Model Checking of Concurrent Algorithms: From Java to C

Cyrille Artho[1], Masami Hagiya[2], Watcharin Leungwattanakit[2], Yoshinori Tanabe[3], and Mitsuharu Yamamoto[4]

[1] Research Center for Information Security (RCIS), AIST, Tokyo, Japan
[2] The University of Tokyo, Tokyo, Japan
[3] National Institute of Informatics, Tokyo, Japan
[4] Chiba University, Chiba, Japan

Abstract. Concurrent software is difficult to verify. Because the thread schedule is not controlled by the application, testing may miss defects that occur under specific thread schedules. This problem gave rise to software model checking, where the outcome of all possible thread schedules is analyzed.
Among existing software model checkers for multi-threaded programs, Java PathFinder for Java bytecode is probably the most flexible one. We argue that compared to C programs, the virtual machine architecture of Java, combined with the absence of direct low-level memory access, lends itself to software model checking using a virtual machine approach. C model checkers, on the other hand, often use a stateless approach, where it is harder to avoid redundancy in the analysis.
Because of this, we found it beneficial to prototype a concurrent algorithm in Java, and use the richer feature set of a Java model checker, before moving our implementation to C. As the thread models are nearly identical, such a transition does not incur high development cost. Our case studies confirm the potential of our approach.

1 Introduction

Concurrent software is often implemented using threads [26] to handle multiple active units of execution. Because the thread schedule is not controlled by the application, the usage of concurrency introduces implicit non-determinism. Without proper guards against incorrect concurrent access, so-called race conditions may occur, where the outcome of an two concurrent operations is no longer well-defined for all possible interleavings in which they may occur.

In software testing, a given test execution only covers one particular instance of all possible schedules. To ensure that no schedules cause a failure, it is desirable to model check software. *Model checking* explores, as far as computational resources allow, the entire behavior of a system under test by investigating each reachable system state [10], accounting for non-determinism in external inputs, such as thread schedules. Recently, model checking has been applied directly to software [5, 6, 8, 11–13, 27, 28]. Initially, software model checkers were *stateless:*

after backtracking, the program is restarted, and the history of visited program states is not kept [13]. This makes the model checker more light-weight, at the expense of potentially analyzing redundant states. *Stateful* model checkers keep a (possibly compressed) representation of each visited state. This allows a model checker to backtrack to a previously visited state without having to re-execute a program up to that point, and also avoids repeated explorations of equivalent (redundant) states.

Certain programming languages, such as C [17] or C++ [25], allow direct low-level memory access. *Pointer arithmetic* allows the usage of any integer offset together with a base pointer, making it impossible to guarantee memory safety in the general case. Memory safety implies that memory which is read has been allocated and initialized beforehand. Program behavior is undefined for unsafe memory accesses. More recently developed programming languages, such as Java [14], Eiffel [21], or C# [22], can restrict memory accesses to be always safe. This feature, in conjunction with garbage collection, relieves the developer from the burden of manual memory management. It also makes it easier to define the semantics of operations, and to perform program analysis.

Embedded systems may be implemented on a platform supporting either Java or C. Due to its managed memory, Java is easier to develop for, but the ease comes at the expense of a higher memory usage. Embedded systems, or core algorithms used therein, may therefore be prototyped in Java, and moved to C if resource constraints require it. In such cases, it is useful to verify the Java version in detail before translating it to C for further optimization.

In addition to memory safety, the object-oriented semantics and strong typing of many more recent programming languages facilitate the analysis of the heap structure. This enables efficient execution inside a virtual machine [20] and also allows program states to be mutated and compared easily. Software model checking benefits greatly from this, as stateful model checking can be implemented much more readily for such programming languages. Besides the promise of avoiding redundant states, the statefulness of a model checker can also be exploited for programs that utilize network communication. The results of network communication can be cached, making analysis orders of magnitude more efficient than in cases where the entire environment has to be restarted [4]. Such an input/output cache has been implemented for the Java PathFinder model checker, which analyzes Java bytecode [27]. That cache is one of about 15 available extensions for that model checker, making it much more flexible and feature-rich than its counterparts for C or C++.

Finally, it is easier to debug a concurrent implementation when memory safety is not an issue; the developer can focus on concurrency aspects without worrying about memory safety. Because of this, and the flexibility of Java model checkers, we argue that it is often beneficial to develop a concurrent algorithm in Java first. After analysis, the Java version can be readily converted to C (or C++) using the pThreads thread library. We have successfully applied this paradigm to multiple implementations of concurrent algorithms. Compared to a compilation of Java to machine code, a source-level translation has the advantage that stack memory for non-primitive data types and other optimizations are available. A verification of the resulting C code detects possible translation errors.

Translations from a higher-level language to a lower-level one are common in *model-based code generation.* However, in that domain, more abstract languages such as state charts [15] are common. High-level models prevent low-level access conflicts but cannot be optimized for fine-grained concurrency in ways that Java or C code can. Our translation is on code level, because thread-level parallelism with the explicit usage of mutual exclusion through locking is still prevalent in implementations of concurrent systems.

Related work exists in translating C code to Java [16]. That translation considers self-contained programs and mostly targets the implementation of pointer arithmetic in C as arrays in Java. For concurrent programs, manual case studies have been performed on the conversion of a space craft controller written in C, to Java [7]. The Java version was developed for analysis purposes because no suitable tools for analyzing multi-threaded C software existed at that time. We advocate the reverse direction, a translation from Java to C, because the Java version can be more readily developed, given the higher automation of low-level tasks by the run-time environment, and because more powerful concurrency analysis tools are available.

The rest of this paper is organized as follows: Section 2 introduces threads in Java and C. Section 3 shows our mapping of multi-threaded Java code to C. Experiments are described in Section 4. Section 5 concludes.

2 Thread Models in Java and C

A *thread* is an independent unit of execution, with its own program counter and stack frame, and possibly a separate memory region (thread-local memory) [26]. It typically interacts with other threads through semaphores (signals), and ensures mutual exclusion by using monitors (locks). Threads are started by a parent thread, which is the only thread of execution in a process at its creation. Execution may merge with other threads by "joining" them, waiting for their termination.

The standard version of Java has built-in support for multi-threading. In Java, a thread has its own program counter and stack frame, but shares the main memory with all other application threads. Threads may run on one or more hardware processors, potentially by using time-slicing to support more threads than available processors [14]. The C programming language has no built-in support for threads [17]. However, on most modern operating systems, threads are supported by libraries following the POSIX threads (pThreads) standard [23].

As Java threads were designed with the possibility of an underlying pThreads implementation in mind, the two thread models have much in common. The specified memory model allows each thread to hold shared data (from main memory) in a thread-local cache. This cache may be out of date with respect to the copy in main memory. Whenever a lock is acquired or released, though, values in the thread-local cache are synchronized with the main memory [14].

This behavior of Java is similar to other programming environments, in particular typical environments supporting the pThreads standard. Furthermore, there are no constraints on the thread scheduling algorithm; while it is possible to set thread priorities, both the Java and the pThreads platforms do not have to adhere to thread priorities strictly [14, 23]. Most importantly, the rather unrestricted memory model used by Java does not imply the sequential consistency [18] of a sequence of actions within one thread. Instruction reorderings, which are performed by most modern hardware processors, are permitted, requiring a concurrent implementation to use locking to ensure mutual exclusion and correct execution. As a consequence of this, a concurrent program in Java needs to use the same safeguards that a concurrent program written in C uses.

Finally, variables in Java and C may be declared as `volatile`, disallowing thread-local caching of such values. Because read-and-set accesses of `volatile` values are not atomic, there exist few cases where they are actually used in practice. We do not cover `volatile` values further in this paper.

3 Mapping Java to C

We define a mapping of Java thread functions to C here. This allows a developer to write an initial version of a concurrent algorithm in Java. The version can then be transformed to C, for example, if the memory requirements of Java may not be fulfilled by an embedded platform. The translation is discussed both in terms of differences in the concepts and in the application programming interface (API) of the two implementations. A complete transformation of a program entails addressing other issues, which are mentioned at the end of this section.

3.1 Threads

Both in Java and C, threads are independent units of execution, sharing global memory. Data structures that are internal to a thread are stored in instances of `java.lang.Thread` in Java [14], and of the `pthread_t` data structure when using pThreads [23]. These data structures can be initialized via their constructor in Java or by setting attributes in pThreads. The functionality of the child thread is specified via inheritance in Java, and via a function pointer in C. These mechanisms correspond to the object-oriented paradigm of Java and the imperative paradigm of C, and can be easily transformed. Likewise, functions to start the child thread, join the execution of another thread (awaiting its termination) or terminate the current thread, are readily translated (see Table 1).

3.2 Locks

Some of the concurrent functionality of Java cannot be mapped isomorphically. Specifically, important differences exist for locking and shared conditions (semaphores). In Java, each object may be used as a lock; the initialization of the lock, and access to platform-specific lock properties, are not specified by

Table 1. Comparison between thread primitives for Java and C.

Function	Java	C
Thread start	`java.lang.Thread.start`	`pthread_create`
Thread end	end of `run` method	`pthread_exit`
Join another thread	`java.lang.Thread.join`	`pthread_join`
Lock initialization	implicit with object creation	`pthread_mutex_init`
Acquire lock	`synchronized` keyword	`pthread_mutex_lock`
Release lock	`synchronized` keyword	`pthread_mutex_unlock`
Lock deallocation	implicit with garbage collection	`pthread_mutex_destroy`
Initialize condition	conditions are tied to locks	`pthread_cond_init`
Wait on condition	`java.lang.Object.wait`	`pthread_cond_wait`
Signal established cond.	`java.lang.Object.notify`	`pthread_cond_signal`
Broadcast est. cond.	`java.lang.Object.notifyAll`	`pthread_cond_broadcast`
Deallocate condition	implicit with garbage collection	`pthread_cond_destroy`

Java code and happen internally in the Java virtual machine [20]. In pThreads, on the other hand, lock objects have to be created and initialized explicitly. Locks use the opaque C data type `pthread_mutex_t`, which is initialized through `pthread_mutex_init` and managed by the library. In Java, classes derive from base class `java.lang.Object` and carry their own data, in addition to the implicit (hidden) lock; in C using pThreads, application data and lock data are separate. Therefore, locks in Java have to be transformed in two possible ways, depending on their usage:

Lock only: Instances of `java.lang.Object` carry no user-defined data and may be used for the sole purpose of locking. They can be substituted with an instance of `pthread_mutex_t` in pThreads.

Locks combined with data: In all other cases, instances are used both as data structures and as objects. They have to be split into two entities in C, where application-specific data structures and pThread locks are separate.

Similarly, the syntax with which locking is used is quite different between the two platforms: In Java, a `synchronized` block takes a lock as an argument. The lock is obtained at the beginning of the block and released at the end of it. The current thread is suspended if the lock is already taken by another thread. Locks in Java are reentrant, i. e., nested acquisitions and releases of locks are possible. Furthermore, there exists a syntactic variation of locking, by annotating an entire method as `synchronized`. This corresponds to obtaining a lock on the current instance (`this`) for the duration of a method, as if a `synchronized` block spanning the entire method had been specified.

After transforming `synchronized` methods to blocks, lock acquisitions and releases in Java can be readily mapped to C. The beginning of a `synchronized` block is mapped to `pthread_mutex_lock`, using the lock corresponding to the argument to `synchronized`. Lock releases are mapped likewise (see Table 1). Reentrancy is supported by pThreads through a corresponding attribute. Finally, locks should be explicitly deallocated in C to prevent resource leaks.

```
                  Java                       |                  C
---------------------------------------------+---------------------------------------------
synchronized (lock) {                        |pthread_mutex_lock (&lock);
  while (!condition) {                       |while (!condition) {
    try {                                    |
      lock.wait();                           |  pthread_cond_wait (&cond_var, &lock);
    } catch (InterruptedException e) {       |  // explicit condition variable
    }                                        |  // of type pthread_cond_t
  }                                          |}
  assert (condition);                        |assert (condition);
  ... // condition established               |... // condition established
}                                            |pthread_mutex_unlock (&lock);
```

Fig. 1. Inter-thread conditional variables in Java and C.

3.3 Conditions

Efficient inter-thread communication requires mechanisms to notify other threads about important status (condition) changes. To avoid busy-waiting loops, Java and pThreads offer mechanisms to wait for a condition, and to signal that the condition has been established. The mechanism is similar on both platforms, with one major difference: In Java, locks are used as a data structure to signal the status of a shared condition. In pThreads, there is a need for a separate *condition variable*, in addition to the lock in question.

In Java, due to the absence of condition objects, there always exists a one-to-one relationship between locks and condition variables. In pThreads, several condition variables may relate to the same lock, a fact that is further elucidated below. Figure 1 shows how shared conditions are mapped. The condition itself is expressed through a boolean variable or a complex expression. If the condition is not established, a thread may suspend itself using `wait`, awaiting a signal. Both in Java and C, a lock has to be held throughout the process; Java furthermore requires to check for the presence of an `InterruptedException`, because a waiting thread may optionally be interrupted by another thread.

3.4 Possible implementation refinements for pThreads

There are a couple of differences between Java threads and POSIX threads that allow for a more efficient implementation in C, by exploiting low-level data structures that are not accessible in Java. This may be exploited when translating an algorithm from Java to C. As such a translation cannot always be done automatically, another verification against concurrency defects is advisable when optimizing the C/pThreads implementation.

- When using pThreads, the function executing in its own thread may return a pointer to a thread waiting for its termination. This allows a thread to return data directly, rather via other ways of sharing.
- Separate condition variables in pThreads (`pthread_cond_t`) enable a decoupling of related but distinct conditions. In the experiments, we describe a case where the Java version uses one lock to signal the emptiness or fullness

of a queue. In C, the two cases can be separated, which sometimes yields performance improvements.

- The pThreads library has a function `pthread_once`, for which no direct equivalent exists in Java. This mechanism allows a function to be executed at most once, resembling the way static initializers are used in Java to initialize class-specific data. Unlike static initializers, the execution of `pthread_once` is not tied to another event, such as class initialization.
- In pThreads, it is possible to forgo the acquisition of a lock when the lock is already taken, by using `pthread_mutex_trylock`. In some cases, the same effect may be achieved in newer versions of Java by checking if a particular thread already holds a lock (by calling `Thread.holdsLock`, available from Java version 1.4 and higher).

Furthermore, both platforms offer ways to fine-tune the performance of thread scheduling using specific API calls in Java, and via attributes in pThreads. This does not affect the correctness of algorithms, and is elided here.

Finally, newer versions of Java (1.5 and later) offer read-write locks (`java.util.concurrent.lock.ReentrantReadWriteLock`), and barriers (`java.util.concurrent.CyclicBarrier`), which facilitate the implementation of certain distributed algorithms. Equivalent facilities are provided by pThreads, as `pthread_rwlock_t` and `pthread_barrier_t`, respectively. The translation of these and other similar functions resemble the translations shown above, and are not described in further detail here.

3.5 Other mappings

It is possible to compile Java to machine code, or to automate the mapping of Java programs to C, but the result will not be efficient. Java allocates all non-primitive data on the heap, while C allows complex data to be allocated on the stack. Stack-based allocation requires no explicit memory management or garbage collection, and is more efficient than heap memory. Furthermore, if Java heap memory is emulated in C, that memory has to be managed by garbage collection as well. A manual translation therefore yields better results.

Among library functionality other than multi-threading, Java offers many types of complex data structures such as sets, maps, and hash tables. These have to be substituted with equivalent data structures in C, provided by third-party libraries. In our experiments, we used a publicly available hash table [9] as a substitute for the hash table provided by the Java standard library.

Finally, for programs using input/output such as networking, the corresponding library calls have to be translated. In these libraries, the Java API offers some "convenience methods", which implement a sequence of low-level library calls. The C version may require additional function calls and data structures.

4 Experiments

In our experiments, we verify two concurrent algorithms, and one concurrent client/server program. To our knowledge, no higher-level code synthesis approach supports all types of parallelism used, so we advocate a verification of the implementation itself. We verified the Java and C versions using Java PathFinder [27], and inspect [28], respectively. At the time of writing, they were the only model checkers to support enough of the core and thread libraries to be applicable.

4.1 Example programs

We originally tried to obtain multi-threaded programs written in Java and C from a well-known web page hosting benchmarks for various problems, implemented in different programming languages [1]. Unfortunately, the quality of the implementations is not sufficient for a scientific comparison. The different implementations are not translations from one language to another, but completely independent implementations. Their efficiency, due to differences in the algorithm, may vary by orders of magnitudes.

Hash The first example is provided by a source that strives for a faithful translation of a benchmark from Java to C++ [24]. We then translated the C++ version to C, and implemented a concurrent version in C and Java.

The program counts the number of matching strings for numbers in hexadecimal and decimal notation, up to a given value. It uses a hash table to store the strings, and worker threads to compute the string representations of each number. While the concurrent implementation is not faster than the sequential one, due to contention on the lock guarding the global data structure, it is still a useful benchmark for model checking. The program utilizes the typical worker thread architecture with fork/join synchronization, which is also found in mathematical simulations and similar problems.

Queue This example implements a blocking, thread-safe queue that offers atomic insertions and removals of n elements at a time. The queue uses a fixed-size buffer, and obeys the constraints that the removal of n elements requires at least n elements to be present, and that the buffer size may not be exceeded. When these constraints cannot be fulfilled, the queue blocks until the operation can be allowed. The queue uses a circular buffer, which wraps around when necessary.

The C version of the queue is used in an ongoing project about model checking networked software [19]. The algorithm has originally been developed and verified in Java, before it has been translated to C, inspiring this paper.

Alphabet client/server The last benchmark is a client/server program. The alphabet client communicates with the alphabet server using two threads per connection: a producer and a consumer thread. The server expects a string containing a number, terminated by a newline character, and returns the corresponding character of the alphabet [3]. In this case, both the client and the server are multi-threaded, and were model checked in two steps; in each step, one side is run in the model checker, using a cache layer to intercept communication between the model checker and peer processes [4]. For the alphabet server, we used both a correct and a faulty version. The faulty version included a read-write access pattern where the lock is released in between, constituting an atomicity race [2], as confirmed by an assertion failure that checks for this.

4.2 Translation to C

Translation of the Java version to the C version proceeded as described in Section 3. For the hash benchmark, we kept the optimization where the C version allocates a single large buffer to hold all strings [24]. This optimization is not (directly) possible in Java. In the Java version, locking was used implicitly by wrapping the global hash table (of type `java.util.HashMap`) in a synchronized container, using `java.util.Collections.synchronizedMap`. A corresponding lock was used in the C translation.

In the queue example, we split the conditions for fullness/emptiness into separate condition variables, as described in Section 3. There were no special issues when translating the alphabet client/server. However, for the experiments, the execution of the peer processes had to be automated by a script, which checks for the existence of a temporary file generated whenever the C model checker inspect starts a new execution run.

4.3 Verification results

All experiments were run on the latest stable release of Java PathFinder (4.0 r1258) and the C model checker inspect, version 0.3. We analyzed the default properties: the absence of deadlocks; assertion violations; and, for Java, uncaught exceptions. Table 2 shows the results. It lists each application (including parameters), the number of threads used, and the time and number of transitions taken for model checking the Java and C version, respectively.

Being a stateful model checker, Java PathFinder (JPF) can check if transitions lead to a new or previously visited state. In the latter case, the search can be pruned. The ratio of such pruned branches to new states grows for more complex cases. This is indicated as a percentage in Table 2; one should keep in mind that previously visited states may include entire subtrees (with additional redundant states), so the percentage is a lower bound on the potential overhead of a stateless search. The C model checker fares much better on small systems with fewer states, as its lightweight architecture can explore more execution runs in a given time than JPF does. One should note that transitions are not always equivalent in the two versions, owing to differences in the language semantics of Java and C, and in the implementations of the model checker platforms.

Inspect had an internal problem when analyzing the alphabet client. We expect such problems to disappear as the tool becomes more mature. Other than that, the alphabet server case stands out, where inspect was very fast. In the correct version of the alphabet server, there is no data flow between threads. The data flow analysis of inspect recognizes this, allowing inspect to treat these thread executions as atomic, and to skip the analysis of different orders of network messages on separate channels. After the insertion of an atomicity race [2] into

Table 2. Verification results for Java and C versions of the same programs.

Application	# thr.	Java				C	
		Time [s]	Transitions			Time [s]	Trans.
			new	visited	vis./new [%]		
Hash (4 elements)	1	1.36	73	34	46	0.03	91
	2	2.76	1,237	1,500	121	0.76	1,438
	3	128.84	124,946	218,748	175	10.72	18,789
	4	> 1 h				288.02	501,576
Hash (8 elements)	1	1.41	121	58	47	0.04	147
	2	3.56	2,617	3,416	130	119.27	181,332
	3	283.90	381,233	748,583	196	> 1 h	
	4	> 1 h				> 1 h	
Hash (17 elements)	1	1.56	205	100	48	0.07	268
	2	7.25	9,882	13,709	138	> 1 h	
	3	1034.51	1,617,695	3,386,868	209	> 1 h	
Queue (size 5, atomic insert/remove with two elements)	2	1.45	121	72	59	0.06	77
	3	2.15	958	699	72	0.91	1,130
	4	23.25	47,973	81,849	170	54.67	62,952
	5	236.72	494,965	975,576	197	> 1 h	
	6	2622.14	4,982,175	12,304,490	246	> 1 h	
Alphabet Client (3 messages)	3	3.01	1,607	4,226	262	Assertion failure inside inspect model checker	
	4	20.12	21,445	83,402	388		
	5	291.95	275,711	1,423,326	516		
Alphabet Client (4 messages)	3	3.83	2,354	6,032	256		
	4	32.68	35,159	133,556	379		
	5	553.87	501,836	2,533,616	504		
Alphabet Client (5 messages)	3	4.63	3,281	8,234	250		
	4	50.73	53,957	201,122	372		
	5	972.50	843,521	4,182,406	495		
Correct Alphabet Server (3 messages)	3	8.08	589	1,164	197	0.14	33
	4	21.29	12,635	36,776	291	0.15	42
	5	124.75	89,590	351,517	392	0.19	51
Correct Alphabet Server (4 messages)	3	8.61	959	1,903	198	0.14	36
	4	30.48	22,560	65,617	290	0.15	46
	5	253.93	179,197	704,855	393	0.19	61
Correct Alphabet Server (5 messages)	3	9.23	1,455	2,894	198	0.14	39
	4	44.55	37,327	108,466	290	0.17	50
	5	391.17	326,862	1,287,935	394	0.21	61
Atomic-race Alphabet Server (3 messages)	3	7.45	141	225	159	1.83	2,633
	4	9.63	146	333	228	43.64	76,502
	5	11.79	158	457	289	2905.33	3,565,667
Atomic-race Alphabet Server (4 messages)	3	7.60	183	304	166	1.79	2,747
	4	9.82	190	453	238	44.43	79,213
	5	12.04	204	619	303	2542.21	3,667,525
Atomic-race Alphabet Server (5 messages)	3	7.76	231	395	170	1.86	2,861
	4	10.04	240	591	246	45.20	81,924
	5	12.26	256	805	314	2541.16	3,769,383

the alphabet server, transitions inside a thread are broken up, resulting in an explosion of the state space. JPF has an advantage in that case, because the caching of network input/output [4] enables the model checker to generate most interleavings of network messages in memory, as opposed to having to execute the peer process many times (up to 113,400 times for five messages).

5 Conclusions

Nowadays, embedded systems may be developed either in Java or C. Java offers easier development, but a translation to C may be necessary if system constraints require it. We show that a development approach where a concurrent core algorithm is developed in Java and then translated to C. Concurrency primitives in Java can be readily mapped to POSIX threads in C. A direct, automatic translation from Java to C is theoretically possible, but a manual translation may yield a more efficient program. Areas where the C code can be optimized include memory allocation and a more fine-grained treatment of condition variables.

Because concurrent software is difficult to verify, we believe that software model checking is an invaluable tool to analyze multi-threaded code. Software model checkers for Java are currently more flexible and powerful than for C. Because of this, it can be beneficial to develop a concurrent algorithm in Java first. Our case studies confirm the viability of the approach.

Acknowlegdements

We would like to thank the research team developing *inspect* for their feedback and advice on using their tool.

This work was supported by a *kakenhi* grant (2030006) from JSPS.

References

1. The computer language benchmarks game, 2010. `http://shootout.alioth.debian.org/`.
2. C. Artho, A. Biere, and K. Havelund. Using block-local atomicity to detect stale-value concurrency errors. In *Proc. ATVA 2004*, volume 3299 of *LNCS*, pages 150–164, Taipei, Taiwan, 2004. Springer.
3. C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS EUROPE 2008*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.
4. C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto. Cache-based model checking of networked applications: From linear to branching time. In *Proc. ASE 2009*, pages 447–458, Auckland, New Zealand, 2009. IEEE Computer Society.
5. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient dynamic analysis for Java. In *Proc. CAV 2004*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.

6. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstractions for model checking C programs. In *Proc. TACAS 2001*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.
7. G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on Martian rover software. *Formal Methods in System Design*, 25(2):167–198, 2004.
8. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. on Software Eng.*, 30(6):388–402, 2004.
9. C. Clark. C hash table, 2005. `http://www.cl.cam.ac.uk/~cwc22/hashtable/`.
10. E. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT Press, 1999.
11. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS 2005*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.
12. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. ICSE 2000*, pages 439–448, Limerick, Ireland, 2000. ACM Press.
13. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. POPL 1997*, pages 174–186, Paris, France, 1997. ACM Press.
14. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley, 2005.
15. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
16. Y. Kamijima and E. Sumii. Safe implementation of C pointer arithmetics by translation to Java. *Computer Software*, 26(1):139–154, 2009.
17. B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, 1988.
18. L. Lamport. How to Make a Multiprocessor that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 9:690–691, 1979.
19. W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto. Introduction of virtualization technology to multi-process model checking. In *Proc. NFM 2009*, pages 106–110, Moffett Field, USA, 2009.
20. T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.
21. B. Meyer. *Eiffel: the language.* Prentice-Hall, Upper Saddle River, USA, 1992.
22. Microsoft Corporation. *Microsoft Visual C# .NET Language Reference.* Microsoft Press, Redmond, USA, 2002.
23. B. Nichols, D. Buttlar, and J. Farrell. *Pthreads Programming.* O'Reilly, 1996.
24. W^3 Systems Design. C++ vs Java, 2009. `http://www.w3sys.com/pages.meta/benchmarks.html`.
25. B. Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1997.
26. A. Tanenbaum. *Modern operating systems.* Prentice-Hall, 1992.
27. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
28. C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *Proc. ATVA 2008*, volume 5311 of *LNCS*, pages 126–140. Springer, 2008.