CTPLAN: A planning-based approach to automatically detecting flaws in concurrent algorithms

Deborah A. Frincke   Myla Archer   Karl Levitt
University of California, Davis

Abstract

Development of correct algorithms for the synchronization of concurrently executing processes can be a difficult task. Most concurrent program debuggers have been developed in order to assist the programmer once an error has appeared. This paper takes an alternate approach of attempting to discover the presence of possibly undetected flaws. This approach is embodied in a prototype system (CTPLAN) that may be used to develop tests for common flaws in concurrent programs. To illustrate our methodology, CTPLAN has been used to detect errors in several algorithms, most notably the incorrect simplification of Dekker's algorithm described in [Hym66]. CTPLAN can in principle be used to detect indeterminacy, deadlock, livelock, violation of mutual exclusion, unfairness, and starvation.

1 Introduction

Development of correct algorithms for the synchronization of concurrently executing processes can be a difficult task. Potentially, one must consider all possible interleavings of process execution before the algorithm may be said to be correct. Most concurrent program debuggers have been developed in order to assist the programmer once an error has appeared. Some concurrent program debuggers exhibit the 'probe effect'; i.e., the debugger actually masks flaws in the program due to its insertion of print or debugging statements. These statements can alter the timing (and thus the interleaving) of process execution, and therefore fail to expose certain faults [MH89]. There are debuggers that can detect potential race conditions that may occur during execution; unfortunately, these 'intelligent' debuggers frequently lead to so many false positives that their value is greatly reduced[HKMC90]. Verification techniques for concurrent programs have been extensively studied, the goal being to avoid the exploration of all execution interleavings[Bar85][FS81].

This paper describes an alternate approach to debugging and verification, namely, attempting to produce process execution sequences that lead to errors. In terms of cost and coverage, our approach to testing lies between conventional debugging and verification. It is more systematic than debugging, but cannot explore all of the cases possible with verification. However, our approach gives a counter example illustrating a detected flaw, something that it is difficult for a verification system to produce. Our approach is embodied in a prototype system, CTPLAN, that may be used to develop tests for common flaws in concurrent programs. We see CTPLAN as an intermediate step in the development process, performed after simple testing and before verification. The kinds of errors that CTPLAN can detect are more subtle than those that can be quickly found by conventional testing; however, these errors should be removed before beginning the expensive process of formal verification. CTPLAN, written in Prolog, is based upon the methodology used in TPLAN [FLA90] to detect security flaws in operating system specifications. CTPLAN operates on the transformed code of concurrent algorithms, and uses classical Artificial Intelligence planning techniques [Nil80] to develop tests to detect flaws. CTPLAN attempts to find a sequence of operations (a plan) that produces the specified flaw, as contrained by time and memory limitations.

CTPLAN can detect a variety of flaws, including deadlock and violation of mutual exclusion. Algorithms from [PS87] and [Ray86] were used as examples. A number of concurrent algorithms containing flaws have actually been published, e.g. in [Hym66], [VA86], and others. It is of particular interest that CTPLAN can be used to detect these flaws. In this paper, it is shown how CTPLAN detects the error in the incorrect simplification of Dekker's algorithm described in [Hym66].

Use of a system such as CTPLAN is of the most benefit when the programmer is faced with the task of determining whether or not a program contains a specific behavioral flaw. When dealing with synchronization of processes, it is often easier for the programmer to state what should not happen, rather than what should happen. This specification is the goal which CTPLAN uses to construct a plan. For example, per-

0-8186-2605-4/91 $1.00 © 1991 IEEE

mitting two processes to modify the value of a global variable at the same time is usually undesirable. This flaw, a violation of mutual exclusion, may be described easily:

$$\neg((\text{ Process 1 executes statement } P_1 \text{ modifying variable X at time t}) \cap (\text{ Process 2 executes statement } P_2 \text{ modifying variable X at time t }))$$

However, the algorithm that actually prevents this from happening is much more difficult to state. Further, the granularity of the algorithm's encoding may also affect the presence of a flaw, as well as CTPLAN's ability to detect it.

A major contribution of this work includes the representation of characteristic properties such as deadlock, livelock, and indeterminacy as predicates that become the goal for a planner. Additionally, certain heuristics that reduce the search space, such as loop detection, are identified.

In Section 3, we discuss in greater detail ways that CTPLAN can be applied to classical algorithms to detect flaws. In Section 4 we discuss ways in which various algorithms are actually encoded for CTPLAN's use, including the effect of the granularity of the statement translation. In Section 5, we give a brief overview of CTPLAN's Prolog implementation. In Section 6, we describe our ongoing work in applying CTPLAN to additional classes of flaws (such as lack of fairness), and improvements in CTPLAN's implementation.

2 Related Work

Other authors have studied methods for exposing flaws in software. Typically, as in [OFT81], dataflow analysis techniques have been used to study sequential programs. Taylor[Tay83] has extended existing techniques to concurrent programs, emphasizing detection of parallelizable code segments with special attention to Ada. Such segments would be candidates for interleaving and would have to be so explored in a testing system. Knowledge-based techniques have also been applied to the problem of debugging. [Sev87] identifies kinds of knowledge that a debugger could use—for example, knowledge about what a program should do and should not do, likely flaws (especially in concurrent programs), and the granularity of testing. The tools surveyed do not attempt to automatically generate test cases. A more recent knowledge-based debugging system is described in [TFC89], which uses a knowledge base to reduce the data from a debugging session to allow for more easily understood replays.

3 Applying CTPLAN to specific algorithms

The most common flaws in algorithms involving concurrently executing processes are: violation of mutual exclusion, deadlock, livelock, indeterminacy, fairness violations, and starvation. In this section, CTPLAN will be applied to some specific algorithms to exhibit how three of these flaws (mutual exclusion, deadlock, starvation) can be detected.

The algorithms within this section have certain common characteristics. The most important is that processes following the algorithm each contain a ***critical section*** of code. The purpose of mutual exclusion is to prohibit more than one process from executing its critical section at a time. Processes may manipulate their own local variables, or shared variables; in general, the critical section of code is used to read or modify a shared variable. Some of the algorithms shown include ***semaphores***. Semaphores may be considered special variables that are manipulated by the atomic instructions `P(sem)` and `V(sem)`. The instruction `P(sem)` will block further execution by the process until the value associated with `sem` is nonzero, and will then decrement this value and permit execution to continue. The instruction `V(sem)` never blocks, and increments the value associated with `sem`; its usual purpose is to unblock a process waiting on `sem`.

In [Dij65b], the following constraints were given for any algorithm correctly solving the mutual exclusion problem:

1. No assumption is made concerning the instructions or number of processes supported by the machine, except that reading, writing, or testing is considered to be atomic.

2. Execution speed of competing processes is assumed to be non-zero.

3. Processes in non-critical section cannot prevent another process from entering the critical section.

4. A process requiring access to a critical section cannot be delayed indefinitely.

3.1 Exclusive access to critical sections

3.1.1 Readers/Writers

The classical readers/writers problem involves a data item that is shared by competing concurrent process, some reading from the data object, others modifying the data object. In a correct solution to this problem, a reader and a writer process may not both manipulate the shared data object simultaneously, though multiple reading of the object is allowed [CHP71]. There are many variations to this problem, for example blocking reader processes when writer processes are waiting. Alternately, one may grant priority to processes based on order of arrival, but still permitting multiple readers to share the object. The simplest version of readers/writers, implemented using semaphores, is shown in Figure 1. CTPLAN easily detects the flaw in the writer process shown in Figure 1 (a writer process is never blocked, and thus may interfere with a reader process), and can handle versions containing multiple reader processes and writer processes as well single instances of the reader process and writer process.

3.1.2 Hyman's Simplification of Dekker's algorithm

As mentioned earlier, producing a correct algorithm for mutual exclusion is nontrivial. This is best exhibited by Hyman's 'simplified' version of Dekker's Algorithm for mutual exclusion involving two processes, published in *Communications of the ACM* in 1966 [Hym66]. This algorithm, shown in Figure 2, does not in fact prohibit the two processes from entering their critical sections at the same time. Dekker's Algorithm, which correctly enforces mutual exclusion, is shown in the same figure.

Raynal describes the flaw in Hyman's algorithm thus: turn is initially 0, and Process 1 sets `flag[1]` true, and then finds `flag[0]` false. Process 0 will then set `flag[0]` to true, find turn is 0, and enter its critical section. Process 1 then assigns turn the value 1 and also enters its critical section. Figure 3 describes a session with CTPLAN that uncovers this flaw. Figure 2 gives the CTPLAN encoding of Hyman's algorithm when the process id is 1 (transformed incorrectP1). Note that In describes the initial state, Out describes the flaw (inCS1, inCS2) are simultaneously true), Result describes the entire state when the test plan terminates, and Plan describes the sequence of process interleavings that result in the flaw (Section 5 describes state specification in more detail). incorrectPc1, incorrectPc2 refer to the program counters belonging to Process 1 and Process 2; similarly, inCS1, inCS2 are boolean variables that correspond to Process 1, Process 2 executing within their critical sections.

3.2 Deadlock and Starvation: Dining Philosophers

The Dining Philosophers problem, proposed and solved by Dijkstra[Dij65a], is another classic synchronization problem. In its simplest form, one considers $2N + 1$ philosophers seated around a circular table, with $2N + 1$ forks between them. In the middle of the table there is a bowl of food. Each philosopher alternately eats and thinks. In order to eat, the philosopher must pick up the two forks on either side (clearly, not all philosophers can eat at once). When a philosopher is done eating, the forks are replaced.

The non-solutions to this problem are interesting, in that they may exhibit both ***deadlock*** and ***starvation.*** Deadlock may occur if the algorithm permits hungry philosophers to hold forks that they are not using; for example, all the philosophers might decide to pick up their left forks at the same time. On the other hand, starvation may occur if the algorithm can in certain situations prevent a hungry philosopher from ever holding a fork; if philosophers $K + 1$ and $K - 1$ exhibit the following pattern, then philosopher K will starve: philosopher $K + 1$ becomes hungry and eats, philosopher $K - 1$ becomes hungry and eats, philosopher $K + 1$ stops eating, philosopher $K + 1$ eats again, philosopher $K - 1$ stops eating, etc.

CTPLAN's current support for deadlock and livelock detection is limited. Given a state, CTPLAN can determine whether or not that state is in fact deadlocked (or livelocked). Additionally, CTPLAN can develop a test plan that will achieve the deadlocked state, if such a plan exists. Work is in progress to improve CTPLAN's support for detection of these flaws; in particular, it is expected that CTPLAN will be able to detect deadlock in many cases without requiring the programmer to give specifics about the actual locked state. CTPLAN's support for starvation detection is based on its ability to show livelock and lack of progress. By instrumenting the transformed algorithm with dummy variables, the programmer can use CTPLAN to generate a sequence of instructions that show that particular processes are prevented from making progress. If these instructions form a cycle, it indicates that the algorithm may permit starvation.

```
=====================
  Reader Process
=====================

P(mutex) ;
 readcount := readcount + 1 ;
 if readcount = 1 then P(wrt) ;
V(mutex) ;
<critical section: reading>
P(mutex) ;
 readcount = readcount - 1 ;
 if readcount = 0 then V(wrt) ;
V(mutex) ;
```

```
=====================
  Writer Process
=====================

P(wrt) ;
<critical section: writing>
V(wrt) ;
```

```
=====================
Flawed Writer Process
=====================

<critical section: writing>
```

Figure 1: A correct solution to readers/writers.

```
=============================================================================
Dekker's Algorithm (correct)
=============================================================================
1::flag[i] = true ;
2:: while flag[j]  do
3::  if turn = j  then
4::     begin
5::         flag[i] = false ;
6::          while turn = j
7::              do skip
8::              enddo ;
9::         flag[i] = true ;
10::     end
11::  endif
12:: enddo
13:: <critical section>
14:: turn = j ;
15:: flag[i] = false ;
```

```
Hyman's incorrect algorithm
1::flag[i] = true ;
2:: while turn <> i  do
3::      while flag[j]  do
4::          skip ;
5::        enddo ;
6::    turn = i ;
7::  enddo ;
8::<critical section>
9::flag[i] = false ;
```

```
Transformed incorrectP1
1::flag1 = true
2::if turn /= 1 then goto 3
2::if turn == 1 then goto 10
3::<label>
4::if flag0 == true then 5
4::if flag0 /= true then goto 7
5::<label>
6::goto 4
7::<label>
8::turn = 1
9::goto 2
10::<label>
11::inCS2 = true
12::flag1=false, inCS2 = false
```

Figure 2: Dekker's (correct) and Hyman's (incorrect) Algorithm; CTPLAN translation of Hyman's algorithm for Process 1.

```
: findPlan(6, In, Out, Plan), updateState(In, Result, Plan)?

In = state(symtab([]),
   symtab([[turn, 0], [inCS1, false], [inCS2, false],
        [flag0, false], [flag1, true]]),
   symtab([[[incorrectPc0, 1], 0], [[incorrectPc1, 1], 0]]))

Out = state(symtab([]),
    symtab([[turn, dontcare], [inCS1, true], [inCS2, true],
         [flag0, dontcare], [flag1, dontcare]]),
    symtab([[[incorrectPc0, 1], dontcare], [[incorrectPc1, 1], dontcare]]))

Plan = [[[incorrectPc1, 1], 1],
      [[incorrectPc1, 1], 2],
      [[incorrectPc1, 1], 4],
      [[incorrectPc0, 1], 1],
      [[incorrectPc0, 1], 2],
      [[incorrectPc0, 1], 11],
      [[incorrectPc1, 1], 8],
      [[incorrectPc1, 1], 9],
      [[incorrectPc1, 1], 2],
      [[incorrectPc1, 1], 11]]

Result = state(symtab([]),
      symtab([[turn, 1], [inCS1, true], [inCS2, true],
           [flag0, true], [flag1, true]]),
      symtab([[[incorrectPc0, 1], 11], [[incorrectPc1, 1], 11]]))
```

Figure 3: CTPLAN's version of the flaw in Hyman's Algorithm.

4 Encoding the algorithms

The way in which algorithms are encoded for CTPLAN has an enormous effect on the type of flaw that may be detected. In particular, certain flaws will only be detected if the statements are translated with a fine-grained level of atomicity, and others will be more easily detected with a large-grained level of atomicity, due to reduced search time. This section first discusses the language features currently implemented by CTPLAN, then describes how granularity affects the flaws CTPLAN can detect, and finally describes how scheduling of statements of competing processes is implemented.

4.1 The Language

At present, CTPLAN uses a Pascal-like minilanguage language to describe algorithms (see Figure 4). The only specialized software support for synchronization is the semaphore. This structure was included so that algorithms that use such structures could be easily implemented and processed; in addition, other specialized language structures used for synchronizition (such as monitors and conditional critical regions) may be readily implemented using semaphores. **Test-and-Set** and **Swap** are included because these instructions are fairly typical of the type of hardware level support provided for synchronization; they are defined to be atomic [PS87]. Once an algorithm has been described in this fashion, it is translated into Prolog statements such as the one in Figure 7[1], which corresponds to the first line in Hyman's Algorithm (Figure 2).

This language subset is sufficient to describe a wide range of algorithms. For example, Figure 5 shows the translation of a **while-loop** into CTPLAN's minilanguage. However, it is often necessary to implement certain language features in terms of these atomic statements, which may affect the flaws that can be detected.

4.2 Granularity

As mentioned earlier, the granularity of the algorithm translation affects the flaws that may be discovered. For example, consider Figure 6, which shows two translations of a simple conditional statement into CTPLAN, where B is a global variable initialized to 0 and *Cond* is a local variable (note that only one of the statements Atomic::1a, Atomic::1b will execute; similarly, only one of the statements NonAtomic::2a, NonAtomic::2b will execute).

[1]Section 5.1 defines the variables used in the Figure.

If two processes are permitted to execute this statement simultaneously, then the first translation will terminate with $B = 1$, while the second will terminate with either $B=1$ or $B=2$ (assuming B is initially 0). If the variable B is supposed to take on values 0 or 1 only, then CTPLAN will produce the plan:

```
[  [NonAtomic, Process1, 1],    [NonAtomic, Process2, 1],
   [NonAtomic, Process1, 2a],   [NonAtomic, Process2, 2a]]
```

If the first translation is used, CTPLAN will not uncover the flaw. In fact, this flaw will not exist if the underlying architecture of the machine atomically executes conditional statements. Since the second translation involves more statements, flaw detection in this environment will take longer. It is therefore more efficient to begin examining algorithms at the coarsest granularity, and then increasing the level of granularity to match that of the underlying machine architecture or software support. However, examining context dependent algorithms beginning with the finest possible granularity can be used in determining the characteristics of architectures under which they will execute correctly.

5 Implementation

As mentioned earlier, CTPLAN is based upon TPLAN. Throughout this section, major differences between the execution methodology used for the two systems are noted. CTPLAN, like TPLAN, has been implemented in Prolog. There are four types of rules:

- **Algorithm:** These rules embody the CTPLAN translation of the algorithm to be examined.
- **Input:** These rules define the flaw to be examined, and the initial state of the system (including the number of processes executing).
- **Architecture:** These rules define the CTPLAN state; for example, the variable symbol table.
- **Planning:** These rules are used to examine the algorithm and plan to reach the goal state containing the desired flaw.

In addition to producing test plans, CTPLAN may be used to simulate the execution of a series of statements and to test the validity of a sequence of statements. When provided with an initial state and a test sequence of statements, CTPLAN will produce the state that will result if they are executed, provided that the test sequence is valid.

if *Cond* **then goto** *label*
if *Cond* **then** *Statement*
goto *label*
variable = *expression*
label
P(*semaphore***)**
V(*semaphore***)**
func Test-and-Set(var *flag***): bool**
Swap (*A*, *B*)

Figure 4: CTPLAN Language

```
<while Cond do Si od; Sj>

 becomes:
    W  ::if Cond then goto Si'
    W  ::if not Cond then goto Sj'
    Si':: <label>
    Si :: ...
    Si1:: goto W
    Sj':: <label>
    Sj :: ...
```

Figure 5: Encoding While-loops

if $B{=}{=}0$ then $B = B + 1$ else *skip* fi	
Atomic	Non-Atomic
Atomic::1a:: **if** $B{=}{=}0$ **then** $B = B + 1$ Atomic::1b:: **if** $B!{=}0$ **then** *skip*	NonAtomic::1::*Cond* = $(B{=}{=}0)$; NonAtomic::2a::**if** *Cond* **then** $B = B + 1$; NonAtomic::2b::**if not** *Cond* **then** *skip* ;

Figure 6: Two translations of a conditional statement

```
changes(incorrectPc1, 1, variable, flag1) .
changes(incorrectPc1, 1, pc, 1) .
prestatement(incorrectPc1, 1, 0) .

statement(incorrectPc1, Id, 1, state(Semaphores, Progvars, Progcounters),
 state(Semaphores, NewProgvars, NewProgcounters)) :-
lookupSymtab(Progcounters, [incorrectPc1, Id], 0),
        updateSymtab(Progvars, [flag1, true], NewProgvars),
        updateSymtab(Progcounters, [[incorrectPc1, Id], 1], NewProgcounters) .
```

Figure 7: Statement in process IncorrectP1::1:: flag1 = true

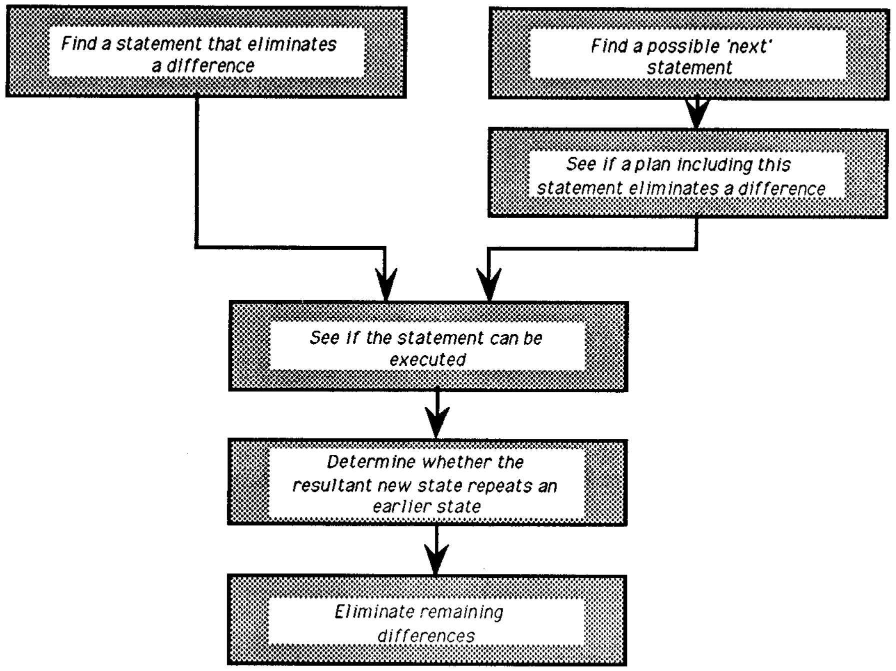
Find a statement that eliminates a difference
Find a possible 'next' statement
See if a plan including this statement eliminates a difference
See if the statement can be executed
Determine whether the resultant new state repeats an earlier state
Eliminate remaining differences

Figure 8: Finding a plan step

5.1 Algorithm Rules and Input Rules

CTPLAN's Algorithm Rules are used to encode the algorithms to be tested. Figure 7 gives an example of the rules that must be defined for each algorithm statement. The rules have the form shown in Figure 9.

Input Rules define the initial and final state of the system. Each state describes the following: the state of the system variables and semaphores, the processes that will exit, and the code each process executes. CTPLAN's purpose is to determine the sequence of statements that will transform the initial state into the final state. Figure 10 describes an initial and a final state that CTPLAN will use to expose a flaw in an incorrect version of readers/writers (the 12 shown in both plans is the test case number of the input). The two semaphores, mutex and wrt, are shown to have the initial value of 1; global variable readcount has value 0. There are three processes: [writePc, 1], [readPc, 1], [readPc, 2]. In the initial state, the program counter for each of these processes is set to 0. In the final state, all values are set to dontcare, with the exception of the program counters. If a variable has value dontcare, then CTPLAN will modify it as needed.

5.2 Architecture Rules

CTPLAN algorithms are made up of ordered statements rather than TPLAN's operation specifications. TPLAN's architecture rules are more complex than CTPLAN's rules, since TPLAN's architecture describes a machine architecture consisting of heterogenous components, and CTPLAN's architecture describes a runtime state consisting of a collection of homogeneous symbol tables.

The current implementation of CTPLAN does not explicitly encode a scheduler, and does not embody more complex structures such as monitors.

CTPLAN maintains a state having four components: a symbol table of semaphores, a symbol table of variables (local and global), a table of program counters for the various processes that will be executing the algorithms, and a history of all past states.

Note that Pc points to the statement just executed rather than the next statement to be executed. This is advantageous when jumping to a loop, or executing semaphores or other potentially blocking statements, since it is clear whether or not an attempt has been made to execute the statement. Particularly for blocking statements, it will permit CTPLAN to first select statements that have *not* yet been attempted, rather than those which have been tried, since these are most changes(*Algorithm-name*, *Process-identifier*, *Type-of-state-object*, *Name-of-state-object*)

prestatement(*Algorithm-name*, *Current-statement-number*, *Previous-statement-number*)

statement(*Algorithm-name*, *Process-identifier*, *Current-statement-number*,
Incoming-state, *Outgoing-state*)
where

Algorithm-name	the name of the algorithm
Current-statement-number	the step in the algorithm
Type-of-state-object	statement changes this object type: variable, semaphore, pc
Name-of-state-object	the actual name of the object changed
Previous-statement-number	possible preceeding statements
Process-identifier	actual process executing the statement
Incoming-state	state before execution
Outgoing-state	state after execution

Figure 9: Form of algorithm rules

```
initState(12, state(symtab([[mutex, 1], [wrt, 1]]),
   symtab([[readcount, 0], [inCS1, false], [inCS2, false]]),
   symtab([[[readPc, 1], 0], [[writePc, 1], 0],
[[readPc, 2], 0]]))) .

finalState(12, state(symtab([[mutex, dontcare], [wrt, dontcare]]),
    symtab([[readcount, dontcare],
    [inCS1, dontcare], [inCS2, dontcare]]),
    symtab([[[readPc, 1], 9], [[writePc, 1], 3],
[[readPc, 2], 9]]))) .
```

Figure 10: Input for multiple readers/writers

likely to succeed.

5.3 Planning Rules

CTPLAN's planning rules govern the way in which tests that expose algorithm flaws are found. Figure 8 shows how each new plan step is chosen:

1. CTPLAN searches the Algorithm Rules to find a statement within a process that can either immediately eliminate a difference, or, if it cannot, can potentially lead to a statement that can eliminate a difference (these are found by backtracking through the algorithm execution steps).
2. CTPLAN next checks to see if the statement can be executed through to completion (recall that certain statements, such as `P(semaphore)` may block).
3. CTPLAN then looks to see if execution of the statement would duplicate a previous system state exactly. Since CTPLAN permits looping, this is necessary to eliminate infinite attempts to execute the same series of statements. It will also produce shorter test plans than if states are permitted to repeat. This step is also necessary to detect the possiblity of livelock and starvation.
4. Steps 1-3 are then repeated, until all differences have been eliminated.

CTPLAN searches for 'forward differences' rather than 'backward differences', as is the case in TPLAN. In general, one has more information about the starting state of an algorithm than about the final state of an algorithm. Most concurrent algorithms have definite specifications about the starting state of the variables. For example, the values of the semaphores in the readers/writers solution are specified (Figure 1). This information is not readily available for the final state, since it is often the incorrect usage of these semaphores and global variables that result in the flaw that is to be detected.

6 Ongoing Work

One important collection of improvements to CTPLAN involves modifying the criteria that determine when a particular statement is considered for inclusion in a plan. In CTPLAN's first planning step, preference is given to selection of a statement that can immediately eliminate a difference between the current state and the goal state. In effect, a metric exists by which preference is given based on reduction of distance from the goal state. More refined metrics might give preference to one statement over another by other criteria. For example, it might be advantageous to give preference to potentially blocking operations when they can be executed, since the operations might not be eligible for execution at a later stage. The search for refined metrics is one type of improvement in the heuristics of CTPLAN being attempted. In conjunction with this goal, new versions of CTPLAN will support experimentation to improve search performance by permitting on-the-fly modification of metrics, and will explore the use of combined forward and backward searches to determine plan steps.

A second impreovement to the heuristics of CTPLAN would allow one to search for shorter plans first by temporarily suspending the search from an initial plan segment when it becomes 'long,' in favor of exploring other initial plan segments. This is motivated by the conjecture that an execution sequence corresponding to a shorter plan has a higher probability of actually occurring, thus permitting the developer to concentrate on removing the most probably execution errors first.

Another important collection of improvements to CTPLAN involves examining intermediate members of the sequence of states defined by the steps in the test plans CTPLAN generates. This would permit easy identification of loop sequences that have undesirable characteristics besides exact repetition of a previous state; for example, one might observe that a particular process is not executed, even though other changes have been made in the system state. This technique may be used to restrict search, and will help in detection of unfairness. It would also permit the user to specify that CTPLAN achieve its goal plan without passing through a state satisfying some intermediate specification. For livelock detection it is necessary to improve user control over what is considered progress, rather than simply defining progress as 'reduced differences between states.'

There are several other improvements currently under development. At present, the user must translate algorithms by hand from their original language into CTPLAN's target language. This tedious process will be replaced by automatic translation from different high-level programming languages into CTPLAN's language, and give the programmer control over the granularity of the translation. In support of this goal, CTPLAN will be augmented to handle additional high-level programming language constructs, such as monitors. Such structures will involve adding scheduling rules to CTPLAN, and these rules would help eliminate some plans. Deadlock detection will be improved by augmenting CTPLAN so that it reports to the user when it reaches an intermediate state such that no further steps (or useful steps) can be taken. The search for flaws can be improved by omitting blocks of code that are shown by syntactic analysis to be noninterfering. Semantic analysis is required to show that other blocks are noninterfering; CTPLAN can be used to show such noninterference.

CTPLAN currently requires the user to specify a goal state for each program being analyzed. Generic specifications that characterize flaws will be developed to reduce the user's effort.

References

[Bar85] H. Barringer. *A Survey of Verification Techniques for Parallel Programs.* Springer-Verlag, 1985.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[Dij65a] E. Dijkstra. Cooperating sequential processes. Technical report, Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[Dij65b] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[FLA90] D. A. Frincke, K. Levitt, and M. Archer. A planning system for the intelligent testing of software. *Fifth Annual Knowledge-Based Software Assistant Conference*, Sept 24-28 1990.

[FS81] L. Flon and N. Suzuki. The total correctness of parallel programs. *SIAM J. Computing*, 10(2):227–246, May 1981.

[HKMC90] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. *Supercomputing*, 1990.

[Hym66] H. Hyman. Comments on a problem in concurrent programming control. *Communications of the ACM*, 9(1):45, January 1966.

[MH89] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, pages 593–623, December 1989.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, 1980.

[OFT81] L. J. Osterweil, L. D. Fosdick, and R. N. Taylor. Error and anomaly diagnosis through dataflow analysis. *Proceedings of Summer School on Computer Program Testing*, pages 35–63, 1981.

[PS87] J. L. Peterson and I. Silberschatz. *Operating systems concepts.* Addison-Wesley Publishing Company, 1987.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, 1986.

[Sev87] R. E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, pages 20–32, May 1987.

[Tay83] R. N. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[TFC89] J. J. P. Tsai, K-Y Fang, and H-Y Chen. Debugger for concurrent programs. *Proceedings 13th Annual International Computer Software and Applications Conference*, September 1989.

[VA86] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. *Proceedings 27th IEEE Symposium on Foundations of Computer Science*, 27:233–243, 1986.