Applying Static Analysis to Large-scale, Multi-threaded Java Programs

Cyrille Artho and Armin Biere
Swiss Federal Institute of Technology
Institute of Computer Systems
ETH Zentrum, RZ H, CH-8092 Zürich, Switzerland
{artho,biere}@inf.ethz.ch

Abstract

Static analysis is a tremendous help when trying to find faults in complex software. Writing multi-threaded programs is difficult, because the thread scheduling increases the program state space exponentially, and an incorrect thread synchronization produces faults that are hard to find.

Program checkers have become sophisticated enough to find faults in real, large-scale software. In particular, Jlint*, a very fast Java program checker, can check packages in a highly automated manner. The original version,* Jlint1*, still lacked full support for synchronization statements in Java. We extended Jlint1's model to include synchronizations on arbitrary objects, and named our version* Jlint2*. Our statistical analysis proves that these extensions are relevant and useful. Applying Jlint2 to various large software packages, including commercial packages from Trilogy, found 12 faults, two of which related to multi-threading.*

1 Introduction

In recent years, the use of multi-threaded software is becoming increasingly widespread. Especially for large servers, multi-threaded programs have advantages over multi-process programs: Threads are computationally less expensive to create than processes, and share the same address space.

Before multi-threading was part of programming languages, it usually could only be used via libraries (e.g., the "POSIX threads" in C or C++ [25]). The Java programming language includes multi-threading as a language feature. Other languages, such as Ada [33], have similar features, but never became as widespread as Java. In addition to easier use of multi-threading, hardware support in the form of *symmetric multi processing* (SMP) [32] is becoming more powerful and cheaper.

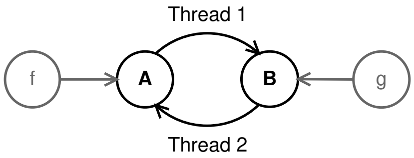
Trilogy, a large software company in Austin, Texas, heavily uses multi-threaded Java programs; therefore its developers have to cope with multi-threading problems on a daily basis [4]. The main motivation for writing multi-threaded programs is that Java makes it easy; despite this, writing correct multi-threaded software, as the following section shows, is still very hard. This research started as an investigation of the usability of program checkers in this commercial environment.

1.1 Problems with multi-threading

Non-trivial multi-threaded programs require *synchronization* between threads. Several variants have been proposed [34, 35, 36]: A *semaphore* only allows a certain number of threads to enter a critical section and execute a specific block at any given time [34]. A *monitor* (see Section 1.1) only allows a single thread to execute a certain block at any time [36]. This is a special case of a semaphore and used in Java for synchronization. The entrance to the monitor is guarded by a *lock;* only a thread holding that lock can enter the monitor, and only one thread may hold the lock at a time. In Java, one lock is associated to each object and class [30].

When synchronization between threads is required, the two most common problems are *race conditions* and *deadlocks.* This paper focuses on these two issues, omitting other problems, such as livelocks.

A race condition occurs when several threads access the same resource simultaneously, without sufficient protection. To ensure the absence of a race condition for an object o, one examines the *lock set* L_o, the set of locks held by a thread when accessing o. A checker has to ensure that an object o is 1) only read when a thread holds at least one lock in L_o and 2) only written when a thread holds *all* locks in L_o [12]. A race condition in a program means that operations using the shared value may yield inconsistent results.



Thread 1

```
method f() {
  synchronized(A) {
    synchronized(B) { }
  }
}
```

Thread2

```
method g() {
  synchronized(B) {
    synchronized(A) { }
  }
}
```

Figure 1. A deadlock and its lock graph.

A deadlock is the situation where threads wait for each other while holding, and not relinquishing, resources that another thread needs to continue. The *lock graph* is defined by the order in which threads access the locks. If the lock dependencies in the lock graph can be arranged as a partial order, i.e., no cycles are present, then the program will not have a deadlock. The opposite is not always true. Figure 1 depicts a constellation of two threads competing for two resources, illustrated by an incomplete Java program. If both threads hold one lock each, none of them can continue because the second lock they need is already taken.

1.2 Structure of this paper

Existing work is shown in Section 2. Section 3 describes how multi-threading is implemented in Java. An overview of Konstantin Knizhnik's Jlint1 is given in Section 4. Our statistical analysis, described in Section 5, helped us to focus on the most important problems. After a preliminary analysis of various static checkers, we decided to use Jlint1 and enhance its functionality, creating Jlint2. Its extended model is documented in Section 6. Section 7 shows the results from using Jlint on Trilogy's and other code.

2 Existing work

Classical reliability testing consists of running the program for an extended period of time, to see if it fails. Checking program properties at run time is called *dynamic checking*. Traditional testing is weak for multi-threaded programs, because the execution is *non-deterministic:* since the thread schedule cannot be influenced, the outcome of a program may vary even if the input is the same. The number of potential thread schedules is *exponential* in the number of threads, so the schedules cannot be tested exhaustively. In order to solve this problem, the tool *Rivet* creates all *relevant* schedules, schedules that produce a different outcome [13]. Another related attempt is tracking the history of a program in order to deduce *possible* behaviors of the program under several schedules [2, 15].

Static checking analyzes program properties by using compile-time information [7]. Traditionally, a model of the program had been created manually, in form of a mathematical specification. For the last few years, models have been successfully been generated automatically, from the program source or object code. Once the model is created, various approaches exist for checking those.

A coarse distinction can be made between static checkers, between *model checkers* and *theorem provers*: Model checkers operate directly on a model of the program, such as a call graph or a finite state machine [10]. Such a model may be an abstraction of the *control flow* or *data flow* of a program and is commonly expressed in some variant of temporal logic [37]. Approaches based on model checking include the tools *Bandera, JPF,* and *Jlint.* Bandera is a publicly available model checker; it encompasses several tools that tackle different parts of program verification [11, 16]. *JPF(2)* [20, 21] is the second generation of a Java program checker by NASA. Bandera and JPF both work in conjunction with the *SPIN* model checker [6]. Since Bandera and JPF became available too late to be included in our preliminary tool evaluation, we focused on Jlint. Jlint is a fast, publicly available checker [1]. In this paper, we will always use Jlint when referring to common features of Jlint1 and Jlint2. Jlint was used for the project described here, and extended to become Jlint2.

Theorem provers require a translation of the program into logic formulae, in first order or second order logic. These formulae are then processed by a theorem prover. It should be noted that the two approaches are often combined, so the boundaries are blurring [18]. *ESC/Java* is the second generation of a theorem prover designed for *Modula* programs, and includes Java, too [12]. Work is also in progress in the *SLAM* project, where several approaches are tried with the goal of ensuring reliability in software [19, 22].

3 Multi-threading in Java

The Java programming language has special support for multi-threaded programming, most importantly, the `Runnable` interface and the `Thread` class, as well as keywords and methods for communication between objects (`synchronized`, `wait`, `notify`). This paper focuses on `synchronized` statements. There are two types of synchronization:

In a `synchronized(resource)` *block*, the resource is always given explicitly. Before the block can be executed, the current thread has to obtain, or already own, the lock on `resource`. Only when a thread owns the lock, it can execute the block guarded by it. The lock is released when the synchronized block or method is exited. In case a lock is obtained several times, releasing it will only decrement a counter; only when that counter is zero, the ownership on the lock will actually be relinquished.

If a *method* is synchronized, the current thread will obtain a lock on the current instance (this) at the beginning of the method, and release it before returning. In static methods, a lock associated with that class is used [30].

In the Java Virtual Machine, these two kinds of synchronization are implemented in different ways: Synchronized blocks are implemented with special instructions. Synchronized methods are indicated by a special flag in the method descriptor; the virtual machine itself has to take care of acquiring and releasing the lock [31]. This is due to historical reasons; synchronized blocks are more general, so that mechanism could be used for all cases.

4 Jlint

In this paper, we investigate Jlint [1], a static program checker created by Konstantin Knizhnik. Jlint checks Java bytecode for inconsistencies, which include null pointers, array bounds violations, inheritance and finally, being the focus of this publication: multi-threading problems. Jlint's functionality is hard-coded into the program, and cannot be influenced by annotations or templates. However, command line switches can be used to enable or disable certain categories of warnings, or entire groups of them. This makes it very fast and easy to use. Because Jlint works on the compiled Java class files, it can also check libraries where the source code is not available. This feature is also heralded as important for *Purify* in the context of dynamic checking [23].

Jlint works in two passes: in the first pass, all class files are read into memory. Most checks are done locally, while each method is processed. During this first pass, the call graph of the analyzed classes is built. This call graph includes certain extra information, such as whether methods are synchronized. This elegant model is sufficient for checking deadlocks among synchronized methods. Jlint also builds the accessor dependency graph, which enables it to check for race conditions. Despite its limitations, Jlint is in practice as good as any other currently available program at checking multi-threading problems, as our statistical analysis shows (see Section 5).

5 Statistical analysis

Our statistical analysis shows how frequent the different Java synchronization mechanisms are in practice. It helped us to focus our improvements of Jlint on the most common synchronization mechanisms.

The following packages, whose source code encompassed nearly a million lines of code together, were analyzed: all class files coming with Sun's Java Developer Kit version 1.3, Doug Lea's concurrency package [3], a data warehousing tool developed at the ETHZ [5], and Trilogy's core packages. The analysis comprised two major steps:

The first part was a count of synchronized methods and blocks. The relative numbers are of particular interest, because they indicate the importance of synchronized blocks compared to synchronized methods.

A closer analysis of synchronized blocks followed. The interesting cases are synchronizations on this, the current instance, or on a "constant" field whose content does not change after initialization, i.e., outside the constructor. Within the constructor, the field is protected against concurrent access because it is still invisible to other threads.

The foremost result was that synchronization statements are fairly rare in large Java programs. This makes sense, since a good design tries to keep the number of synchronization points low, and restricts them to as few classes as possible. Moreover, classes implementing a wrapper functionality need fewer synchronization statements, because the synchronization is mostly performed in the underlying software.

As Table 1 shows, synchronized methods are more common than synchronized blocks, which sum up to 46 %. In synchronized blocks, synchronizations on this and fields whose value does not change after initialization are very common. These results were surprisingly consistent across all analyzed modules, although some packages use almost only synchronized methods.

Category	#	%
synchronized methods	1351	53.76
synchronized(this)	209	8.32
synchronizations on a "constant" field	582	23.16
other cases	371	14.76
Total	2513	100.00

Table 1. Frequency of different synchronizations in the analyzed Java packages.

All in all, the trend is evident that simpler cases prevail, even in complex packages. Of course, the statistics only indicate the frequency of different synchronization mechanisms, not their complexity. It is not certain whether faults are more likely in the more complex cases.

6 Call graph extension

Jlint's model allows checking for deadlocks among synchronized methods. Figure 2 shows such a case, where `synchronized` methods of two classes are used recursively. Two threads calling the methods `A.f` and `B.f`, respectively, can cause a deadlock, because they both have to obtain a lock on the classes `A` and `B` in order to complete the recursive method call.

```
class A {                        class B() {
  static synch. f() {              static synch. f() {
    B.g();                           A.g();
  }                                }
  static synch. g() { }            static synch. g() { }
}                                }
```

Figure 2. Two recursive synchronized method calls causing a deadlock.

Such `synchronized` methods only constitute 54 % of all synchronization statements used in Java. Therefore an extension for supporting the other 46 % of all cases, `synchronized` blocks, would improve Jlint's fault finding capabilities significantly.

When analyzing `synchronized` blocks, solving the *aliasing problem* is the essential point of the analysis. One has to know whether two references point to distinct objects [8]. Nested data structures and the fact that objects keep their state during execution make this problem very hard [9].

In most cases, the shared resource that is locked on is either a (`static`) class variable, or an instance variable. In both cases, the value is assumed to be constant during the execution of the thread – the field is usually never changed after initialization. As the statistical analysis shows, 85 % of all variables that are synchronized on, including the current instance, `this`, in `synchronized` blocks and methods, are initialized in the constructor and stay unchanged afterwards. Therefore, the extension to `synchronized` blocks, solving the aliasing problem only for fields (or attribute variables) of single instances, but across method calls, would increase the applicability of Jlint from 54 % to 85 %.

Inside the virtual machine, `synchronized` blocks are implemented with two special bytecode operations: `monitorenter` and `monitorexit` [31]. Both operations take the top element from the stack as their argument. Therefore, the *alias* of each value on the stack needs to be tracked during execution. This was done by extending Jlint1's data structure and adding extra instructions for analyzing operations that alter a value on the stack, and also for the `new` operation. This instruction reserves memory for a new object instance and pushes its reference onto the stack. Our extension only covers the alias of any stack element referring to a field or a local variable; it does not cover equalities among different fields or across classes, e.g., arguments of function calls.

With the aliasing problem solved – at least for the stack values within the virtual machine – the question is how to extend the existing call graph model to include `synchronized` blocks as well. In our approach, these blocks are treated like method calls: for each `synchronized` block in a method, a call to a *pseudo method* `<synch>` is added to the call graph. Nesting of `synchronized` blocks is modeled with nested pseudo method calls.

```
static f() {
  synch.(a) {
    synch.(b) { g(); }
}
```

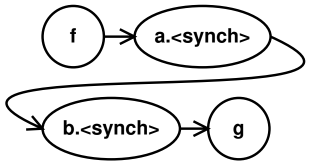


Figure 3. Jlint2's extended call graph model.

In the example in Figure 3, method `f` acquires a lock on the variable *a*. The edge

$$\mathtt{f} \rightarrow \mathtt{a}.\langle\mathtt{synch}\rangle$$

in the call graph represents this synchronization. The same method acquires another lock *b within* the first synchronized block. The edge

$$\mathtt{a}.\langle\mathtt{synch}\rangle \rightarrow \mathtt{b}.\langle\mathtt{synch}\rangle$$

is added for that block. If the same lock is released and then re-acquired, the same pseudo method is used; only the *nesting* of method calls and blocks matters, not their *order*. Moreover, the *type* of the lock is unimportant, because its name is unambiguous. Method calls from within `synchronized` blocks are treated accordingly: The call from the innermost block to method `g` is modeled by the edge

$$\mathtt{b}.\langle\mathtt{synch}\rangle \rightarrow \mathtt{g}$$

Our extension combines the nesting of `synchronized` methods and subsequent method calls with `synchronized` blocks. The final model, as it was implemented, includes the full class names in the call graph. Method calls from `synchronized` blocks to other classes are not included in the call graph because the call graph would grow too big for the current implementation of Jlint. This restriction confines deadlock detection to deadlocks within and across methods of the same class. Jlint cannot detect deadlocks across different classes, except for deadlocks across `synchronized` methods. Also, inheritance is not fully covered, as the behavior of superclass methods is assumed to be consistent with inherited methods, with respect to synchronization. Dynamic class loading is not supported yet. A future extension could include such features.

7 Application of Jlint

This section summarizes the faults that Jlint2 found in 15 small example programs, and various large software packages. The test examples served to assess Jlint2's capabilities, and also to evaluate the extensions. For the large software packages, the goal was to find as many faults as possible, both multi-threading and other faults. The large software packages varied between 25,000 and 100,000 lines of code each.

7.1 Test examples

Jlint2 was evaluated with 15 small test examples: The first six examples exhibit deadlocks using incorrect locking orders, or problems with `wait` and `notify`, where a thread calling these methods holds too many locks. Another example contains a subtle race condition due to incomplete locking, as shown in Derek Bruening's SplitSync [13]. The remaining examples comprise eight complex locking schemes. Four examples are variations of shared buffer implementations, with producer and consumer tasks. Two of these implementations are correct. In addition to that, three solutions to the *Dining Philosophers problem* [26], one of which is faulty, were checked. The ESC/Java tree example [12] can also be counted towards this category, where a complex nesting of locks is given by a recursive data structure, and therefore cannot be fully evaluated at compile-time.

In four of these 15 cases, Jlint1, which only supported `synchronized` methods, successfully detected the deadlock in the program and did not issue a warning when the program was correct (see Table 2). Jlint1 failed to detect the race condition in the SplitSync program. The analysis of the ESC/Java tree example seemed successful, but only because Jlint1 ignored the critical part of the program. The output about the seven complex examples was inconclusive, and clearly showed that locking schemes operating on complex compound data structures cannot be analyzed by any of the static checkers mentioned in Section 2 yet. Jlint1 gave a correct output in four cases; in remaining four cases, it issued a spurious warning or failed to detect a fault in the program.

	Jlint1	Jlint2
Correct output	4	6
Missed fault/spurious warning	4	2
Beyond scope of Jlint	7	7

Table 2. Improvements for multi-threading problems made with Jlint2.

With support for `synchronized` blocks, Jlint2 is able to successfully analyze all six simpler deadlock examples. There is no support for special race conditions such as the one shown in SplitSync, so Jlint2 still cannot detect the fault in this case. The ESC/Java example is no longer ignored, but too complex for Jlint2 to analyze, so Jlint2 now issues a spurious warning. Jlint2's capability to detect deadlock problems is significantly stronger than Jlint1's, as the table shows. In the remaining part of this section, we demonstrate Jlint2's results for real world programs.

7.2 Sun's JDK packages

Jlint2 as such is capable of analyzing large packages, such as Sun's JDK packages. However, the number of potential deadlock warnings was very high (several thousand). By filtering most warnings, the number could be reduced to a few hundred, which is manageable. Because the Java Foundation Classes are quite mature in version 1.3, and we wanted to focus our efforts elsewhere, Jlint2's output for these classes was not reviewed.

7.3 Doug Lea's concurrency package

Doug Lea's concurrency package implements various more advanced Java concurrency mechanisms, such as shared-read locking [3]. The number of spurious warnings produced for this package was very high. Most of these warnings were caused by Jlint's lack of a complete control flow analysis, and could be dismissed quickly after a manual review. Another large part of very similar, redundant warnings were caused by a fault in Jlint2, which could not be reproduced when running it on fewer or simpler modules. After subtracting those, 29 warnings remained; 26 were potential deadlock warnings. Because they referred to interactions between methods and `synchronized` blocks, they are difficult to analyze for someone not familiar with the package. It is likely that the concurrency package is correct, and these warnings show cases where static analysis is very difficult. If a call graph browsing tool had been available, reviewing the warnings would have been lot easier.

Jlint2 successfully found three potential race conditions. In one case, a node was inserted into a list of locks. This is a difficult situation for static analysis, because a program checker has to understand how the nodes in a list are connected. In two other cases, the correct operation depends on other methods or on the values of certain counters. To our knowledge, no currently available tool is capable of analyzing such properties automatically.

7.4 ETH data warehouse tool

Jlint2 was very successful in analyzing the data warehousing tool from ETH, which serves to analyze astronomical observations [5]. Among other problems, Jlint2 found two interesting race conditions. In a class implementing a resource pool, each access to a *single* element was guarded by synchronization. However, an operation re-allocating the *entire* pool, to change its size, was lacking such an access protection.

```
class ResourcePool {
  Object[] resources = new Object[100];

  public setSize(int newSize) {
    resources = new Object[newSize];
      // race condition!
  }
}
```

Figure 4. A race condition when reallocating an array object.

Warning category	#	C
Lock variable change outside constructor or synchronization.	3	0
Missing `super.finalize()` call.	12	1
Possible deadlock: loop in locking graph (`synchronized` methods).	13	0
Possible `null` pointer reference because parameter is not checked.	23	6
Possible `null` pointer reference because of unexpected input or state.	6	2
Other	4	1
Total	61	10

Table 3. Faults diagnosed and confirmed by Jlint2 in Trilogy's software.

Because the object itself changes in Figure 4, a simple `synchronized(resources)` will not work. The solution is to synchronize on an additional object when changing the resource itself, such as `this` or an extra `Object resourceLock`.

7.5 Trilogy's middleware

The analyzed packages of Trilogy's software are all server-side engines, running as middleware between other tiers. They are part of Trilogy's MCC e-commerce suite, and similarly to most server-side Java software, heavily multi-threaded. This made them a perfect target for our static checker. Moreover, these packages are rather large, ranging from 25,000 to over 100,000 lines of code.

The overall distribution of synchronization statements was very similar to the total in Section 5. The core engine utilized most synchronization statements, while other engines, embodying business logics, used fewer. The more often synchronizations were needed in a certain package, the more likely was the occurrence of complex cases. For instance, inter-object interactions are difficult to analyze.

In one of Trilogy's packages, Jlint2 discovered a race condition. In most packages, the race condition warnings were too numerous to be analyzed, because Jlint does not understand the concept of shared-read access. If the only write access is inside the constructor, then suppressing race condition warnings is simple. Other cases will need a more detailed lock set analysis (see Section 1.1, [15]), in order to ensure that no race condition occurs.

Table 3 summarizes the kinds of faults detected by Jlint2. The numbers in the middle column already exclude multiples of warnings referring to the same issue at different lines in the source code, but include spurious warnings. The second column states the confirmed warnings, i.e., actual faults. Other than the race condition mentioned above, none of the multi-threading warnings could be confirmed as a fault.

Apart from race conditions, Jlint also found many `null` pointer problems and one `int` overflow, where the value was left shifted by 32 *before* it was converted to a `long`, and a few `super.finalize` calls that were not included. In total, ten confirmed faults were found from these warnings.

The tested packages were fairly mature and had been in use for quite some time; therefore it is possible that they will not have any deadlocks anymore. It is also likely that in more mature code, the remaining faults reside in the communication and control flow between modules; Jlint's analysis cannot cover this.

7.6 Usefulness of Jlint

Besides the impressive number of confirmed faults found, the effort required to find them, using a static checker, is of course very important in judging Jlint2's value as a utility. The time needed to *install* and *master* a tool influences how readily developers will accept it. Jlint2 is easy to install, and most of the warnings are self-explanatory. It does not require any annotations, so it can directly be used on production code. As a consequence of this, we have chosen Jlint2 at Trilogy, where the developers do not have time to learn a complex tool or add annotations to the source code.

The *performance* of a tool is also important. Jlint2 runs very fast. For checking all `java.*.*` class files coming with Sun's JDK 1.3, being 250,000 lines of code, Jlint2 requires little more than a second on a Pentium III/700 MHz! This leaves room for more sophisticated analysis algorithms.

Ideally, each warning refers to a *fault* in the software. This would, however, restrict the scope of current static checkers too much [12]. Sometimes, spurious warnings were given because Jlint could not deduce enough context from the program source code; in other cases, Jlint's model was too simple. Jlint does not support shared-read access and complains about possible race conditions in such cases. Because of the huge number of warnings, usually several hundred per package, certain categories of warnings had to be ignored. This reduced their number to usually 20 – 30 per package, which is quite manageable. Out of these, roughly 10 % were confirmed as faults.

The *review* of the warnings is an important step as well, because most warnings do not correspond to a fault. Moreover, it is usually left to the developer to think of a fix for the detected faults. This is sometimes very easy: warnings about local properties, such as results of local `null` pointer analyses, are usually reviewed within seconds, and fixed quickly. Race conditions also tend to be easy to review. However, for the analysis of possible deadlocks, the developer has to understand synchronization dependencies between objects. Because Jlint cannot print the part of the call graph that would illustrate the conflict, and does not show the lock set of each thread when there could be a deadlock, reviewing deadlock warnings manually is quite difficult. For someone who is working on the code, it is usually still possible to examine the warnings in a rather short time. Someone who is not directly involved with the code cannot do this, though. An *inspection tool,* such as a call graph browser, would remedy this situation.

8 Conclusions

This paper showed how we successfully used Jlint2, a static Java program checker, to analyze large-scale, industrial software. Static analysis is a promising way of tackling multi-threading problems.

We showed how `synchronized` blocks in Java can be modeled as special methods. This allowed us to include `synchronized` blocks in the method call graph and treat method calls and synchronizations uniformly. As our statistical analyses showed, even an incomplete alias analysis is sufficient to cover 85 % of all cases.

We had extended the original Jlint1 and created our extended version, Jlint2. Jlint2's finer grained deadlock checks found only minor faults – possibly because the software checked was already fairly mature. Nevertheless, the 15 test examples showed that Jlint2 is now capable of finding a much wider range of faults than before. This is a first, important step towards covering the full semantics of multi-threading. It remains to be seen whether the hardest 15 % of all cases are the most error prone.

In order to cover the remaining, hard, cases, a more complete alias analysis, and a more flexible framework, will be needed. Including more context in the analysis will eliminate certain spurious warnings. However, even the current Jlint2 is a valuable tool for finding faults early, or for preparing a code review.

References

[1] Jlint1 `http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm`, Jlint2 `http://artho.com/jlint/`

[2] VisualThreads `http://www5.compaq.com/products/software/visualthreads/`

[3] Doug Lea's concurrency package `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`

[4] Trilogy Inc. `http://www.trilogy.com/`

[5] HEDC - HESSI Experimental Data Center `http://www.hedc.ethz.ch/`

[6] G. J. Holzmann. *Design And Validation Of Computer Protocols.* Prentice Hall, USA 1991.

[7] A. T. Chamillard. *An Empirical Comparison of Static Concurrency Analysis Techniques.* PhD thesis. University of Massachusetts at Amherst, May 1996.

[8] W. Landi and B. Ryder. Pointer-induced aliasing: A Problem taxonomy. *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* 93–103, January 1991.

[9] J. Hogg, D. Lea, A. Wills, D. deChampeaux, R. Holt. The Geneva Convention On The Treatment of Object Aliasing. *OOPS messenger 1992,* USA 1992.

[10] E. M. Clarke, O. Grumber, and D. A. Peled. *Model Checking.* MIT Press, USA 1999.

[11] J. Hatcliff and O. Tkachuk. *The Bandera Tools for Model-Checking Java Source Code: A User's Manual.* Kansas State University, USA 2001.

[12] K. Rustan M. Leino, G. Nelson, and J. B. Saxe. *ESC/Java User's Manual.* Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[13] D. Bruening. *Systematic Testing of Multi-threaded Java Programs.* Master's Thesis, MIT, May 1999.

[14] P. Godefroid. *Model Checking for Programming Languages using VeriSoft.* Bell Laboratories, Lucent Technologies, USA 2000.

[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. *Proceedings of the 16th ACM Symposium on Operating System Principles,* 27-37, Saint Malo, France, October 1997.

[16] J.C. Corbett, M.B. Dwyer, J. Hatcliff, and R.. Bandera: A source-level interface for model checking Java programs. *Proc. 22nd International Conference on Software Engineering,* June 2000.

[17] K. Rustan M. Leino, J. B. Saxe, and R. Stata. *Checking Java programs via guarded commands.* Technical Note 1999-002, Compaq Systems Research Center, May 1999.

[18] D. L. Detlefs, K. Rustan M. Leino, G. Nelson, and J. B. Saxe. *Extended Static Checking.* Compaq Systems Research Center, December 1998.

[19] T. Ball, S. Chaki, S. K. Rajamani. Parameterized Verification of Multithreaded Software Libraries. *TACAS 2001,* LNCS 2031, April 2001, 158-173.

[20] W. Visser, K. Havelund, G. Brat, S. Park. Model Checking Programs. *International Conference on Automated Software Engineering,* September 2000.

[21] G. Brat, K. Havelund, S. Park, W. Visser. Java PathFinder, Second Generation of a Java Model Checker. *Workshop on Advances in Verification,* July 2000.

[22] T. Ball, S. K. Rajamani. *Boolean Programs: A Model and Process for Software Analysis.* MSR Technical Report 2000–14, USA 2000.

[23] R. Hastings and B. Joyce. Purify: fast detection of memory leaks and access errors. *Proceedings of the Winter Usenix Conference,* 1992.

[24] J. C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE transactions on software engineering, Vol. 22,* No. 3, March 1996.

[25] IEEE. *Threads Extension for Portable Operating Systems (Draft 6).* February 1992, P1003.4a/D6.

[26] A. Silberschatz, G. Gagne, P. Baer Galvin. *Applied Operating Systems Concepts.* 1st edition, USA 2000.

[27] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* 2nd edition, Addison-Wesley, USA 1999.

[28] S. Oaks, H. Wong. *Java Threads.* O'Reilly, USA 1997.

[29] D. Flanagan. *Java In A Nutshell.* 3rd Ed.. O'Reilly, USA 1999.

[30] J. Gosling, B. Joy, G. Steele, G. Bracha. *The Java Language Specification.* 2nd Ed., Addison-Wesley, USA 2000.

[31] T. Lindholm, F. Jellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, USA 1999.

[32] Intel Corporation. *Pentium Pro family developer's manual, volume 3: Operating system writer's manual.* Intel Corporation, 1996, Order number 242692.

[33] United States Department of Defense. *Reference Manual for the Ada programming language.* DoD, Washington, D.C., January 1983. ANSI/MIL-STD-1815A.

[34] E. W. Dijkstra. Co-operating sequential processes. *Programming Languages,* F. Genuys, Ed. Academic Press, New York, 1968, 43-112.

[35] C. A. R. Hoare. Towards a theory of parallel programming. *Operating Systems Techniques,* Academic Press, New York, 1972, 61–71.

[36] B. Hansen, P. The programming language Concurrent Pascal. *IEEE Trans. Software Eng. 1, 2* (June 1975), 199–207.

[37] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science 13,* 45–60.