SRC Technical Note

1999-003

November 1999

Selected 1999 SRC Summer Intern Reports

Compiled by James Mason

COMPAQ

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

This document features informal reports by interns who spent the summer of 1999 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 the research staff. The primary goal of this technical note is to describe the summer research projects. However, 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.

Type-based Race Detection for Java

Stephen Freund, Stanford University

The summer project described here consisted of joint work with Cormac Flanagan, my host at SRC.

Introduction

A race may occur in a concurrent program when two threads access a shared memory location at the same time. This situation often causes unintended behavior ranging from memory corruption to execution failure. Since the effect of a race depends upon the interleaving of program execution, races may be difficult to locate and fix, even after their effects have been observed.

To avoid race conditions, programmers often adopt a programming discipline in which shared resources are guarded by locks. Before accessing any shared structure, the necessary lock must be acquired. This discipline ensures that no two threads ever access the same resource at the same time. Using locks in this fashion shifts the problem of preventing races to one of enforcing the locking discipline.

This project summary describes a static analysis technique that supports this locking discipline in concurrent Java programs. The analysis technique, which is presented in the form of the static type system, was designed so that it has the following desirable features:

1. a sound formal foundation
2. low programmer overhead
3. the ability to check a reasonable set of programming idioms

The following section describes an extension to the Java type system that captures locking information; the third section describes a prototype implementation, and the fourth section summarizes our experiences with it.

Type System and Annotation Language

The project's starting point was to work from a race-free type system for a concurrent object calculus [Flanagan and Abadi 1999] [2]. The most important features associated with the adaptation of this type system to Java are presented in the following examples. To preserve compatibility with the standard Java compilers, the additional type information used in our analysis is written in special comments in the code, similar to those of *escjava* [Leino et al. 1999] [4]. These annotations are comments that begin with the character '#'. The following class is a monitored counter:

```
class Counter {
    private int c = 0 /*# guarded_by this */;
    private void set(int x) /*# requires this */ {
        c = x;
    }
    public  void increment() {
        synchronized(this) {
            set(c+1);
        }
    }
}
```

The *guarded_by* annotation on the field *c* indicates which lock must be held to access that field, and the *requires* clause on the set method indicates which lock (or locks) must be held prior to invoking that method. To typecheck a program, a conservative approximation of the set of locks held at each program point is determined, and the checker then verifies whether the constraints expressed in the annotations are satisfied on each field access and method invocation.

As part of this verification process, the analysis needs to determine whether a specific lock is in the lock set. The set membership test requires some notion of equality between lock names, but since our analysis cannot rely on run-time values, we approximate run-time value equivalence with syntactic equality.

Another common programming idiom is to create unsynchronized classes and require the client to provide the necessary synchronization. This type of class may be expressed in our type system using classes parameterized by lock names. The following code example shows how to write a counter monitored by a lock in the client code:

```
class Counter/*# {ghost Object o} */ {
    private int value = 0  /*# guarded_by o */;
    private void set(int x) /*# requires o */ {
        value = x;
    }
    public void increment() /*# requires o */ {
        set(value+1);
    }
}

Object mutex = new Object();
Counter/*#{mutex}*/ c = new Counter/*#{mutex}*/();
```

It is often the case that a significant fraction of a concurrent program does not use synchronization at all. To avoid the need to require locks on objects that are not shared between threads, we introduce the notion of a *thread_local* class into the type system. A *thread_local* class is a class whose instances are never shared between threads, indicated with the annotation *thread_local* on the class declaration. This type of class requires no synchronization on field accesses, and a class may be thread local only if:

1. no instances of the class are stored in fields of a shared class
2. the class is not a subclass of *java.lang.Thread*

The first requirement is checked with a simple escape analysis. For technical reasons, programs using *thread_local* classes are assumed to have additional run-time checks on down casts. One final feature added to the type system is an escape mechanism to circumvent the analysis when it is too restrictive. As usual, it is the programmer's responsibility to ensure the validity of each use of these escapes.

Implementation

Rccjava, a prototype type checker, was implemented as an extension to an existing Java front-end. The main additions to the standard Java type checker were the algorithm to compute lock sets, the notion of syntactic equality, and classes parameterized by lock names. Several annotation inference techniques were also incorporated into the implementation in order to reduce the number of annotations required for large programs. These inference techniques include analysis to determine whether an unannotated class is *thread_local* or *thread_shared*, and also include the assumption that unannotated fields of shared classes are guarded by the lock of the self object.

Experimental Results

The prototype implementation was used to check race conditions in a number of programs. Four representative examples are:

1. the *java.util.Hashtable* class
2. the *java.util.Vector* class
3. Ambit, an implementation of mobile agents [1]
4. WebL, an interpreter for WebL programs [3]

The following chart summarizes the number of annotations required by *rccjava*:

Program	Size (lines)	Number of annotations	User time (hours)
Hashtable	434	46	1
Vector	440	15	0.5
Ambit	4,500	37	3
WebL	20,000	421	16

The large number of annotations in Hashtable may be attributed to the use of type parameters which require an annotation on each reference to a parameterized type name. The two larger examples required approximately 20 annotations per thousand lines of code. One race was found in the Vector class, and several races were found in the WebL code.

Conclusions and Further Work

The initial experiments with *rccjava* indicate that it is a useful tool for detecting races. While more difficult to use than dynamic tools like Eraser [Savage et al. 1997] [5], it is not subject to the same coverage problems as those tools. In addition, the annotation overhead is lower than some other static analysis techniques, such as using *escjava*. However, that tool captures a much broader range of program properties and has not been tuned specifically to race detection.

The most important direction for future work is to reduce the annotation requirements. We are currently exploring better annotation inference algorithms and the possibility of using feedback from dynamic tools to help infer annotations. There are also additional features to implement, such as reader-writer locks and parameterized methods.

References

[1] Luca Cardelli, Mobile ambient synchronization. Technical Note 1997-013, Digital Systems Research Center, Palo Alto, CA, July 1997.

[2] Cormac Flanagan and Martin Abadi. Object Types against Races. Proceedings of CONCUR 99, August, 1999.

[3] Thomas Kistler and Johannes Marais. WebL - a programming language for the Web. In Computer Networks and ISDN Systems, Volume 30, pages 259-270. Elsevier, April 1998. Also appeared as SRC Technical Note 1997-029.

[4] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq Systems Research Center, Palo Alto, CA, May 1999. Also appeared in Formal Techniques for Java Programs, workshop proceedings. Bart Jacobs, Gary T. Leavens, Peter Muller, and Arnd Poetzsch-Heffter, editors. Technical Report 251, Fernuniversitat Hagen, 1999.

[5] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. ACM Transactions on Computer Systems (TOCS), 15(4):391-411, November 1997. Also appeared in Proceedings of the Sixteenth ACM Symposium on Operating System Principles, October 5-8, 1997, St. Malo, France, Operating System Review 31(5), ACM Press, 1997, ISBN 0-89791-916-5, pp 27-37.

Hilltop: Experiences Building a Web Search Engine

George Andrei Mihaila, University of Toronto

I'm currently in the last year of my Ph.D. program at the University of Toronto, working with Alberto Mendelzon, my advisor. My research is in the area of Web querying and database integration. During my summer internship at SRC I worked with my host, Krishna Bharat, on improving the quality of Web search results.

More specifically, I worked on designing and implementing a connectivity-based search engine. We started from Topic Distillation algorithms. Topic Distillation algorithms assign authority scores for pages by computing the principal *eigenvector* of the adjacency matrix of a subgraph of relevant pages. According to this scheme, highly connected pages receive a high score which roughly corresponds to the subjective notion of a good quality page. The intuitive reason for this is that if a page is referred from many different places, it is probably a good page. The key factor influencing the quality of the results is how the subgraph is selected. Typically, this subgraph is constructed by taking the back and forward links from a small set of pages on the query topic. This method has the potential of including pages which are not germane to the topic--which in turn affects the final ranking.

Our objective was to design a conservative version of Topic Distillation in which we consider only a small fraction of the Web, carefully selected based on connectivity properties.

The first step in implementing this idea consisted in pre-computing the set of pages we needed to work with. For this, I used SRC's connectivity server which maintains connectivity information about a large fraction of the Web (about 150M pages). For me, this was a great opportunity, as I was able to easily test several hypotheses using real data, without the need to access the Web (which would have been much too slow). By applying several connectivity-based filters on this data, a set of about 2.5M URLs was selected. We then used the SRC's Mercator Web crawler to download all these pages. Once we had this data, I wrote a program using the NI2 indexer library to build an index of the relevant text and links from these pages. This too was a very interesting experience for me, as it provided the opportunity to learn about the data structures and operation of the inverted index system in NI2

Finally, my host and I discussed several ranking algorithms. After experimenting with a number of different ranking functions, we decided on a final algorithm and I implemented it on top of the NI2 query library. As a last step, I wrote a limited HTTP server program that provides Web access to the query interface.

After everything was up and running, we needed to test it with real users in order to compare it with other popular search engines. My co-summer interns were very helpful in this phase, as they volunteered to participate in our user study. At this stage of the project, I learned a lot about conducting objective tests with information retrieval systems. It was also satisfying to see people finding what they were looking for among the highest ranked results.

In conclusion, my summer internship at SRC provided an excellent learning environment. Having the opportunity to work with such knowledgeable and enthusiastic people was truly inspirational.

Toward More Informative ESC/Java Warning Messages

Todd Millstein, University of Washington

Introduction

This year will be my fourth as a PhD student in computer science at the University of Washington. My advisor is Craig Chambers, and my research involves the study of object-oriented programming languages that support multimethods. In particular, I have designed modular static typechecking algorithms for such languages.

I chose to do an internship at SRC for several reasons. I have been interested in formal methods for many years, so I was happy for the opportunity to learn about the challenges and tradeoffs involved in creating a practical program verification tool. I also knew a bit about SRC and thought that it would be a fun and exciting work environment. Finally, I was interested in taking a break from my PhD research and getting a glimpse of life outside academia.

Extended Static Checking

The Extended Static Checker for Java (ESC/Java) is a tool designed to catch common programming errors at compile-time. A programmer writes an ordinary Java program but adds annotations, such as pre- and postconditions on methods. Each annotated method is translated into a logical formula which is valid if and only if the method meets its specification. A theorem prover searches for counter-examples to the logical formula, which correspond to possible errors in the original Java method. A key design principle of ESC/Java is modular checking, which means that each method is checked in isolation, given only the specifications, and not the implementations, of other methods in the program.

More Informative Warning Messages

Reducing program checking to theorem proving allows ESC/Java to check a wide variety of program properties and to leverage existing theorem proving technology. However, this architecture makes error reporting challenging. In particular, it is difficult to turn counter-examples from the theorem prover into useful user-level messages about the original Java method.

Previously, ESC/Java was able to deduce from a counter-example which ESC/Java annotation had failed to verify. For example, if a method's postcondition is potentially violated, ESC/Java reports this information to the programmer. My summer project consisted in extending this "report warning" in a way that would generate more specific (and useful) information. We were most successful in targeting specific programming situations that ESC/Java users have found hard to understand.

Our work took the following direction: Since methods often have numerous possible execution paths, depending on the choice made at each branching point, we implemented a way to extract--from the counter-example--a complete trace of the particular execution path through the method that caused the warning to occur.

We then targeted two specific kinds of ESC/Java warnings that can be hard to understand:

The first--a common warning--occurs when a class has a particular kind of sharing constraint on one of its fields, where the sharing constraint is insufficiently annotated. For this we devised ESC/Java support for specifying this constraint and for understanding when an implicit constraint of this kind is potentially violated.

The second involved providing support for suggesting annotations to remove spurious warnings related to Java's covariant subtype rule for arrays. In particular, *S[]* is treated in Java as a subtype of *T[]* when *S* is a subtype of *T*. The use of this rule is not always provably safe at compile-time. Therefore, Java enforces a run-time safety check on arrays. Because of this potential lack of compile-time safety, ESC/Java issues warnings unless there are sufficient annotations to allow arrays to be verified statically.

Reducing the ESC/Java Annotation Burden

For the last few weeks of my internship, I worked on a completely different problem--trying to reduce the annotation burden on programmers. If ESC/Java is run on an unannotated program, many spurious warnings will result (for example, a pointer dereference will cause a null dereference warning to be issued). Our observation is that spurious warnings can be reduced by checking a method using some context from its callers and callees, rather than performing completely modular checking. To this end, we implemented an infrastructure in ESC/Java to allow method calls to be inlined and checked by ESC/Java in several different ways. We have just begun designing experiments that make use of this new infrastructure.

Chasing Races

Silvija Seres, Oxford University

This is a brief report on the work performed by Silvija Seres during her summer research internship at SRC in 1999, together with her host, Greg Nelson. The main theme of the work was evaluating two tools developed at SRC for checking the correctness of multi-threaded Java programs: Eraser, a dynamic race detector, first described at SOSP in 1997 [1], and ESC/Java [2], a verification-based static program checker. There are two main research results: porting Eraser to work on Java programs, and applying ESC/Java (for the first time) to a sizable multi-threaded program. As a source of test cases we used the Mercator [3] Web crawler by Allan Heydon and Marc Najork.

Our goal was not to build tools but to evaluate them. Nevertheless, some tool-building was inevitable. To achieve the necessary expressibility with ESC/Java, we needed to add one new pragma and some flexibility for the use of so-called ghost variables. With the help of Sanjay Ghemawat, we implemented Eraser for Java. The Eraser algorithm finds races by checking all loads and stores to detect shared variables that are accessed by different threads without using a lock to ensure mutual exclusion.

While checking Mercator, we identified six basic concurrency control techniques:

1. In the "monitored object" technique, the methods of a class provide internal synchronization before accessing shared variables.

2. In the "client locking" technique, this synchronization is the responsibility of the client of the shared class.

3. In the "per-thread object" technique, no particular instance of the shared class is accessed by more than one thread.

4. In the "read-only data" technique, thread-safety is achieved by making the data immutable after initialization.

5. In the "exclusive/shared" technique, the program uses exclusive/shared locks, also called reader/writer locks, which are a well-known kind of lock that can be held by a thread either in exclusive mode or in shared mode.

6. In the "temporal separation" technique, conflicting accesses are guaranteed to be non-concurrent because the computation imposes an order on them. There are many examples, of which the most typical is a pipeline in which the shared data is accessible to one thread per stage.

We have also found places in Mercator where these techniques are used in combination.

ESC Java's annotation language was designed to specify the monitored object technique, but all the others represent specification challenges. Eraser deals with the first four techniques automatically, and the rest require minimal annotations. We have designed annotation strategies to allow ESC/Java checking of all the techniques. Our strategies make heavy use of ghost variables and of the ESC/Java annotations "defined_if" and "writable_if" (the latter of which we had to add in the course of our research).

We haven't annotated all of Mercator for ESC/Java, so all of the races that we found were found with Eraser. Mercator consists of approximately 25,000 lines of code, and Eraser produced fifteen warnings on the first run. Of these warnings, two were real races and the rest were false alarms. The two races were fixed, and sixteen lines of Eraser annotations were added to supress the false alarms.

We have also checked Pachyderm [4], the Java mail and news system built by Andrew Birrell et. al. There were a large number of warnings, primarily because Pachyderm uses the Java windowing classes, which seem to be a fulsome source of races. Andrew claimed that three of the warnings were worth acting on in his code.

Our first conclusion is that Mercator uses more synchronization techniques than were envisioned by the designers of ESC/Java.

Our second conclusion is that Eraser is more effective than ESC at finding race conditions, especially given a large, unannotated program.

Third, ESC/Java requires far more annotation work. A result of this, however, is that it allows for the documentation of the synchronization design of the program.

We have not experienced coverage to be a problem with either tool, but it seems still to be an open issue for each tool, for different reasons. In Eraser we don't have a clear coverage measure, because in addition to passing through each branch of the program at least once, one needs to check that each shared variable has been accessed by at least two threads.

References

[1] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. ACM Transactions on Computer Systems (TOCS), 15(4):391-411, November 1997. Also appeared in Proceedings of the Sixteenth ACM Symposium on Operating System Principles, October 5-8, 1997, St. Malo, France, Operating System Review 31(5), ACM Press, 1997, ISBN 0-89791-916-5, pp 27-37.

[2] David Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, CA, December 1998. Also see: Extended Static Checking Web page.

[3] Allan Heydon and Marc Najork. Mercator: A Scalable, Extensible Web Crawler. To appear in World Wide Web, December 1999. See also: The Home page of the Mercator Web Crawler.

[4] The Pachyderm Email System

Directional Array Microphone

Richard Turner, University of Belfast

I am working towards a PhD at The Queen's University of Belfast and collaborated with Stefan Ludwig and Bob McNamara on a board called the Directional Array Microphone (DAM).

Bob McNamara was looking into microphone arrays and thought they were interesting things to investigate. So a piece of general-purpose data capture and delivery hardware was designed and built. This summer I wrote the firmware of the FPGA, implemented some tools to interact with the DAM board, and did some fun DSP stuff with the microphone array.

The DAM board is made up of eight Analog to Digital Converters and a connector for daughter boards. There can be a maximum combination of 40 inputs and/or outputs per DAM board (where the daughter boards supply the extra inputs and outputs). If 40 microphones are not sufficient for the task, multiple boards can be synchronized to build larger systems and two Ethernet physical layer chips are supplied to do this. The board has an EPROM for configuration data for the FPGA (firmware). The firmware is used to combine the different parts of the board and daughter boards together.

I spent most of the summer working on the firmware. It generates the control signals for the ADs while the data returned from the ADs is assembled together into packets. As there is no Ethernet controller on the board, the firmware has to place the protocol information at either end of the packet and send it to the Ethernet link. It also receives packets which can be used for the reprogramming of the EPROM. The DAM uses one of the 100 Mbps Ethernet links to transport data to and from a PC while the software on the PC is used for digital signal processing.

What interesting things, then, can be accomplished using an array of microphones? Two areas, in particular, appeared worthy of research: high-quality speech acquisition and the location of sound sources.

Speech recognition software requires high-quality speech with low background noise and little change in the quality of the signal. A straighforward solution would be to use a head microphone. As the microphone is close to the source of sound, the signal from the microphone will mostly contain the required speech with little background noise. However, if we place the microphone a more convenient location further away, more background noise will be introduced into the signal. Now, if we use more than one microphone, "beamforming" can be performed. What this means is that we adjust the gain of sound depending on the direction of arrival. So in the direction of the sound we want to capture, we can have a large gain, while we opt for a low gain from the rest of the arrival directions. As a result, we implemented a simple delay & sum beamformer.

The other task we accomplished with the microphone array was to track the sound source. Once we calculate the time difference of the sound arrivals between two microphones, we can work out the direction of the sound through triangulation measurements. This information is then fed back into the beamformer. To demonstrate this, we wired up the microphones in SRC's forum and showed how the system figured out the position of the speaker.

Early Anaylsis Techniques for ProfileMe

Kip Walker, Carnegie Mellon University

My summer internship at SRC coincides with the end of my third year in the Computer Science PhD program at Carnegie Mellon. My work there is focused on adaptive mobile information access in the Odyssey project, with Professor Satyanarayanan, known more widely as Satya, as my advisor.

The goal of my summer work was to initiate analysis techniques for the ProfileMe data generated by Alpha 21264A processors. The Continuous Profiling Infrastructure was able to display raw events and aggregate event samples in simple ways, but no code existed for explaining static and dynamic stalls, calculating the cost of traps, and other important analyses.

The early part of the summer was consumed by becoming aquainted both with the large existing code-base, and the architecture of the 21264A. Technical documentation was certainly useful, but the best learning tool turned out to be the register-transfer-level simulator GUI. Simple code sequences could be simulated, with some of the internal processor state displayed in graphical form. Since CPI analysis techniques are based on a solid understanding of the inner dynamics of the chip being profiled, this background work was necessary for my project.

The first problem I tackled was one of software engineering. Adding specialized analysis routines for ProfileMe data illustrated the need to create an interface for determining what analyses could be performed given particular samples. For example, neither of the primary data analysis methods (execution count estimation and blame assignment) were initially supported for ProfileMe samples.

The obvious follow-on to this project was supporting execution count estimation using ProfileMe samples. This was easily accomplished, since the number of ProfileMe samples for a given instruction is directly proportional to the number of times it executed! Event-based sampling necessitated tricky heuristics for deriving the execution count from the number of samples.

The second phase of the project involved trying to "recover" cycles which were impossible to measure using the ProfileMe hardware as implemented. Several conditions cause the "retire delay" number reported to be shorter than it actually was. By consulting the RTL simulator and hardware specification, we were able to determine where a portion of the missing cycles were being spent.

The bulk of the summer ended up being focused on the analysis of ProfileMe data about "_trapping_" and "_aborted_" instructions. In an out-of-order processor, significant amounts of time may be spent on instructions that are ultimately discarded. Raw data from ProfileMe can identify instructions that frequently trap, but cannot convey the cost of the traps, and may not even clearly indicate the cause of the trap!

One technique we developed was on off-line algorithm for processing trap and abort samples to estimate how many aborts were due to each trapping instruction; this is a good first step at determining the cost of each trap. The success of this method needs to be analyzed better. Because of the statistical nature of the data we gather, and the complexity of run-time behavior, it is very difficult to correlate aborts with the traps that caused them. With more time and clever heuristics, I'm sure this tool will prove to be useful.

A second technique focused on replay traps, where two memory operations interact dynamically in such a way that bad data may be seen by future instructions. To fix such traps, one must know the identity of both instructions involved; only one of the instructions is directly identified by the ProfileMe hardware. We came up with a novel solution for this problem.

I found SRC to be a wonderful environment, full of very friendly and brilliant people, focused on a variety of important problems, and possessing a spirit of collaboration that was refreshing. Much credit goes to Sharon Perl, for making things run so smoothly. In addition, many thanks to my hosts, Bill Weihl and Mark Vandevoorde, and the other great people with whom I got to work closely.

The MS Powerpoint presentation entitled Using Interpretation for Profiling the Alpha 21264a provides additional details on the above work.

Verifying Temporal Formulas in the Temporal Logic of Actions

Lucian Wischik, University of Cambridge

This report describes a summer project undertaken by Lucian Wischik of the University of Cambridge, at Compaq Systems Research Center. He was supervised by Leslie Lamport and Yuan Yu.

The Temporal Logic of Actions

"Engineers should be able to specify and verify their systems directly and conveniently in logic." This comment offers a pragmatic motivation for using Lamport's Temporal Logic of Actions (TLA), and its associated model-checker (TLC). The comment also implies a distinction between TLC and other comparable model-checkers, where specifications must be translated into a special-purpose language. (A model of a specified system, in this context, is a sample execution trace. To 'check' a model is to ensure that it satisfies the given verification conditions).

The goal of the project was to extend TLC to verify arbitrary temporal formulas. Hitherto, it could verify only "always" formulas about states:

```
   "always, the state will be such with no dangling pointers"
```

Now it can verify more general formulas:

```
   "if ever I put a datum into my reliable-transport-protocol,
   then it must eventually come out the other end."
```

The problem of verifying arbitrary temporal formulas about states is a standard one, addressed by other model-checkers as well. It is solved using the "tableau" technique of Clark and Emerson [1981]. We briefly outline this technique below. (Temporal formulas are ones involving the predicates `[]` always, or <> eventually. For instance, `[](a => <>b)` means that every occurence of `a` must eventually be followed by an occurence of `b`).

However, the application of this technique raises new problems for the Temporal Logic of Actions. In fact, they arise as a direct consequence of the very *actions* that give the logic its name. (Actions relate the next state of the system to the current state and describe the transitions of the system. An example action is: `x'=x+1`.)

Checking fairness: disjunctive normal forms

The first problem concerns *fairness* criteria. A fair transition is one that must eventually be taken (assuming that it is possible). In conventional model-checkers, fairness is specified explicitly in the special-purpose specification language. For example, to say that the scheduler must eventually allow the process to proceed, we might write:

```
   fair; x := x+1;
```

In TLC, fairness is expressed as a logical predicate on actions, i.e., "Always, eventually, `x'=x+1`"

```
   [] <> (x'=x+1)
```

Note that this is merely a logic formula, and can appear anywhere in the specification or verification conditions (whereas the keyword `fair` can obviously appear only as part of the specification program).

The new technique we introduced to deal with such formulas, in specification or verification conditions, involves converting the problem into *disjunctive normal forms*. These forms always have the same structure:

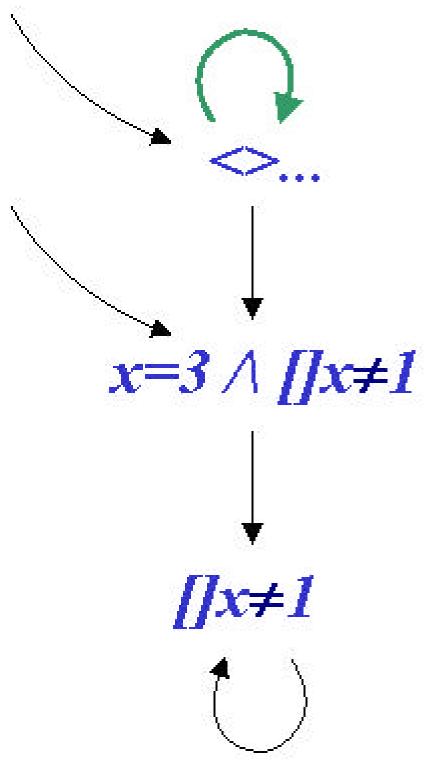
```
      (<>[]ea1 /\ []<>ae1 /\ misc1)
   \/ (<>[]ea2 /\ []<>ea2 /\ misc2)
   \/ ...
```

Observe that, within each disjunct, the fairness formulas `<>[]` and `[]<>` are all gathered together. This gives them a straightforward decision procedure: an infinite cycle in the system's behaviour satisfies `<>[]p` if `p` is true *everywhere* in the cycle; and it satisfies `[]<>q` if `q` is true *somewhere* in the cycle. We provided an algorithm to convert arbitrary expressions into normal form, proved that the conversion preserved meaning and that the normal forms were unique, and implemented the decision procedure.

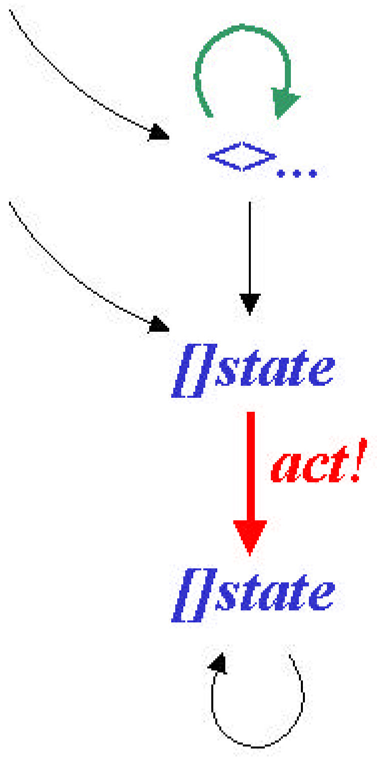
Checking arbitrary temporal formulas: the tableau

The fairness formulas above are a generalisation of the fairness specifications present in other model-checkers. But the Temporal Logic of Actions actually allows even greater generality -- it allows arbitrary temporal formulas involving actions and states. To handle such formulas in full generality requires that the standard tableau technique be modified to handle actions. Unfortunately, a way to implement this modification was not discovered until late in the summer, so there was insufficient time to fully develop the theory. (The disjunctive normal forms, although just a special case, are still important: they are essentially an optimisation that reduces the exponential cost of the fairness tableau).

The tableau technique is as follows: To check whether a sample execution trace satisfies a given temporal formula on states -- for instance, `Not ([] (a => <>b))` -- we construct a particular (non-deterministic) finite state machine. This machine accepts only those traces which satisfy the formula. We run the sample execution-trace in parallel with the machine. If the trace is accepted by the machine, then it satisfies the temporal formula! The machine and example formula is illustrated below:



The suggested modification of the technique, so that it can check arbitrary formulas on actions as well as on states, is illustrated in the example machine below:



$$\Diamond(act \land \Box state)$$

Implementation

The techniques described above--for converting the problem of fairness criteria into disjunctive normal forms, and for the tableau technique for checking arbitrary temporal formulas--were implemented within TLC. As presented, the techniques may be prohibitively expensive in time and space. They have not yet been tested on real-world examples. It remains an open question as to whether there is any practical, engineering benefit to the verification of temporal formulas. Hopefully, the two new techniques introduced in this summer project will eventually lead to an answer.