Java Model Checking

David Y.W. Park   Ulrich Stern   Jens U. Skakkebæk*
David L. Dill

Computer Science Department
Stanford University
Stanford, CA 94305

{parkit,uli,jus,dill}@cs.stanford.edu

Abstract

This paper presents initial results in model checking multi-threaded Java programs. Java programs are translated into the SAL (Symbolic Analysis Laboratory) intermediate language, which supports dynamic constructs such as object instantiations and thread call stacks. The SAL model checker then exhaustively checks the program description for deadlocks and assertion failures. Basic model checking optimizations that help curb the state explosion problem have been implemented. To deal with large Java programs in practice, however, supplementary program analysis tools must work in conjunction with the model checker to make verification manageable. The SAL language framework provides a good starting point to interface new and existing analysis methods with the model checker.

1 Introduction

The Java programming language is becoming increasingly popular for writing multi-threaded applications. In particular, many Internet servers are written in Java. Since Java has multi-threading built in among other advantages, we expect it to gain popularity in other areas such as embedded systems where multi-threading is useful.

Developing multi-threaded programs is notoriously difficult, however. Subtle program errors can result from unforeseen interactions among multiple threads. In addition, these errors can be very hard to reproduce since they often depend on the non-deterministic behavior of the scheduler and the environment.

It is thus desirable to provide tools for software developers that automatically detect errors due to multi-threading. The tools should generate detailed error traces to help the developer during the debugging phase. We have developed such a tool based on *model checking*. A model checker explores all reachable states of a system model, checking whether they satisfy the user-provided correctness specification including the absence of deadlocks and assertion failures. Our tool verifies models described in the SAL (Symbolic Analysis Laboratory) intermediate representation [10], that Java, among other languages, can be translated down to.

Our research focus was in developing a framework that is tailored towards software verification. The SAL model checker supports dynamically changing data structures, which are used, for example, to model Java object creation and call stacks. Popular traditional model checkers like SPIN [12] or Murφ [5] support only constant-size data structures, although an extension of SPIN called dSPIN [4] has been recently developed to support dynamic data structures. Nonetheless, the SAL model checker and the broader SAL language framework was initially designed with software verification in mind, with the intention of providing a system that can deal efficiently with the dynamic aspects of software.

The main challenge in model checking is the *state explosion problem* – the number of states in the model is frequently so large that model checkers exceed the available memory and/or the available time. We have incorporated into our tool two tech-

*Jens Skakkebæk is now at Adomo Inc., Cupertino, CA.

niques to combat state explosion, a form of partial order reduction and hash compaction, and are in the process of implementing additional optimizations.

We also plan to integrate the SAL model checker with program analysis tools that prepare large programs for efficient verification. Our model checker, for instance, could serve as a back-end to the Bandera [2] framework, giving us direct access to Bandera's slicing and abstraction tools that may turn intractable verification models into tractable ones. To allow for easy integration with Bandera, we chose to use McGill's Jimple [17], a three-address representation of Java byte-code on which the Bandera tools operate, as the input language of our model checker.

Related works include the model checkers Java PathFinder [11] and JCAT [3], both of which translate Java into SPIN's input language PROMELA. Since SPIN does not support dynamic data structures, they have to allocate fixed-size heaps and stacks. In addition, both tools translate directly from Java source code rather than from byte-code. Hence, they cannot be easily integrated into Bandera nor verify programs where only the byte-code is available. Moreover, the translation process is more complicated since several advanced Java features like exceptions have a simpler byte-code representation than a source-code one. The Java PathFinder group at NASA is in the process of addressing many of these issues with a new version of their model checker.

Recently, there has been increasing interest in verification tools that rely on the execution of actual code, eliminating the need to represent program states and statements using a specialized description language. VeriSoft [9], for example, can detect errors in C-style concurrent programs by monitoring program execution and systematically directing the scheduler. More recently, Bruening has integrated a deterministic tester into the Rivet Virtual Machine at MIT, that can detect deadlock and assertion failures in Java programs [1]. The tool relies on checkpointing system states in the virtual machine to backtrack to previous states during testing, but like VeriSoft, does not store states that it has already visited. Hence, the disadvantage of these tools is that the same state may be visited multiple times. Consequently, large portions of the state space may be explored redundantly, and non-terminating programs (e.g., server-side processes that loop indefinitely) become problematic. The SAL model checker borrows ideas from VeriSoft-like tools, but couples them with model checking techniques to efficiently explore the state space. Furthermore, our model checker is written in C++ with run-time efficiency in mind.

After giving a high-level overview of the model checker in Section 2, details of the model checker and the methods to increase the size of the models that it can handle are explained in Section 3. Section 4 describes the translation steps from Jimple to an executable model checker. We give results on several Java sample programs in Section 5, and conclude in Section 6.

2 Overview of the Model Checker

The SAL model checker explores all reachable states of a given Java program. The two sources of nondeterminism during this state exploration are the choice of the next thread to run (scheduling) and the input values from the environment. The model checker currently detects deadlocks and assertion violations.

The model checker executable is generated in a sequence of translation steps that includes converting Jimple into the SAL intermediate language [10]. Using an intermediate representation has two advantages. First, languages other than Java can be translated into the intermediate representation, reducing the effort to develop a model checker for each new language. Second, other analysis tools that accept SAL as their input language can be readily used to analyze Java.

SAL is a language for describing transition systems. The transition system is described with a set of guarded commands, each of which consists of a boolean condition on the current state and an associated action that changes the current state into the next state. SAL has two slightly different forms – Level 1 and Level 0. While SAL Level 1 has explicit guarded commands, SAL Level 0 folds the guarded commands into one large transition function.

The SAL language has several features that make it a good target for software model checkers. First, SAL provides unbounded arrays whose sizes vary dynamically. These arrays are used, for example, to hold dynamically created Java objects and threads. (A Java thread is an object of class `java.lang.Thread`.) Second, SAL has abstract data types that can be used as unions. A stack frame, for example, is modeled as a record containing, among other fields, a union that can hold values for each possible method return type.

The model checker executable is generated in the following steps as depicted in Figure 1.

- **Java** and **Java byte-code**. Either language can be used as input language to our model checker.
- **Jimple**. Generated by Java byte-code to Jimple translator, which we extracted from the Bandera framework.
- **SAL Level 1**. Generated by our Jimple to SAL translator.
- **SAL Level 0**. Generated by a trivial translator from SAL Level 1.
- **C++**. Generated from SAL Level 0 by our SAL to C++ translator. An executable of the model checker is obtained by compiling this C++ file together with the C++ model checking core.

3 Model Checking SAL

The SAL model checker does an exhaustive search of the state space by either a depth first or breadth first traversal. Starting from the initial state, it considers the set of rules whose guarding conditions are enabled in the state. For each enabled rule, the checker generates a successor state by executing the associated action. The resulting state is stored in a hash table so that it is not re-expanded if it is encountered again. A brute force approach would generate a new successor state for each enabled rule that is fired, where a rule would roughly correspond to one program statement. However, this results in many unnecessary interleavings between statements that are independent of one another.

As an initial step to combat state explosion, we adopt the strategy of executing a sequence of rules local to a thread atomically and interleaving threads only when a global operation is performed. In particular, the SAL model checker uses a technique called *atomic blocks* that Bruening developed for the Rivet Deterministic Tester [1]. Secondly, we use hash compaction to contain the otherwise unmanageable memory usage necessitated by large state spaces and state vectors. These model checking techniques are described in the rest of this section.

3.1 Atomic Blocks

The SAL model checker uses a form of partial order reduction called *atomic blocks* [1], that is driven by synchronization constructs and for which static analysis is not required. The main idea is to execute one thread as long as possible before generating a new state, after which other threads may be scheduled. This is safe provided the operations of the other threads can not possibly interfere with the operations of the current thread being executed. The algorithm differs from the usual VeriSoft-like approach where a global state is defined by the execution of a "visible" operation, i.e. an operation on a shared variable. Because threads in Java share the same address space, most variable accesses are potentially visible operations and the interleaving of threads may become too fine grained.

The atomic block method assumes that accesses to shared variables are always protected by locks. Although our tool does not currently enforce this, it will be extended with techniques used in data race detection tools for multi-threaded programs. Race detection methods based on the *happens-before* relation check that conflicting memory accesses from different threads are ordered by synchronization events [13] [6]. Another example is Eraser [15], a dynamic race detection tool developed at DEC. Eraser keeps track of what locks each thread owns when it accesses a shared variable at run time. When the set of locks one thread holds is disjoint from the set of locks another thread owns when they access a common variable, Eraser issues a warning since the variable should have been protected by a common lock. Finally, static type-based analysis has also been successful in detecting race conditions in large Java programs [7].

Having stated the assumptions, a brief summary of Bruening's atomic block algorithm is given here. At a state s, we generate its successors by taking each enabled thread in turn and executing it until

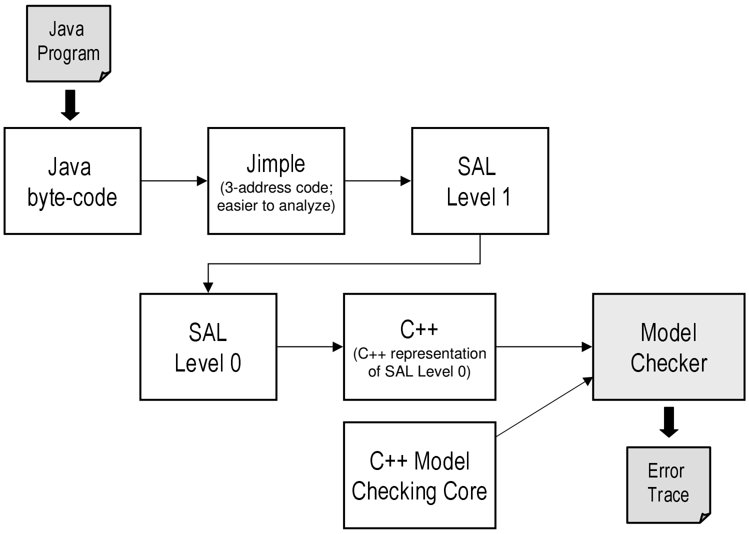

Figure 1: Translation steps in the model checker

an unlock operation is performed or the thread dies. Ignoring nested locking for the moment, note that in Figure 2(a) the sequence of statements (rules) between *Unlock(A)* and *Lock(B)* must only modify variables local to the thread since we assume that accesses to shared variables require the acquisition of locks. Hence, it is unnecessary to interleave such statements with statements from other threads.

Similarly, the statements within the synchronized region delimited by *Lock(B)* and *Unlock(B)* can be executed atomically since no other thread can access B while the lock is held. Furthermore, region X and region Y can actually be executed in the same atomic block because any operation by any other thread that may execute between the two regions must be independent of region X. We only have to consider schedules where such operations occur before region X. Therefore, we can safely execute each thread until an unlock is performed, at which point we return the new state.

In the case of nested synchronization blocks, the atomic block algorithm will span multiple locking operations. In Figure 2(b), for instance, the atomic block for T1 will begin with *Lock(A)* and end with the first unlock, namely, *Unlock(B)*. This may seem as though we are neglecting certain schedules, e.g., the case in which thread T2 modifies B while T1 has locked A but not B. Yet, such schedules *can* be ignored since we will consider the schedule in which T2 modifies B before T1 locks A; in terms of assertion checking, it does not matter whether T2 accesses B before or after T1 accesses A since A and B are independent.

3.2 Deadlock Detection

In general, a deadlock will be recognized by the model checker when a state has no successors and it is not a valid termination state. The only problem that atomic blocks pose is that certain deadlocks can be missed. In the case illustrated in Figure 2(b), the particular schedule required to realize the deadlock is never executed. As noted before, the atomic block algorithm will never execute the schedule in which T2 locks B immediately after T1 locks A but before T1 locks B. The lock-cycle deadlock that results from this schedule would have been caught if atomic blocks were delineated by locking as well as unlocking.

However, since larger atomic blocks result in fewer unnecessary interleavings, and in turn fewer states, we adopt Bruening's approach in delineat-

```
synchronized(A) {    |   Lock(A)                T1:             T2:
...                  |   ...                    =========       =========
}                    |   Unlock(A)              Lock(A)         Lock(B)
                     |                          ...             ...
...                  |   region X...            Lock(B)         Lock(A)
                     |                          ...             ...
synchronized(B) {    |   Lock(B)                Unlock(B)       Unlock(A)
...                  |    region Y...           ...             ...
}                    |   Unlock(B)              Unlock(A)       Unlock(B)

                    (a)                                  (b)
```

Figure 2: (a) Synchronized regions and atomic blocks. (b) An example of nested locking.

ing atomic blocks by unlocks only and detecting lock-cycle deadlocks by analyzing states for the existence of a cycle in the hold-wait relationship between threads. By keeping track of the most recently released lock for each thread, we can look for cycles where T1's most recently released lock is owned by T2, whose most recently released lock is owned by T3, and so on until we get to some thread Tn whose most recently released lock is owned by T1.

This check is linear on the number of threads, and is done only when a thread fails to obtain a lock (i.e. potentially closes a hold-wait chain into a cycle). Hence, the deadlock check introduces only a marginal overhead in most cases. Furthermore, a particular lock-cycle deadlock will always be detected (unless other deadlocks are detected first) since all schedules of atomic blocks are considered and the atomic blocks stop at each unlock, effectively considering all the locks each thread has accessed.

3.3 Implementation Specific Optimizations

We have also implemented simple optimizations that turn out to be critical in saving time and memory in model checking with atomic blocks. First, if an atomic block consists of more than one rule, a new state is instantiated once and each subsequent rule in the atomic block updates values in the same instantiation. This is permissible because intermediate states within the atomic block do not have to be stored during model checking. Since state vectors modeling dynamic structures like the thread stack and heap can get very large, creating a copy of the current state after each rule turned out to be a bottleneck in execution time. In some cases, the optimization led to an order of magnitude improvement in execution time.

Another source of run time overhead was the linear traversal over the set of rules required in determining which rules are enabled at each new state. Since most of the rules map to individual statements in a sequential process, the value of the program counter in the current state is used to locate in constant time the next enabled rule hashed on the PC value.

Finally, atomic blocks are aborted when a thread fails to acquire a lock and a successor state is not generated. This significantly reduces the number of schedules that need to be considered, reducing the size of the explored state space while preserving deadlocks and assertion failures. The informal justification is as follows. Suppose we have a thread T1 holding lock A and a thread T2 that blocks on the lock. The schedule in which T2 blocks on lock A does not have to be considered since the model checker will execute some schedule in which T1 first releases lock A allowing T2 to obtain the lock. Before T1 releases lock A, moreover, T1 can not access the same shared variables that T2 accesses before T2 tries to lock A since this will lead to a lock-cycle deadlock which our model checker will detect. (Note that locks can not be unlocked in the middle of an atomic block so T2 will be holding all of the locks on variables it accesses before blocking on lock A.)

Therefore, even when T1 is executed first, releasing lock A, the variables that T2 accesses before trying to lock A will not have been modified by T1 and it would effectively be as if T1 was signalled and allowed to continue. In addition, aborting atomic blocks when a thread blocks on a lock can not cause any deadlocks to go undetected since our lock-cycle deadlock checking functionality detects all deadlocks as long as atomic blocks end at each unlock and thread termination.

3.4 Hash Compaction

To reduce the memory requirements of the model checker, we have implemented hash compaction [18, 16]. Hash compaction reduces the memory requirements of the state table, which stores all states reached during verification and is used to decide whether a newly reached state is new or has been visited previously. Instead of storing the full state descriptor in this table, hash compaction stores only a (hash) signature. The memory savings come at the price of a certain probability that the verifier incorrectly claims that an erroneous protocol is correct. This probability, however, becomes negligibly small when choosing the signature size appropriately. Typical signature sizes are between 16 and 45 bits, resulting in memory savings of often more than two orders of magnitude.

4 Translation Steps

4.1 Translating Jimple to SAL

The SAL language is used to model the Java (Jimple) program as a state transition system. SAL has state variables, nondeterministic inputs, an initialization function, and a transition function that is a collection of guarded commands (or rules).

The state of the Java program is modeled in SAL using the following state variables: (1) each thread contains a program counter (`PC`), stack (array of frames), and stack pointer (`CurrentFrame`), etc.; (2) each object contains a class id, the fields, a counter for locking, etc. To select the next thread to execute, we use a nondeterministic input (`TID`).

Each Jimple statement is translated into a guarded command. For example, the Jimple statement

```
i0 = 1
```

is translated into

```
(PC[TID] = label_0) -->
  next(Stack)[TID]
    [CurrentFrame].localVariables.i0 = 1;
  next(PC)[TID] = label_1;
```

where `(PC[TID] = label_0)` is the guard condition, followed by the SAL statements that can be executed if the condition is true.

The translator has to deal with all advanced features of Java like inheritance, overriding, overloading, dynamic method lookup, exception handling, etc. For detailed descriptions of the semantics of these features, refer to the Java Virtual Machine Specification [14]. We believe that most of these features are easier to deal with at the Jimple level than at the Java source code level.

Exception handling, for example, is implemented by four SAL rules and several SAL tables that are generated at compile time. The first rule is passed the location label of the statement that throws the exception, and uses the `method_table` to look up the identifier of the method in which the exception is thrown. The second rule searches the `exception_table` for this method for a `catch` clause that handles the exception. A `catch` clause handles an exception only if the class of its parameter is the class of the exception or a superclass of the class of the exception. The `subclass_table` is used for this subclass relationship test. The third and fourth rules deal with the case that during the search no `catch` clause is found that handles the exception. Typically, the third rule is executed, returning from the method and re-throwing the exception. If the first stack frame of the thread had been reached, however, a Java run-time error occurred and this is signalled to the user.

4.2 Translating SAL to C++

The SAL description of the program is translated into a C++ source file that is #included in the model checking code, and the result is then compiled into an executable that outputs a trace if any deadlock or assertion failure exists. The included source file contains:

1. A C++ class with various accessors for each distinct type in the SAL description. Unbounded arrays are used to model data structures such as the stack and heap, and are important in storing per-thread data given that the actual number of threads can not be determined statically. Unbounded arrays are translated into dynamically resizing vectors where a designated default value represents the infinite sequence of array elements to the right of the last non-default element in the array. When an index n greater than the current maximum index k is accessed, the vector is resized to size $n+1$, letting elements at index $k+1$ through $n-1$ be default values. Likewise, when the last non-default element in the vector is set back to a default value, the vector is resized to size $k+1$ where k is the index position of the next right-most non-default element.

2. Functions simulating SAL guarded commands. The *guard* function takes in the current state and the current values for the non-deterministic inputs and returns the next enabled rule to execute. The *apply* function then executes the enabled rule and any subsequent rules that fall under the same atomic block, and returns the resulting successor state to the model checking routines. Each successor state is generated by systematically incrementing the non-deterministic input values which includes the thread ID to schedule next.

5 Results

Table 1 gives the examples on which we tried the model checker. The source code for the examples is available at

`http://verify.stanford.edu/uli/java/`

The results we obtained for the examples are given in Table 2. For each example, we ran the checker under three different modes:

1. Noatomic: Atomic blocks turned off, interleaving of threads fined grained as possible at the level of individual rules.

2. LockUnlock: Atomic blocks delineated not only by unlocking but by locking as well. Does not require special lock-cycle deadlock detection method described in Section 3.

3. Unlock: Usual atomic blocks delineated by unlocking only.

Note that we achieve larger state reductions for the larger examples. In some cases, the reduction achieved by atomic blocks is three orders of magnitude. There is, however, many areas still in the implementation of the model checker where the execution time can be optimized.

Before hash compaction, moreover, the KSU Pipe example could only enumerate 140,000 states before running out of memory in the *noatomic* mode. With hash compaction, both the KSU Pipe and the ReaderWriter examples were able to be completed in the *noatomic* mode.

6 Conclusion

Tools to verify software have become almost a necessity as both the complexity of software and the extent of its use in critical systems are continually increasing. Given the usually intractable state spaces of software programs, different kinds of abstractions, optimizations, and "tricks" must be employed in concert to tackle the verification.

The SAL framework provides the means to easily integrate new and existing verification techniques around a common intermediate representation. This paper gives preliminary results on applying this framework to model checking Java programs. Support for dynamic data structures in SAL has made translating Java down to SAL direct and straightforward. Atomic block reduction and hash compaction demonstrate promising results in dealing with large state spaces and state vectors. We plan to extend the SAL model checker with additional state space reduction techniques, including symmetry reduction and a partial order reduction method called *persistent sets* [8] that will eliminate unnecessary interleavings even at the atomic block level. Yet ultimately model checking optimizations alone will not be sufficient to curb the state explosion problem. The model checker must be supplemented with front-end slicing and abstraction tools as well as other static program analysis tools. The SAL framework has provided a good starting point

Table 1: Example programs

example	description	primitives	# prim.
Bruening's SplitSync	two threads access shared variable	synchronized	2
CS193k ReaderWriter	two reader and two writer threads	synchronized, wait/notify	4
CS193k TurnDemo	two threads synchronize using a semaphore	synchronized, wait/notify	6
NASA's classic	two threads communicate using events (deadlocks)	synchronized, wait/notify	4
NASA's ksu_pipe	2-stage pipeline	synchronized, wait/notify	4

Table 2: Results on example programs

example	algorithm	states	rules	time	state reduction
Bruening's SplitSync	noatomic	1763	4090		1.0
	lockunlock	57	77		53.1
	unlock	37	43		95.1
CS193k ReaderWriter	noatomic	261 838	1030 130	442s	1.0
	lockunlock	848	2184	1.91s	309
	unlock	528	1356	1.54s	496
CS193k TurnDemo	noatomic	26 145	68 715	30.4s	1.0
	lockunlock	385	617	2.57s	67.9
	unlock	166	236	2.40s	158
NASA's classic	noatomic	45 924	118 047	46.6s	1.0
	lockunlock	322	554	2.38s	143
	unlock	143	234	2.11s	321
NASA's ksu_pipe	noatomic	3990 883	14 022 723	6401s	1.0
	lockunlock	28 357	92 334	51.5s	141
	unlock	4991	15 762	11.8s	800

to research the ways that different analysis tools interface and interact with each other.

References

[1] D. L. Bruening. Systematic testing of multi-threaded Java programs. Master's thesis, MIT, 1999.

[2] J. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, 2000. To appear.

[3] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.

[4] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *6th International SPIN Workshop on Practical Aspects of Model Checking*, 1999.

[5] D. L. Dill. The Murφ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–3, 1996.

[6] A. Dinning and E. Schonberg. Detected access anomalies in programs with critical sections. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, 1991.

[7] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *ACM Conference on Programming Language Design and Implementation*, 2000. To appear.

[8] P. Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem. In *Lecture Notes in Computer Science*, volume 1032, 1996.

[9] P. Godefroid. Model checking for programming languages using VeriSoft. In *24th ACM Symposium on Principles of Programming Languages*, pages 174–86, 1997.

[10] The SAL Group. The SAL Intermediate Language. Technical report, UC Berkeley, SRI, Stanford University, 1999.

[11] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*. To appear.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–94, 1997.

[13] L. Lamport. Time, clock, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, 2nd edition, 1999.

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

[16] U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration*. PhD thesis, Technical University of Munich, 1997.

[17] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, 1998.

[18] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification. 5th International Conference*, pages 59–70, 1993.