses mostly post processing and result ranking to alleviate these problems. The following are some practical problems of the simplistic approach,

- Locking information may not be accurate. For example, semaphores are used in the Linux kernel both as locks and as a rendezvous mechanism to implement producer-consumer relationships, where the producer `up`s the semaphore and the consumer `down`s the semaphore. If these are treated as locks, the consumer will be holding the lock for ever after this.
- False paths exist, for example in parameter-controlled locking where both the locking and unlocking operation are placed inside the `if` statement controlled by the condition. Two extra false paths will be generated for this setting, one locking without unlocking and one unlocking without locking. This leads to false positives.
- Certain part of the code may not be multithreaded. For example, the boot process of the Linux kernel is mostly single-threaded.
- Variable x may not need to be protected. Some are only read by multiple threads, and some are modified in an carefully arranged way so that they always represents consistent state.

RacerX handles these problems by giving scores to potential errors found. The errors are then sorted by their scores before presented to the user, in the hope that most relevant errors are on top of the list, just as Internet search engines do with their results. Unfortunately, the techniques to determine the scores in RacerX are very ad-hoc and some are complex. Here we list how RacerX handles the last problem to show the flavor of the method. See [7] for a complete treatment.

Handling variables not requiring protection
Count number of times that X was the first, last or only object in a critical section
+4 if only object > 1 times, +2 if 1 time.
+1 if first object > 0 times.
+1 if last object > 0 times.
Compute z-test statistic based on count of how often protected with any lock versus not protected.
+2 if $z > 2$.
-2 if non-global and $z < -2$.
Count the number n of unprotected variables in the non-critical section.
+2 if $n > 4$.
+1 if $n > 1$.
Non-atomic updates: writes to $>$ 32-bits or bitfields.
+1.
Access was a write.
+1.

4.3 Applications

Results are reported of using RacerX on the Linux kernel, the FreeBSD kernel and the kernel of a commercial OS, System X. The annotation density is about 100 per 1M. For Linux the reported results are, 3 bug, 2 unconfirmed, 2 benign and 6 false positives.

5 Discussion and Conclusion

Tools for automatic checking of concurrency bugs are still quite far from satisfactory. This is probably due both to the inherent complexity of concurrent systems and the in-expressiveness of popular systems languages like C. At a high level, the tools we surveyed take two radically different approaches to these problems. Rccjava and atomicity types extends the language and let the programmers provide more information through annotations. RacerX, on the other hand, tries to infer concurrency control scheme of the program by program analysis with little help from user annotations. Of course the RacerX way makes the tool easier to use and thus effortlessly deployable. However, from the techniques and results reported, the error ranking techniques are ad-hoc, sometimes subjective and possibly domain specific. All three applications of RacerX are operating system kernels which share common concurrency patterns. RacerX may or may not work well with other applications and may need significant tweaking. On the other hand, language-based solutions are harder to deploy because of the porting/annotation overhead. Once deployed, it can make the development more productive.

Alias resolution is an important problem in all these tools. The existence of reference and pointer types mandates some form of alias resolution to determine the alias relationships of references and pointers. All three tools take simple approaches to this problem. Rccjava and atomicity types allow aliasing of `this` and the actual object in locking annotations. External locks extends this aliasing resolution somewhat by allowing an object contained in a parent object to be protected by locks on the parent object or any final fields reachable from it. RacerX obviate the alias problem by representing local and parameter pointers by their type names rather than the thing they point to. This is a conservative approximation that can lead to spurious race reports. Although this will not miss any real races, it may make the real ones harder to identify by the user because of the extra spurious ones.

The theory of *movers* looks surprisingly similar to the $two-phaselocking$ scheme in database concurrency control. $Two-phaselocking$ states that a transaction is guaranteed to be *isolated* from other transactions if it did not release any lock before it acquires all locks it needs. Because lock acquisitions are right movers and lock releases are left movers. This corresponds directly to the structure of an atomic action by the theory of movers, i.e. "right-movers - atomic action - left movers structures". Putting the work in this context makes it clearer and we may well be able to borrow from the database research community. Additionally, in this sense the name "atomicity" types is actually a misnomer and should be called "isolation" types, in the ACID (Atomicity, Consistency, Isolation and Durability) terms. The `atomic` methods in [4] are not *atomic*, because they do not survive program or machine crashes, but *isolated* from other concurrent actions.

References

[1] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Roby. Bandera: a source-level interface for model checking java programs. In *International Conference on Software Engineering*, pages 762–765, 2000.

[2] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report #159, Palo Alto, USA, 1998.

[3] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, British Columnbia, Canada, 2000.

[4] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003.

[5] K. R. M. Leino, G. Nelson, and J. B. Saxe. Esc/java user's manual. Technical Report 2000-002, Compaq SRC, October 2000.

[6] R. Lipton. Reduction: A method of proving properties of parallel programs. 18:12:717–721, 1975.

[7] D. on Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlock. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, New York, USA, 2003.

[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[9] N. Sterling. Warlock - a static data race analysis tool. In *USENIX Winter*, pages 97–106, 1993.

[10] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker, 2000.