A Type and Effect System for Atomicity – A Review

Hoang Thai Duong

ABSTRACT

In this report, we summarize and analyze the PLDI 2003 paper "A Type and Effect System for Atomicity" by Flanagan and Qadeer [1]. We show that while this paper is the first to use static analysis to successfully verify a fundamental correctness property in multithreaded programs that is atomicity, its approach has limitations that should be, and indeed have been, addressed in future work.

1. INTRODUCTION

There have been strong beliefs in recent years that parallel computing is the future for computing. As a consequence, multicore machines, and multithreaded software that can take advantage of such machines, have becoming increasingly more popular. Compared to serial programs, multithreaded programs are harder to test and debug. The main reason is that the interleaving of threads results in many complex execution scenarios and makes synchronization bugs non-deterministic, thus hard to reproduce and fix.

One common source of bugs in parallel programs is race condition, where more than one threads try to access a shared variable simultaneously, and at least one of the accesses is a write. There have been works trying to detect race conditions using a variety of techniques. However, Flanagan and Qadeer [1] argue that the absence of race conditions is neither necessary nor sufficient for a program to be correct. They advocate checking for a stronger non-interference property called atomicity, where atomic methods can be assumed to execute in "one step", without interferences of methods from other threads. To that end, the authors propose a type system to specify and verify the atomicity of methods in multithreaded Java programs. They show that their approach successfully finds atomicity violations in the JDK's codebase that can lead to crashes.

This report elaborates more on the importance of atomicity, and discusses race conditions and atomicity in a larger context. We also give reasons to justify the use of a type system instead of other static analysis techniques. In the same spirit, we states the advantages and disadvantages of the static analysis approach taken by [1] compared with other methods such as dynamic analysis and model checking used in later papers. Finally, we explain in detail and give examples to illustrate the limitations of the proposed type system when verifying atomicity in practice.

2. SUMMARY

2.1 The Need for Atomicity

As mentioned, race conditions are a common source of synchronization bugs. As an example, consider the Java program fragment below.

```
class Account {
  int balance = 0;
  int deposit(int x) {
    balance = balance + x;
  }
}
```

The code above can behave unexpectedly when two or more calls to `deposit` are interleaved. The final value of balance may only reflect one of the many calls. That is because the program contains a race condition on the variable balance. We can protect that variable by putting the update statement into a `synchronized` context, thus getting rid of the race condition.

```
int deposit(int x) {
  synchronized (this) {
    balance = balance + x;
  }
}
```

However, race-freedom does not imply a program is error-free. Consider the code below where we extends the Account class above to include two more methods.

```
int readBalance() {
  synchronized (this) { return balance; }
}

void withdraw(int amt) {
  int b = readBalance();
  synchronized (this) { balance = b - amt; }
}
```

There are no races in either method, but an execution like the one below will cause an incorrect behavior of the method `withdraw`.

```
Thread 1: int b = readBalance();
Thread 2: deposit(10);
Thread 1: synchronized (this) {
            balance = b – amt;
          }
```

The reason the above execution can happen is because in the `withdraw` method, its statements can interleave with those from other threads and change the program's behavior. We can rewrite the method to make it atomic and at the same time rewrite `readBalance` to get rid of the redundant synchronized context as follows.

```
int readBalance() { return balance; }

void withdraw(int amt) {
  synchronized (this) {
    balance = balance – amt;
  }
}
```

Now there is a race condition on the variable balance, but it is benign as it does not affect the correctness of the program. We can conclude that a race condition does not necessarily mean a program has bugs, and the absence of race conditions does not imply a program is correct either.

The problems with race-conditions call for the need of another, stronger non-interference property that is atomicity. A method is atomic if for any execution of the program in which the method's statements are arbitrarily interleaved with statements from other threads, there is an equivalent execution of the program in which the method's statements are executed serially without interferences. In other words, if a method is atomic, the scheduler can freely interleave the execution of that method with other threads, the result of the program is the same regardless. Atomicity as a concept is useful since once a method is known to be atomic, it can be further analyzed and reasoned about using sequential reasoning techniques. Atomicity is a fundamental correctness property since atomic operation is a common way for ensuring thread safety. In practice, a lot of interface methods are intended to be atomic (or thread-safe). Therefore, it is useful to provide programmers with a methodology to specify and verify the desired atomicity of methods. To achieve that goal, Flanagan and Qadeer [1] propose a type system for specifying and checking atomicity properties of methods. We will give an overview and assess this type system for atomicity in subsequent Subsections.

2.2 Lipton's Theory of Reduction

The type system presented in [1] is based on Lipton's theory of right and left movers [2]. An action is a right mover if it can always be swapped with its immediately following action from another thread without changing the resulting state. A left mover is defined similarly. In a multithreaded programming context, a lock acquiring action is a right mover, while a lock releasing action is a left mover. A variable access while holding the corresponding lock is both a right and left mover. The following example illustrates the use of this theory to verify atomicity.

```
void deposit(int x) {
  synchronized (this) { b = b + x; }
}
```

Below are two different executions of the code above. In the first execution, the operations of method deposit are interleaved with other operations from another thread. Then, the operations that are movers are swapped with their neighbors resulting in the second, non-interleaving, execution of `deposit`. By swapping operations according to the rules of Lipton's theory, method `deposit` can be proven atomic.

$$\rightarrow aquire \rightarrow x \rightarrow r = b \rightarrow y \rightarrow b = r + x \rightarrow z \rightarrow release \rightarrow$$
$$\rightarrow x \rightarrow aquire \rightarrow r = b \rightarrow b = r + x \rightarrow release \rightarrow y \rightarrow z \rightarrow$$

In general, the theory states that any sequence of actions consisting of a sequence of right movers, followed by at most one atomic action, followed by a sequence of left movers, can be considered atomic.

2.3 Atomicity Types

The proposed type system assigns to each expression an atomicity type. At the fundamental level, there are five basic atomicities described below.

- $const$: An expression e has type $const$ if valuating e does not depend or change any mutable state.
- $mover$: An expression e is a $mover$ if it can be swapped with its immediate neighboring operations of other threads.
- $atomic$: An expression e is $atomic$ if its execution can safely be considered a non-interleaved one.
- $cmpd$: An expression e has atomicity type $cmpd$ if it does not belong to one of the above categories.
- $error$: An expression e has atomicity type $error$ if it violates the locking rules used by the program.

In an intuitive way, iterative closure (where an expression is executed a number of times) and sequential composition (when two expressions are executed one after another) of the five basic atomicity types are defined. We give some examples below.

$$\begin{aligned} mover^* &= mover \\ atomic^* &= cmpd \\ cmpd^* &= cmpd \\ mover; atomic &= atomic \\ atomic; atomic &= cmpd \\ cmpd; cmpd &= cmpd \end{aligned}$$

Two atomicities can also be joined together by the joint operator ($\sqcup$). This joint type ($\alpha_1 \sqcup \alpha_2$ where α_1 and α_2 are basic atomicities) reflects the atomicity type of an `if` statement.

Since some expressions have different atomicity types depending on which locks are held at the time of executing the expression, the set of atomicities is also extended with conditional atomicities. For example, accessing a protected field with the corresponding lock held is a $mover$, but without the lock held it is an $error$. If an expression e has a conditional atomicity type of the form $l?\,a\colon b$, it means the atomicity type of e is a when lock l is held, otherwise it is b.

Iterative closure ($*$), sequential composition (;), and joint operator ($\sqcup$) are intuitively extended to conditional atomicities as well. Some of the rules are provided below.

$$\begin{aligned} (l?\,a\colon b)^* &= l?\,a^*\colon b^* \\ (l?\,a_1\colon a_2); b &= l?\,(a_1; b)\colon (a_2; b) \\ (l?\,a_1\colon a_2) \sqcup b &= l?\,(a_1 \sqcup b)\colon (a_2 \sqcup b) \end{aligned}$$

For subtyping to work and thus the type system can be more precise, an order is defined over the set of atomicity types. The ordering rules can be summarized as follows.

$$const \sqsubset mover \sqsubset atomic \sqsubset cmpd \sqsubset error$$

$$\begin{aligned} a &\sqsubseteq a^* \\ a &\sqsubseteq a; b \end{aligned}$$

$l?\,a_1\colon a_2 \sqsubseteq b$ if $a_1 \sqsubseteq b$ and l is held, or $a_2 \sqsubseteq b$ and l is not held.

Finally, a set of rules are defined so that atomicity types can be easily manipulated and simplified. Some of them are reproduced below.

$$\begin{aligned} (a^*)^* &\equiv a^* \\ (a; b); c &\equiv a; (b; c) \\ a; const &\equiv a \\ a; (b \sqcup c) &\equiv a; b \sqcup a; c \\ (a \sqcup b)^* &\equiv a^* \sqcup b^* \end{aligned}$$

2.4 Type Annotations

A subset of the Java language is extended with type annotations so that the programmer can specify the intended atomicity type of each method and other synchronization information needed to type-check the program. In particular, each field can be protected by a lock with the `guarded_by` annotation. A method declaration can specify a set of locks that are supposed to be held when the execution of the method begins using the `requires` annotation. An example is provided below.

```
class Account {
  int balance write_guarded_by this = 0;
  atomic int withdraw(int x) requires this { … }
}
```

In practice, the annotations are implemented as special forms of Java comments.

2.5 Type Checking Rules

The core of the type system is a set of rules to verify the atomicity of expression given the atomicity types of its components. The system takes as input a fully annotated program and verifies the intended atomicity of each method. A type judgment has the form $P; E \vdash e: t \,\&\, a$ and it means for program P, under type environment E, expression e has type t and atomicity a. Note that even though the authors of [1] use the term "atomicity types", the atomicities are actually specified as an effects; the type of each expression is just its ordinary type in Java. We reproduce four rather simple rules below and briefly explain their meaning. The rest of the rules and more detailed explanations can be found in [1] or [3].

- [WHILE]

$$\frac{\begin{array}{c} P; E \vdash e_1: int \parr a_1 \\ P; E \vdash e_2: t \,\&\, a_2 \end{array}}{P; E \vdash while\ e_1\ e_2: int \,\&\, (a_1; (a_2; a_1)^*)}$$

- [REF GUARD]

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

- [CALL]

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

- [SUB]

$$\frac{\begin{array}{c} P; E \vdash e: s \parr a \\ P \vdash s <: t \\ P; E \vdash a \sqsubseteq b \end{array}}{P; E \vdash e: t \,\&\, b}$$

The [WHILE] rule says that if e_1 has atomicity a_1, e_2 has atomicity a_2, the expression $while\ e_1\ e_2$ has atomicity $(a_1; (a_2; a_1)^*)$. The [REF GUARD] rule is used when e is an expression accessing a variable fd guarded by lock l. Expression e has atomicity type $mover$ provided l is held, otherwise it is an $error$. The [CALL] rule says that the atomicity type of a call expression is the sequential composition of the atomicities of its arguments followed by the annotated atomicity of the callee. The [SUB] rule allows for subeffecting, making the system more precise. One important point to note is that in practice, the type system relies on a race condition checker called `rccjava` [4] to obtain the set of locks held at each program point. `rccjava` is the checker for another type system developed by the same authors to detect race conditions.

2.6 Evaluation

The authors of [1] manually annotate several classes in the JDK and show that their type system successfully discovers atomicity violations which are not race conditions in the JDK's `java.util.StringBuffer` and `java.lang.String` classes. They also provide the exact number of annotations needed and the number of lines of code for each class to convince that their type system requires a reasonable amount of annotations. On average, 23.3 annotations are needed per KLOC.

The authors are also aware the limitations of their approach. The type system does not capture all synchronization mechanisms in practice, so it relies on ad-hoc relaxing techniques such as the `no_warn` and `holds` annotations and the `–constructor_holds_lock` command line tag. The authors also report problems with changing protection mechanism (arrays) and rep-exposure (`java.io.PrintWriter`). They also note a case in `java.net.URL` where it is unclear whether the warnings the type system gives is benign.

3. ANALYSIS AND DISCUSSION

3.1 Atomicity and Race Condition

The authors of [1] state a number of advantages of atomicity to convince it is a more meaningful concept than race condition. It is more accurate, however, to view atomicity and race condition as just two correctness properties at two different levels of abstraction. Race condition is at the lower level, where the concern is about guarded variable being accessed by more than one threads. Atomicity is at a higher level, where the effect of interleaving executions is concerned. There are certainly other correctness properties at higher level of abstraction. Consider the `class Account` example above; one can, for example, require that the first call must be a `deposit` call instead of a `withdraw` call, since in the beginning there is nothing to `withdraw`. In general, multithreaded methods may not only need to be atomic, they may also need to be executed in certain order for the program to work correctly.

It is mentioned in [1] that race freedom is neither necessary nor sufficient to ensure the absence of synchronization bugs (actually [16] points out that the data races allowed in [1] is not benign in the Java or C# memory models, only for sequentially consistent systems). Another reason why it makes more sense to say that atomicity and race condition are just two concepts at different levels of abstraction is that the same thing can be said about atomicity. First of all, the lack of atomicity does not mean there are bugs. There are synchronization mechanisms that may not rely on atomic operations, such as `barriers,` or any synchronization mechanism that does not used share variables. Then obviously there are programs that fail even though all of its methods are atomic. For an example, consider the `class Account` again, a call to `withdraw` when the balance is `0` may cause unexpected behaviors.

It is also worth noting that atomicity does not "encompass" or "replace" race condition. Methods that check for race conditions are still useful, since an atomicity checker must rely on a race condition checker to reason about the set of locks held at each program point, thereby showing certain expressions are mover. The type system described in [1] uses `rccjava`, a type system for race condition, in practice. Also, arguably, many atomicity violations are also race conditions [5], thus most of the time one can uncover the majority of atomicity violations using a more lightweight type system for race condition. The authors of [1] should have done experiments with some code base that has both types of violations and provided some statistics to convince the reader that the number of atomicity violations (that are not race conditions) is indeed significant.

Having said the above, it is still true that atomicity is a very important concept and all the advantages of it mentioned in [1] are significant. There are also other reasons why atomicity is a fundamental correctness property that are not mentioned in [1]. For example, data race can be automatically taken care of in a transactional programming model (where memory write and read operations can be made atomic by hardware or software), but atomicity is still a problem [6]. Transactional memory, however, does make ensuring atomicity easier for the programmer, so a type system to verify atomicity for high level code becomes less important. For low level synchronization libraries (e.g. software transactional memory libraries), it's still very valuable.

3.2 Static Analysis

There are a number of ways to verify a method's atomicity, static analysis one of them. The other methods include dynamic analysis [7], model checking [8], and theorem proving [9]. Among the approaches, static analysis is usually conservative (less precise) but sound. Dynamic analysis is often unsound and has small coverage, but it is often automatic and its results are more precise (less approximation). Model checking has the problem of exponential number of states, thus its coverage is often small. Atomicity verification as a problem does not clearly favor any particular approach. In fact, a combination of approaches or a hybrid approach usually works better than any of them alone [20].

The static analysis approach used by [1], however, is the first work to propose the atomicity verification problem and solve it using a program analysis method. It is also not surprising that a type system is used since the authors already developed a type system for race condition checking. The new type system is understandably a natural extension of previous work.

3.3 Type System

As said above, the type system in [1] is naturally extended from another type system developed previously [4]. There may be other reasons as to why a type system is used instead of other static analysis techniques such as data flow (equation-based) analysis, constraint-based analysis, and abstract interpretation. First of all, atomicity is inherently a property of a region of code. It is therefore naturally to express this property as types and/or effects (the type system in [1] is written as an effect system, the real type of an expression is its original Java type). The problem may very well be formulated as solving a set of equations/constraints, but it is just unintuitive and unproductive to think of it that way. As for abstract interpretation, it is not needed here since typing rules can be checked simply by following the syntax; there is no need to work in some abstract domain.

Another unique advantage of using a type system is that the annotations required by the type checker can be meant for human to read as well. The programmer can use the annotations to document and communicate the intended atomicity type for each method. Annotations help programmers maintaining a code base understand and reason about the code better.

Expressing atomicity as types also makes the analysis more modular and scalable. Type checking can be done for each class in isolation and growing the code base has little impact on the system's complexity and performance.

3.4 The Type System for Atomicity

One nice thing about the type system proposed in [1] is that it is expectedly sound, which means if a well typed method is intended to be atomic, it is guaranteed to be atomic. On the other hand, the system is necessarily incomplete. It may give false alarms (i.e. it may deduce that some methods are not atomic while in fact they are). The reason is that the type system does not capture all synchronization mechanism and complex locking idioms used in practice. We will give a brief overview of some of them below and explain why the type system in [1] fails for such cases.

The type system supports re-entrant locks but fails to detect dead-locks. A re-entrant lock is a lock that can be acquired by a thread multiple times in a row without blocking on itself. This situation may arise from, for example, a method traversing a graph and accessing a protected node multiple times. Java has support for such lock, for example, in the JDK class `java.util.concurrent.locks.ReentrantLock`. This situation is not supported by Lipton's theory of reduction but is supported by the atomicity type system in [1]. In Lipton's theory of reduction, there is a key condition that requires the last $k - 1$ statements in a series of k reducible statements be executable. A statement may not be executable if it tries to acquire a lock that is not yet released. So the following block of statements, which is atomic with the support of re-entrant lock, is not reducible according to the theory.

```
acquire(l); acquire(l); release(l); release(l);
```

However, the type system does not enforce the aforementioned condition in the theory, so it can reduce the code above. That means the type system assume all locks are re-entrant by default, which is understandable since the `synchronized` keywords in Java implicitly specifies a re-entrant lock. The problem is then, normal locks are not supported. If in the code above, l is a normal lock, a dead-lock will occur – the thread blocks itself and never progresses. The type system, however, are unaware of the dead-lock and will (wrongfully) consider the four statements as an atomic operation. This issue is also raised in [10] but without explanations.

The type system does not support protected locks. A protected lock is a lock that is protected by another lock. In other words, to acquire a lock `l` would require holding lock `l'` first. There is no way to specify (and thus, check for) this discipline using the atomicity type system. Adding the support for protected locks, if possible, would make the type system significantly more complex, since there can be arbitrary number of protected levels. Another related mechanism where a field is protected by two or more locks is also not supported.

Another drawback of the type system is that it does not support manual locking using mutexes or semaphores. Some intuitively atomic methods are not reducible. The code below is an example.

```
1: while (mutex == 0) { }; // spin lock
synchronized (this) {
  if (mutex > 0)  mutex--;
  else goto 1;
} // acquire lock
doSomething(); // atomic
synchronized (this) { mutex++; } // release lock
```

The `doSomething` call above is atomic since it is put inside a critical section guarded by manual locking using the variable `mutex`. However the type system cannot assign the correct atomicity type to the call because it is not inside any synchronized context. This defect of the type system is also stated (without discussion) in [11], among other defects such that the lack of support for the unique protection mechanism, classes cannot be parameterize by readonly and self, and the restriction that all instances of a class must use the same protection mechanism for a field.

Some limitations of the type system are briefly stated in [1] but without sufficient discussion. Here we give a more detailed explanation of the issues. The first issue has to do with rep-exposure [12]. A rep-exposure happens when an encapsulated field is exposed to the outside of the object containing it. In the paper, the println method in java.io.PrintWriter is used as an example.

```
public void println(int x) {
  synchronized (lock) {
    print(x); println();
  }
}
```

The two inner calls to print and println write to a Writer object which is passed from outside to the PrintWriter constructor. Therefore, another thread could write to that Writer object at the same time as the println method above, causing a data race, even though all statements of println are inside a synchronized block. However, the println method is still atomic if the Writer object is never “leaked” to another thread. The type system cannot cater for such cases and thus can only type check successfully when println is declared cmpd (compound). The authors of [1] suggest using an ownership type system [13] or escape analysis [14] to reason about this case. An ownership type system can help locate where in the heap an object belongs to, while an escape analysis can give information as to which methods have access to a variable. Using those methods, the system can determine whether the Writer object can be accessed by other threads, thus give a more precise atomicity type for the println method. A similar issue caused by the fixed ownership relationship used by the type system in [1] is that it cannot deal with changing protection mechanisms such as the one used for Java’s arrays. An ownership type system can help in this case as well.

Another limitation mentioned in [1] is when other threads have access to this object before a constructor returns. This situation is rare but definitely possible, for example, when a constructor forks another thread and pass the this pointer to that thread. This may cause the constructor to become not atomic even though most of the case it does (since leaking the this pointer in constructor to other threads is rare). The paper deals with this case by using the –constructor_holds_lock flag to force the system to assume that the lock this is held in constructors, but also suggests that an escape analysis may be used instead. Using the flag above is more practical in most cases because it is almost always correct and it gives a more precise results than using a sound escape analysis, which can be too approximate.

The –constructor_holds_lock above is just one of a few relaxation techniques the paper uses in practice. From the examples above and the fact that the system in [1] needs unchecked annotations and assumptions about the synchronization disciplines of programs, we can clearly see that the type system is not expressive enough. It is because not all synchronization disciplines can be captured by the theory of reduction or expressed using types. Some of them can be captured by combining the atomicity type system with other static analyses, others by using ad-hoc annotations and assumptions. Extending the type system itself, if possible, seems risky since one may violate the soundness property of the type system in exchange for more expressiveness.

3.5 Annotations

One aspect of the type system in [1] that receives little treatment is the annotations. In some sense, this is the major weakness of the method. In order to type check a program, the programmer must fully annotate the code with expected locks, expected atomicity types, and special annotations and flags. The density of annotations is not high as shown in Table 1 in [1]. The problem is this atomicity type system relies on another type system for race condition to reasons about locks. The second type system itself needs annotations, thus the true density of annotations is higher than the one shown in [1]. Unfortunately, this point is not raised in the paper and no number is provided. It is, however, noted in a future work that solves the same problem using dynamic analysis [15].

The need for heavy annotations also means the type system is less helpful for legacy code bases. To understand the intended atomicity types of each method for existing code takes time and the programmer writing annotations can easily make mistakes as well. An example supporting this case can be found in [1], where the authors mention they cannot determine the intended atomicity type of the method checkSpecifyHandler in the JDK class java.net.URL.specifyHandlerPerm.

For newly written code, one can also argue that the type system provides little benefit in actually writing multithreaded code, since the programmer must still insert locking calls manually without any help from the system. Also, the authors of [6] note that if a programmer can easily specify atomic regions, he or she can also properly synchronize them without much effort.

3.6 Value and Future Work

As heavy annotations are a major weakness of the type system, future work has been trying to help the programmer more by inferring atomicity types from original programs [3, 17]. Another feature that can be helpful to programmers is the system’s ability to add locking calls automatically based on the programmer’s specifications, described in [10]. Another way to avoid annotations is not to use type system or static analysis at all. Other methods to verify atomicity include dynamic analysis [7, 15], hybrid analysis [21], model checking [8], and theorem proving [9]. Using dynamic analysis, for example, is good for legacy code as well, since it is more or less automatic.

Another weakness of the type system in [1] is its limited expressiveness. There has been future work by the same author addressing this issue using effect system [18], which extends the notion of atomicity beyond reducible methods. Of course, the other approaches mentioned above (e.g. dynamic analysis) often can verify more precise specifications too.

It is worth to note that even though the paper [1] is not the first to define the well-known concept of atomicity (it actually uses a more restricted meaning of the term), it is the first to use a static program analysis method to solve the problem of atomicity verification. It also succeeds in finding true atomicity violations in the JDK libraries, thereby proves its usefulness. With the rise of software transactional memory (STM) where programmer gets small-scale atomicity for free, the type system in [1] is still useful but it becomes less valuable. However, it can work together with a transactional memory system/library to guarantee atomicity and hence reduce the workload on the STM system [21].

4. CONCLUSION

In this report we have summarized and analyzed the paper "A Type and Effect System for Atomicity" by Flanagan and Qadeer [1]. We have justified the motivations behind the work and the choice of method, verified their claims, gave more contexts to their discussions, provided more explanations for some of the points briefly stated in the paper, analyzed in depth some of the paper's shortcomings, and noted future work that extended the paper. We can conclude that the paper is influential in the sense that it spawns a large body of future work focusing on the same or related problems. The method used in the paper which is the atomicity type system, although an extension from previous work, is proven useful and will remain useful (though less so) in the foreseeable future. However, the type system is not without its inherent shortcomings, some of which not given sufficient discussions in the paper. The shortcomings are not trivial to overcome by just extending the type system, thus somehow limit the usefulness of the method in practice.

5. REFERENCES

[1] C. Flanagan and S. Qadeer. A type and effect system for atomicity. *In Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 338—349, 2003.

[2] R. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717—721, 1975.

[3] C. Flanagan, S. N. Freud, M. Lifshin, and S. Qadeer. Types for atomicity static checking and inference for Java. TOPLAS, 30(4):1—53, 2008.

[4] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI 00: Programing Language Design and Implementation*, pages 219—232. ACM Press, 2002.

[5] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI '06*, pages 308—319, New York, NY, USA, 2006.

[6] Shan Lu, Joe Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access-interleaving invariants. In *proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems*, October 2006.

[7] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256—267, New York, NY, USA, 2004.

[8] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *International Conference on Verification, Model Checking and Abstract Interpretation*, pages 175—190, 2004.

[9] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[10] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, pages 346—358, 2006.

[11] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Hune 2005.

[12] D. L. Detlefs, K. R. M. Leino, and C. G. Nelson. Wrestling with rep exposure. Research Report 156, DEC Systems Research Center, July 1998.

[13] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 02: Object-Oriented Programming, Systems, Lanaguages, and Applications*, pages 211—230. ACM Press, 2002.

[14] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12—23, 2001.

[15] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Third Workshop on Runtime Verification (RV03)*, volume 89(2), 2003.

[16] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods (SEFM)*, 2005.

[17] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *Workshop on Types in Language Design and Implementation*, pages 47—58, 2005.

[18] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(3):275—291,2005.

[19] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, Jun. 2005.

[20] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Integrated dynamic and static analysis for atomicity violations. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2009.