SRC Technical Note

2001-004

December 2001

Selected 2001 SRC Summer Intern Reports

Compiled by

Cathy Miller, M.L.I.S.

Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
http://www.research.compaq.com/SRC/

This document features informal reports by interns who spent the summer of 2001 working with researchers at Compaq Systems Research Center (SRC). The interns were graduate students in computer science or electrial engineering Ph.D. programs. Each worked for about three months at SRC, collaborating on a project with members of the research staff. The primary goal of this technical note is to describe the summer research projects. The interns were encouraged to write their reports in whatever format or style they preferred, so that non-technical observations (such as background and impressions arising from their stay) could also be included.

Typsy: a type-based search tool for java programmers

Christie Bolton
University of Oxford

1. Introduction

This summer I spent a very enjoyable three and a half months at SRC. Greg Nelson, my host, and I built Typsy, a type-based tool for java programmers.

Java programmers using unfamiliar packages often need to spend a significant amount of time searching for methods or procedures to perform routine tasks. This search time is spent reading the documentation, thumbing through indexes and grepping through interface files. Our goal for the summer was to build a tool to eliminate or greatly reduce this search time. The idea is that even if the programmer doesn't know the name of the method they probably know something about its argument types and result type. This suggests a search tool that accepts both the signature of the method and a context specifying the relevant libraries. It then returns a list of combinations of methods, fields and constructors with the correct signature.

2. Example

An example of when the tool might be used arose during its construction. We wished to extract the first word from a stream of input. Had we already built the tool our query would have been as follows: our goal type is a string; our given argument is an input stream that we shall call ``in''; and our context is the *java.io* package. The tool comes up with a list of type-correct suggestions each with hyper-text links to the java documentation. These suggestions use combinations of methods, fields and constructors. The fifth suggestion, the procedure that we were looking for, proposes that we construct a *StreamTokenizer* with our given *InputStream*, *in*, as its argument, and then call the *sval* field on the StreamTokenizer.

(new java.io.StreamTokenizer(java.io.InputStream in)).sval

3. Outline of project

Our initial step was a feasibility study: could we build a tool that produced a *single* smallest term of the correct type? We were able to use an in-house package, built as part of ESC/Java, to extract the methods, fields, constructors and supertypes of classes from their java source files. We defined a cost function as a measure of the size of each term and then applied an algorithm based on Knuth's generalization of Dijkstra's shortest path algorithm. Our tool then identified a minimal-cost term of the correct type. Having completed our feasibility study and established that it was possible to extract a single minimal cost term of our desired goal type, the next step was to extend this to finding, say twenty smallest cost terms. We decided to adopt a breadth-first search approach.

For each query, the tool constructs a tree with the desired goal type as its root node and with each layer containing any types that might be directly used in the construction of a type in the layer above. The tree is *pruned* so that the bottom layer contains only given arguments and every other layer contains only those types that can be constructed from the types remaining in the layer below. Various dynamic adjustment heuristics are applied to ensure an appropriate number of suggestions. If there are too few suggestions then a few basic types such as *boolean* and *integer* may be included in suggested procedures. If there are too many suggestions we *snip* and *flag*: we snip the tree at any nodes with too many children and flag the fact that we have done this in our list of suggestions. It is then up to the user to investigate this branch further if they believe that that is where their solution lies.

We gave Typsy an html interface with hyper-links from the methods, fields and constructors included in suggested procedures to Sun's java documentation. We used perl cgi-script to provide the link between the interface and the tool.

A Fast Incremental Update Scheme for PageRank

Steve Chien
University of California at Berkeley

1 Introduction

We began with the broad goal of exploiting changes in the link structure of the web. For our raw data we used the results of two large crawls of the web made at SRC from May and November 2000, and their compact representation in the connectivity server, also built at SRC. While we made a number of observations on the evolution of link structure over time, our most substantial result was an increased understanding of Google's very popular PageRank measure of web page importance and an application of this to a very fast incremental update algorithm for PageRank.

2 PageRank

PageRank is Google's highly successful method for evaluating the importance of web pages and is based on interpreting the web as a very large graph in which pages are vertices and links are directed edges. It is then natural to think of a Markov chain, or a random walk on this graph; from our current page, we choose a random edge to follow. We then modify this random walk so that at any step there is a certain probability that we jump to a random page. PageRank is then the stationary probability distribution of this Markov chain. While PageRank is therefore only an eigenvector computation, the size of the web graph limits us to using power iteration, a relatively slow approach.

PageRank works very well in practice, and some effort has been spent analyzing its effectiveness. In particular, Andrew Ng, Alice Zheng, and Michael Jordan reported that one nice feature of PageRank is its stability under significant changes to the web graph and concluded that it is the random jump that is responsible for this.

3 Our Analysis

We analyzed PageRank from a similar perspective, asking how changes to the links out of a single page affects the PageRank of other pages. Our analysis showed that the random jump probability has the effect of localizing the impact of such updates to a small subgraph surrounding the modified page. This suggested the following update algorithm: We assume we are given the current web graph and the correct PageRank for it. We then discover that a new edge has been added. We respond by constructing a small subgraph around the affected page, and consolidate the remainder of the web graph into a single superpage that simulates the interaction of the rest of the graph with the subgraph. We then compute the new PageRank on the small subgraph, and assume that PageRank has not changed in the superpage. Our analysis shows that this should be an excellent approximation to the updated PageRank.

As an extra result, we showed that PageRank (and a large class of other Markov chains) is also monotonic in the sense that if a page receives a new incoming edge, its PageRank increases.

4 Our Experiments

We implemented our algorithm on the raw data from the May and November 2000 web crawls. In a series of small experiments, we started with the 61 million pages from the May 2000 web crawl and added single edges. We found that our approximations on these changes were extremely accurate. We then tried a much larger scale experiment in which we incrementally incorporated all of the edge changes between May and November, for a total of over 17 million edge additions and deletions. Once again, we showed that our algorithm worked extremely well. [We are preparing to publish a more rigorous and detailed description of our experiments.] In each case, our algorithm performed far fewer computations than full-blown power iteration.

5 Acknowledgements

In all of the above, the first person plural includes primarily my summer advisor Cynthia Dwork, who made the summer an especially rewarding and enjoyable experience. I also greatly appreciate Janet Wiener's indispensable help with the connectivity server, as well as conversations with Li Zhang, Yunhong Zhao, Marc Najork, Mike Burrows, Lyle Ramshaw, and many others.

OS Support for Speculative I/O Prefetching

Keir Fraser
University of Cambridge

1. Introduction

Speculative execution is a technique for prefetching disk blocks before they are required, by pre-executing application code whenever the CPU is idle. Disk requests found by speculative execution are turned into prefetches. The original SpecHint tool, written by Fay Chang, was implemented entirely within user space. Although it was effective for many benchmarks, it is hard to control resource utilisation without assistance from the operating system. This means that SpecHint can harm application performance if, for example, memory is scarce.

My work this summer explored the potential advantages of removing this constraint by implementing and evaluating a design that includes special operating system support for the speculative execution approach. Somewhat to our surprise, our results indicate that allowing such support will not produce larger performance improvements. Nevertheless, where adding such support is feasible, our new design has improved resource control which makes it a more practical alternative to SpecHint.

2. Design overview

Each time a new process is created, an additional process is also forked and marked as the *shadow* of the primary process. The shadow exists for the entire lifetime of its parent, and remains idle until the primary process requests a file region which must be fetched from disk. At that point the shadow is synchronized by copying the primary process's memory and file tables, and becomes runnable.

We take particular care when allocating shared resources to a shadow process to ensure that this does not impact the performance of higher-priority operations. Critical resources which we consider are processing time, memory, and disk bandwidth.

The most difficult resource to control is memory, because it is generally impossible to allocate a memory page to a shadow process without evicting pages which may be more valuable. It is common practice to allocate most of physical memory to either file cache or application virtual memory. Only a small number of pages will typically be available for immediate allocation, and a kernel daemon will periodically run to keep the free list `topped up'. A page allocated to a shadow process may therefore cause another page to be evicted at some later time, when the reclaim daemon runs, but it is impossible to determine that page's value, or even which page it will be. We implement a simple scheme in which shadow pages are initially allocated onto a low-priority page list, thus making them good candidates for eviction, and by preventing shadow processes from marking pages as referenced. Our intention is that high memory demands by a shadow process will cause its own pages, or pages from another shadow process, to be evicted before those of a primary process.

3. Results

We implemented our design within Linux 2.4.8, and evaluated it using a range of disk-intensive applications. These included

- *Gnuld*: object code linker
- *Agrep*: text-file pattern matching
- *XDataSlice*: three-dimensional data visualization
- *PostgresQL*: flexible DBMS, based on POSTGRES
- *Sphinx*: speech-recognition system

Our experiments were conducted on an 866MHz Pentium III with 128MB of memory, running our modified kernel. The test filesystem system consisted of four Compaq RZ1CB Ultra SCSI discs (12ms average seek time) striped into a 16GB array. The maximum transfer rate supported by the discs and the SCSI interface was 40MB/s.

With 128MB of memory, which was more than adequate for the benchmarks we used, the performance improvements were similar to those achieved by the original SpecHint tool.

When memory was reduced to 64MB, we were able to prevent speculative execution from significantly reducing performance on those benchmarks which required a large amount of memory (for example, Sphinx).

Porting the FastVM from Alpha to IA-32

Matthias Jacob
Princeton University

1. Introduction

The goal of the project was to port the existing Java Fast Virtual Machine (FastVM) on the Alpha platform to IA-32. This includes the whole runtime environment as well as the just-in-time compiler. The work during the summer was mainly focused on the just-in-time compiler. The important parts of the project were to bridge the differences between the Alpha and IA-32 architecture and especially to design a fast and efficient register allocator on the IA-32 platform.

2. Alpha vs IA-32

Apart from the general difference of processing 32 bit instead of 64 bit the Intel architecture complicates the implementation of the just-in-time compiler in various aspects:

- no fixed instruction length
- few multi-purpose registers
- floating-point stack instead of registers
- registers bound to certain instructions

This leads to design issues such as having a higher register pressure on the IA-32 architecture and implementing 64-bit operations using multiple 32-bit instructions and allocating additional registers.

2. Stack Frames

Compared to the standard GNU stack frame specification, several things had to be changed for the Java just-in-time compiler. For example, since some values are represented as 64-bit values within Java the stack frame needs to be aligned to 8 byte boundaries instead of 4 byte boundaries. Furthermore, a stack check needs to be introduced in order to be able to do appropriate exception handling. Also, since it was the goal to make as many registers as possible available for general purpose a reserved stack slot is introduced to distinguish the parent from the current stack frame which makes the frame pointer redundant.

3. Floating-Point operations

The floating-point unit in IA-32 uses a floating-point stack instead of directly accessible registers in order to store floating-point values. Floating-point operations can be executed either on the first two elements of the stack or on an element in memory and on top of the stack, whereas the result is stored on top of the stack. There are no transfers possible between the processor registers and the floating-point stack which can become a performance problem if the register allocator doesn't take it into account.

4. Register Allocation

The design goal of the register allocator is to minimize the compile time, and optimize the generated code as much as possible. In total there are 7 registers available for general purpose values depending on whether the instructions are able to address these registers. In addition the floating-point stack can carry 7 floating-point values as well.
The register allocator for the Alpha version of the FastVM uses a data structure that assigns each entity in the program a home location and maps this to temporary locations in the registers. The register allocation itself works on a local scheme for each Java bytecode instruction and doesn't use any global algorithm since this can become expensive.

5. Results

When comparing the ratio of the SpecINT and SpecJVM benchmarks of different JVM implementations on different platforms, the FastVM is one of the leading implementations. It is the question whether it is possible to preserve the performance of the implementation on a different platform. We compared the first prototype implementation of the FastVM for IA-32 to existing JVM implementations by Sun and IBM and found that on several of the SpecJVM benchmarks the FastVM is doing better by about 10%. However, there are still benchmarks that do a large number of floating-point operations, where performance is decreasing. This has to do with the complicated floating-point handling on IA-32. But after all, this project shows that the FastVM can be ported to a different architecture with a minimum amount of work in a few months.

Verifying TLA+ Invariants with ACL2

Carlos Pacheco
University of Texas at Austin

1. Introduction

Reasoning in TLA [5] consists largely of reasoning about actions. By most accounts, 90% of all reasoning in TLA+ specifications [4] occurs at the action level, where temporal logic has been eliminated. Action reasoning alone, for example, is involved in all but the last step of establishing an invariant of a specification. Consider a system with starting state *Init* and next-state relation *Next*. In order to establish an invariant *Inv* of the system, we need two lemmas:

1. *Init => Inv*
2. *Inv /\ Next => Inv'*

Once we establish these two lemmas, one application of a TLA inference rule, along with simple temporal reasoning, lets us establish the invariant at the temporal level (the formula []*Inv*). In the Disk Synod algorithm [1], establishing []*Inv* from formulas like (1) and (2) above takes up one page, while the creation of an invariant and its verification at the action level spans 18 pages.

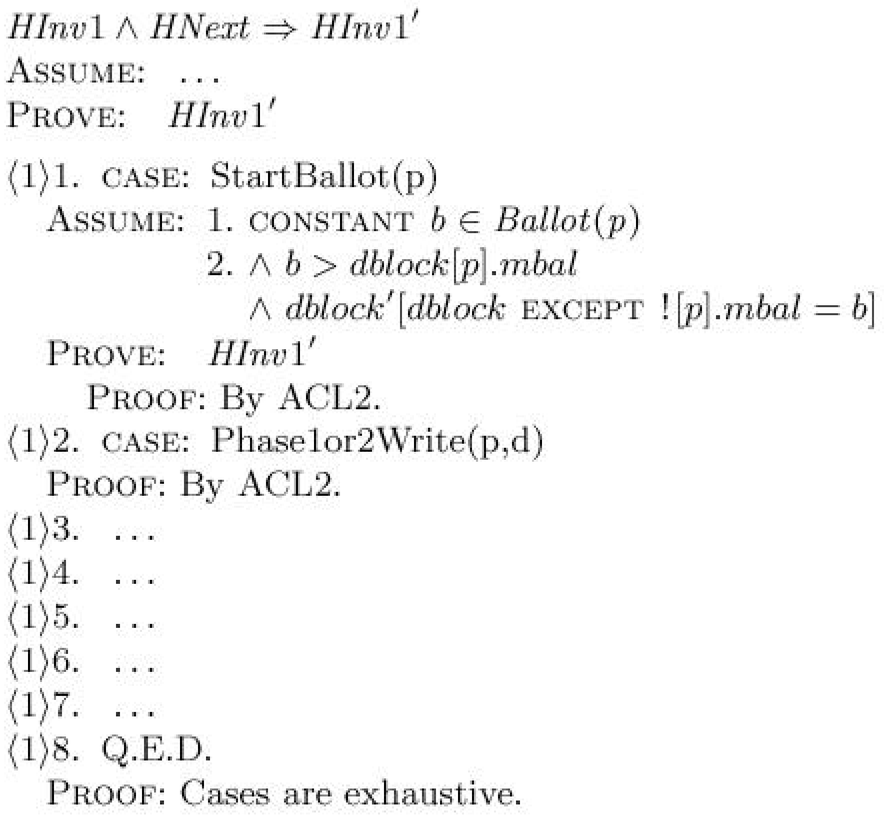
Our goal is to provide mechanical support for proving TLA+ invariants at the action level. A system that deals effectively with action-level formulas would take us a long way in mechanically checking the correctness of specifications.

Our platform of choice for mechanical verification is ACL2 [1]. The ACL2 system is attractive for several reasons. It is among the most automated in the spectrum of theorem provers. It blends arithmetic decision procedures with rewriting techniques. Finally, it is a stable and robust system, designed to tackle industrial-size verification projects.

2. A Mechanical Translator

A first experiment at the University of Texas at Austin [7] consisted in manually translating the Disk Synod algorithm into ACL2 and verifying two invariants of the algorithm. The goal of the summer project was to automate the translation. We wrote a proof-of-concept TLA/ACL2 mechanical translator. The translation scheme is largely based on ACL2's finite set theory [6]. Note the keyword *finite* in the previous sentence: the translator handles a subset of TLA+ concepts, and infinite sets are not allowed.

Our translator not only translates TLA+ specifications, but also *structured proofs* [3] of conjectures about the specifications. In writing a structured proof, we mark some reasoning steps as ``checked by ACL2" and leave others unmarked (Figure 1). We use ACL2 to check only those steps marked as ACL2 steps. The idea is that, short of mechanically verifying every step of a proof, a user might first want to explore pieces of a proof that are not entirely clear or where he lacks confidence. Also, we want to use ACL2 only on steps where it is appropriate to use ACL2: low-level, quantifier-free formulas.



```
$HInv1 \land HNext \Rightarrow HInv1'$
ASSUME:  ...
PROVE:   $HInv1'$
⟨1⟩1. CASE: StartBallot(p)
  ASSUME: 1. CONSTANT $b \in Ballot(p)$
          2. $\land\ b > dblock[p].mbal$
             $\land\ dblock'[dblock$ EXCEPT $![p].mbal = b]$
  PROVE:  $HInv1'$
    PROOF: By ACL2.
⟨1⟩2. CASE: Phase1or2Write(p,d)
  PROOF: By ACL2.
⟨1⟩3.  ...
⟨1⟩4.  ...
⟨1⟩5.  ...
⟨1⟩6.  ...
⟨1⟩7.  ...
⟨1⟩8.  Q.E.D.
  PROOF: Cases are exhaustive.
```

Figure 1. A portion of the proof of invariant HInv1 in the Disk Synod algorithm.

3. Results

Our translator handled the entire TLA+ specification of Disk Synod and two structured proofs of invariants of the algorithm (invariants *I2a* and *I2c* in the Disk Paxos paper [1]). We used ACL2 to check most steps of these invariants. The few steps we avoided were obviously correct high-level steps such as step <1>8 in Figure 1 above, whose correctness follows by the fact that steps <1>1 through <1>7 (not shown in detail in Figure 1) establish the invariant for every disjunct of *HNext*, the next-state action.

Our effort led us to discover some needed hypotheses that were absent in the original structured proofs.

4. Conclusion

ACL2 is a general-purpose theorem prover, and one can use it to verify every step of a proof in almost any mathematical domain, from real analysis to circuit design. In our first experiments, we used ACL2 to verify every step in the proofs of *I2a* and *I2c*. More than half our time was spent trying to reason about simple steps in higher-level concepts like quantification. The main drawback in using ACL2 is the different levels of abstraction at which TLA+ and ACL2 users commonly operate. ACL2 theories are usually fairly low-level, concrete and computational. On the other hand, TLA+ specifications tend to be more descriptive than constructive, and make liberal use of higher-level concepts which are difficult to handle in ACL2's first-order, essentially quantifier-free logic.

Our second approach was to use ACL2 only where it might be suitable--closer to the leaves of a proof, where quantification has been eliminated and all that remains are large but low-level formulas. Although low-level, these formulas are nontrivial and can be a challenge for any theorem prover. Moreover, it is most often in these elaborate steps where errors are uncovered. It is to ACL2's credit that it did so much work with little guidance. At the correct level of abstraction, the prover not only helped us verify statements, but it also pointed the way to omissions and errors with remarkable precision.

In further work, we would continue focusing ACL2's attention on low-level segments of TLA+ proofs, refining our tools and lemma libraries to increase the prover's power in this restricted domain. For the remaining high-level steps of TLA+ proofs, we might recruit a different theorem prover with a logic more expressive than ACL2's. A future proof checker for TLA+ might, in addition to steps labeled ``checked by ACL2," also have steps labeled ``checked by *X*" where *X* is a different theorem prover. The framework for structured proofs we have followed allows for collaboration among multiple provers (each with its own particular strengths) in attacking a verification project.

A. References

[1] Eli Gafni and Leslie Lamport. *Disk Paxos.* in Maurice Herlihy, editor, Distributed Computing: 14th International Conference, DISC 2000 Lecture Notes in Computer Science number 1914, pages 330-344, Springer-Verlag, 2000. [2] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, *Computer-Aided Reasoning: An Approach,* Kluwer Academic Publishers, 2000.

[3] Leslie Lamport. *How to Write a Proof*. American Mathematical Monthly 102, 7 (August-September 1993)} pages 600-608.

[4] Leslie Lamport. *Specifying Concurrent Systems with TLA+.* In M. Broy and R. Steinbruggen, editors, Calculational System Design. IOS Press, Amsterdam, 1999.

[5] Leslie Lamport. *The Temporal Logic of Actions.* ACM Transactions on Programming Languages and Systems, 16(3):872-923, May 1994.

[6] J Moore. *Finite Set Theory in ACL2.* TPHOLS 2001, Edinburgh, September 2001.

[7] Carlos Pacheco. *Reasoning about TLA Actions.* Undergraduate Honors Thesis. Technical Report TR01-16, Department of Computer Sciences, The University of Texas at Austin, May 2001.

Combining Thread-Modular and Procedure-Modular Verification

Sanjit A. Seshia
Carnegie Mellon University

This report describes work done this summer with Shaz Qadeer, my host at SRC.

1. Introduction

Multithreaded software systems are susceptible to a wide range of errors. Many of these errors are synchronization errors arising from data races, i.e., when two threads access or update the same memory location at the same time. To avoid this, programmers place restrictions on when shared locations can be read or written. The most common way of avoiding data races is to associate locks with shared data. However, in practice there can be many different kinds of locking disciplines, and sometimes, correct access may be guaranteed by other techniques, such as temporal separation of accesses. In addition, it is often desirable to check invariants on shared data.

Many techniques have been devised for checking multithreaded programs for simple locking disciplines. However, more complicated ways of avoiding data races cannot be easily handled. Thread-modular verification(TMV) [2] addresses this problem. TMV is a technique for verifying synchronization properties of multithreaded programs without procedures. It works by analyzing one thread at a time, using assumptions on other threads in the environment. Techniques and tools for verifying sequential programs, such as ESC/Java[1], can then be leveraged.

To make TMV useful, one needs to add to it the ability to reason about procedures in a modular manner. Procedure-modular verification is a technique for verifying sequential programs with procedures, one procedure at a time, by using precondition/postcondition specifications at call sites. The goal of our work, then, is to combine the above methods to achieve modular verification of multithreaded programs with procedures. We realized that precondition/postcondition specifications for procedures do not suffice for modular verification in the presence of multiple threads. To overcome this problem, we propose a new procedure specification that overcomes some of the problems with precondition/postcondition specifications.

2. Insufficiency of pre/post conditions

Consider a concurrent program with a single shared variable x. This program uses a readers-writer (shared-exclusive) lock rwl to mediate access to x. Suppose that we want x to obey the invariant that it is always positive when no thread holds rwl in write mode. A fragment of Java code for this program is shown below with ESC/Java style annotations(the "run" method is executed by the thread that is being analyzed):

```
int x;
ReadersWriterLock rwl;
/*@ global_invariant !rwl.hasWriter ==> (x > 0) */

void run() {
   rwl.beginRead();
   x++;
   assert(x > 1);
   rwl.endRead();
}
```

One can convince oneself that the assert enclosed within the beginRead() and endRead() combination should pass. However, we are unable to show this using just pre and post conditions of beginRead(). The pre/post condition specification of beginRead() is shown below:

```
/*@ ensures isReader[currentThread] && !hasWriter */
void beginRead() {
  /* body */
}
```

Notice that we cannot state anything stronger about the post-condition. In particular, we would like to strengthen the post-condition with "x>0", but we cannot because x is only in scope in the client code for the readers-writer lock and not in the code for the lock itself. In fact, different clients might need different, possibly contradictory, ways of strengthening the post-conditions.

3. Atomic specifications of procedures

The fundamental problem described in the previous section (with using pre and post conditions) is that the specification we need is one which captures the atomic transition within beginRead() from the point when there is no writer to the point where the thread executing beginRead() atomically gets the read lock. We call this more precise specification as a "atomic specification" (also called a "performs" specification in our implementation).

```
/*@ performs (isReader[currentThread])
     { \old(!hasWriter) && isReader[currentThread] } */
void beginRead() {
  /* body */
}
```

The meaning of a "performs (v) {P}" specification is that atomically, on some transition in the procedure whose pre-state is governed by P, the thread executing the procedure changes the value of v according to P. The thread executing the procedure leaves v unchanged at all other points.

Given the "performs" specification of beginRead(), we can now use it at the call site to reason that the assert passes. To see this, notice that in the pre-state of the atomic action, !hasWriter is true. In addition, the invariant holds at this point. These two facts imply that x > 0. The predicate x > 0 continues to be true until the end of beginRead() and hence we can conclude that the assert passes.

The performs specification must also be checked on the method implementation. We do this using a monitoring state machine.

4. Implementation

We have extended ESC/Java to do thread-modular verification and to reason about procedures with atomic specifications. The major changes we made to ESC/Java were in generating and placing code that models actions by the environment (constrained by TMV assumptions), code to check for TMV guarantees, and code to check for "performs" specifications. We took an atomic block of statements executed by a thread to be one in which only one read or write to a word-sized shared data location occurs. We model actions by other threads as occuring between atomic blocks; these actions can modify shared state only in a way that is consistent with environment assumptions of the thread being analyzed.

5. Experimental Results

We ran the modified version of ESC/Java on several examples. The largest example was the part of the Mercator Web crawler [3] that deals with readers-writer locks. This code comprises about 1500 lines of Java source code, and we checked it for violations of constraints on writing shared data, and for a few data invariants. We did not find any bugs in Mercator, but we did find a known bug in java.util.Vector.

We also studied the annotation overhead in using TMV. We took the rather non-modular, but low-annotation-overhead, approach of inlining all non-public methods, and checking specifications only of interfaces (viz. public methods and fields). Even with the blow-up in the size of methods from inlining, ESC/Java and Simplify scaled well: for example, checks for all but two methods in Mercator took less than 10 minutes.

6. Conclusions and Future Work

Our initial experiments indicate that thread-modular verification extended with atomic specifications for procedures is a useful technique to check general synchronization properties and data invariants. The tool we built for Java has scaled well to work on real programs. Annotations tend to concentrate at the interfaces; if there is much interface reuse then the cost of adding annotations at the interface can be amortized over several uses.

There are many avenues for future work. We are currently working on safe optimizations to reduce the checking code and the number of places we have to insert checks; some of these have already been implemented. Reducing the annotation burden by infering "boilerplate" annotations (such as those for simple locks) can be added; more complicated annotation inference remains a challenge. Finally, exploring richer method specifications provides many avenues for further work.

References

[1] David Detlefs, K. Rustan M. Leino, Greg Nelson and James Saxe. Extended Static Checking. Research Report 159, Compaq Systems Research Center, Palo Alto, CA, December 1998.

[2] Cormac Flanagan, Steve Freund and Shaz Qadeer. Thread-Modular Verification. Submitted to POPL 2001.

[3] Allan Heydon and Marc Najork. Mercator: A Scalable, Extensible Web Crawler. World Wide Web conference, December 1999.