Automatic Discovery of Mutual Exclusion Algorithms[*]

(Preliminary Version)

Yoah Bar-David
P.O.Box 527, Kfar Hes, Israel
yoah@bardavid.com

Gadi Taubenfeld
School of computer science, the Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel tgadi@idc.ac.il

Abstract. We present a methodology for automatic discovery of synchronization algorithms. We built a tool and used it to automatically discover hundreds of new algorithms for the well-known problem of mutual exclusion. The methodology is rather simple and the fact that it is computationally feasible is surprising. Our brute force approach may require (even for short algorithms) the mechanical verification of hundreds of millions of incorrect algorithms before a correct algorithm is found. Although many new interesting algorithms have been found, we think the main contribution of this work is in demonstrating that the approach suggested for automatic discovery of (correct) synchronization algorithms is feasible.

1 Introduction

1.1 Automatic discovery of correct algorithms

Finding a new algorithmic solution for a given problem is considered as an art. Techniques have been suggested to help with this process, but the core activity relies on human invention and ingenuity. “The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music” [Knu73].

We propose a methodology for automatic discovery of synchronization algorithms for finite-state systems, and demonstrate its feasibility by building a tool that is used to automatically find hundreds of new correct algorithms for the well-known problem of mutual exclusion. The methodology is rather simple, the fact that it is computationally feasible is surprising. It works as follows: assume that you want to solve a specific problem *P*.

1. Write a model-checker *M* for *P*. That is, write a program (or use an existing one) that for any proposed algorithm (solution) *A*, decides whether *A* solves *P*.
2. For a given (restricted) programming language, write a program that will produce syntactically all possible (correct and incorrect) algorithms in that language, under certain user-defined parameters, such as: number of lines of code; number of processes; number, type and size of shared variables; etc.
3. For each algorithm *A* generated in step 2, check if *A* solves *P* (using *M*).

[*] Part of this work was done while the authors were with the Open University of Israel.

We name this methodology *automatic discovery*. The reason that automatic discovery has not been implemented before for synchronization algorithms is probably due to the fact that (automatic) verification is considered to be a time-consuming process, while our brute force approach may require (even for very short algorithms) to try to verify hundreds of millions of incorrect algorithms before finding a correct one. Although the many new algorithms that the tool has found are rather interesting, we think that the main contribution of this work is in demonstrating that the approach suggested for automatic discovery of (correct) synchronization algorithms is feasible.

An important related area of research, which is discussed in Subsection 1.6, is the synthesis of concurrent systems. We observe that unlike in synthesis, our brute force approach heavily depends on model checking, it lets us specify directly: the number of shared objects their type and size, the number of lines of code in a solution, and the programming language used. It also lets us find all solutions in a given *algorithm space* (i.e., the algorithms generated for a given set of parameters) or prove that no solution exists, which in turn enables to find (what is) the shortest solution possible.

1.2 System architecture

The architecture of the tool is shown schematically in Figure 1. Via a user-interface, the user can set the problem parameters: (1) number of processes (2) number of lines of code (3) number, size and type of variables (4) type of (*if* and *while* statement) conditions. The parameters are sent to the algorithm generator, which generates all the possible algorithms according to the given parameters. Each algorithm (which passes the optimization checks) is sent to verification. If an algorithm is verified as correct, it is sent back to the user-interface and displayed. Verification results are also returned to the algorithm generator for use in optimizations. A tool based on this system architecture has been implemented in Java and C++, and has around 10,000 lines of code.

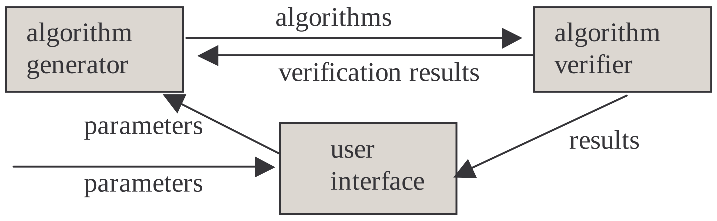

Fig. 1. System architecture for a single computer

1.3 Optimizations and performance

One of the main challenges in building the tool was to be able to process enough algorithms in a reasonable time, so that interesting results can actually be found. To achieve this, many optimizations were implemented (Section 5). As an example for the importance of optimizations, consider the case of mutual exclusion algorithms for two processes with the following parameters: 3 shared bits, complex conditions[1] are allowed, 4 entry commands and 1 exit command. Even for this setting the *algorithm*

[1] Complex conditions are composed of two simple conditions (terms) related by *and*, *or* or *xor*.

space is huge. There are about 10^{21} possible algorithms in the (high-level) generation language[2] (and there are about 10^{48} algorithms in the low-level verification language). Using all optimizations, less than $3 \cdot 10^7$ algorithms were actually generated and tested, requiring about 25 minutes on a Pentium-4/1.6Ghz PC. From all the algorithms in this reduced algorithm space (of size $3 \cdot 10^7$), 105 correct algorithms were found, one of which is the famous Peterson's algorithm (Section 2.6) [Pet81].

While the optimizations reduce the number of algorithms needed to be verified, we apply several other techniques, which dramatically improve the performance. For example, during verification, for each tested algorithm, there is a need to construct a structure called a state transition graph. To improve performance, we do not build such a graph from scratch every time, but rather use a sub-graph of the graph already built for the previously tested algorithm as a basis for building the new graph.

1.4 The model and the mutual exclusion problem

We make the following assumptions. All processes run the same algorithm[3], but processes have unique integer identifiers. In the case of two processes their ids are 0 and 1. An event is either an atomic read or atomic write of a single shared variable. A shared variable can be read by all processes, and written by a single process or by multiple processes, depending on user-defined parameters (single-writer vs. multi-writer). Initially, all shared variables are set to zero.

The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the *remainder, entry, critical section* and *exit*. The mutual exclusion problem is to write the entry and the exit code in such a way that the following two basic requirements are satisfied:

1. *Mutual Exclusion*: no two processes are in their critical section at the same time.
2. *Deadlock-freedom*: if a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.

We will examine also solutions that satisfy the following stronger requirement:

3. *Starvation-freedom*: if a process is trying to enter its critical section, then this process must eventually enter its critical section.

The following measures are used when discussing algorithms: (1) number and size of variables, (2) number of commands, (3) whether the *if* and *while* conditions are simple or complex (i.e., one term or two terms), (4) whether starvation-freedom is satisfied or just deadlock-freedom. These measures are used *only* w.r.t. the specific language we

[2] In the high-level generation language, for the given parameters, the number of assignment statements is 53 and the number of conditions (simple, and complex) is 8081. Thus, each of the 5 lines of code is chosen from: 53 assignments, 8081 *if* statements, 8081 *while* statements, and 3 closing (*endif, else, endwhile*).

[3] That is, the algorithm of process *i* can be obtained from that of process *j* by swapping their ids. This assumption helps in reducing the size of the "algorithm space" and does not affect computability.

use (as defined in Section 3). In some other language one may be able to write a statement like *a*=*b* [*c* [*d*]] and count it as a single line of code and 4 variables, where in our model, this line must be written as: *t*=*c*[*d*] followed by *a*=*b*[*t*] which requires 2 lines of code and 5 variables. In the sequel, the terms *entry commands* and *exit commands* mean language instructions used for the entry and exit sections, respectively.

1.5 New algorithms for mutual exclusion

The system was run to find algorithms for *two* processes using 2, 3, 4, 5 or 6 shared bits. A methodical search was conducted to find the shortest solutions: the number of commands was incrementally increased until a solution was found. The lengths of the *shortest* solutions are summarized in the table below. The number of tested algorithms is displayed, along with the number of correct algorithms found. As a result of optimizations, not all generated algorithms were actually tested. The number of tested algorithms counts only those that were actually tested by the algorithm verifier. All correct algorithms that have been found are new (and shorter than previously known solutions), except for Peterson's algorithm. In the starvation-freedom column, "yes" means that *some* of the algorithms satisfy starvation-freedom (and all satisfy deadlock-freedom). All the tests were performed on a Pentium 4/1.6 GHz PC.

User-defined parameters				Results			
Shared bits	Entry comm-ands	Exit comm-ands	Complex condi-tions	Starva-tion freedom	Tested algorithms	Correct algo-rithms	appx. running hours
2	6	1	Yes[4]		7,196,536,269	0	216
2	7	1			846,712,059	66	39
3	4	1	Yes	Yes	25,221,389	105	0.4
3	6	1		Yes	1,838,128,995	10	47
4	4	1	Yes	Yes[5]	129,542,873	480	1
4	5	1			129,190,403	56	1
4	6	1		Yes	*900,000,000	80	12
5	5	1			*22,000,000	106	0.4
6	5	1			*70,000,000	96	1

In Section 2, various conclusions are drawn from these results. Two of them are: (1) when only *simple* conditions are allowed, the shortest algorithm found has 5 entry and 1 exit commands, and it uses 4 bits (with just 5 entry commands, out of which one has to be a *while* command and one *endwhile*, the algorithm can not assign values to too many variables); (2) when *complex* conditions are allowed, Peterson's algorithm (Section 2.6), which has 4 entry and 1 exit command has been rediscovered by the system and proved to be the shortest possible. Many new solutions with the same parameters have also been found. Burns and Lynch had proved that any deadlock-free mutual ex-

[4] Performed in parallel on 5 computers.

[5] In this setting (4 bits, 4 entry, 1 exit, complex), 4 out of the 480 correct algorithms use only single-writer bits. None of these 4 algorithms satisfies starvation-freedom.

* This run was stopped after a few solutions were found. Not all possible algorithms were tested

clusion algorithm for *n* processes must use at least *n* variables [BL80,BL93], and hence there is no need to search for solutions for two processes using one variable.

There is a simple method, called tournament, which enables to construct a mutual exclusion algorithm for *n* processes from any algorithm for two processes [PF77]. Thus any of the new algorithms for two processes corresponds to a new algorithm for many processes. In a tournament algorithm, the processes are divided into two groups, and in each group the processes compete recursively. The two "winners" use the solution for two processes to determine which is the one to enter its critical section.

1.6 Related Work

The mutual exclusion problem was first presented by Dijkstra in [Dij65]. Numerous mutual exclusion algorithms have been published, some are discussed later in the paper. For a survey of some early algorithms for mutual exclusion see [Ray86].

Model checkers have been successfully implemented and used in practice to verify (among other things) communication protocols and circuit designs. For a general introduction on formal methods, which also includes a short survey of existing model checkers and few notable examples of how they have been used in the industry to aid in the verification of newly developed designs, see [CW96]. A comprehensive presentation of the theory and practice of model checking can be found in [CGP00].

An important related area of research is the synthesis of concurrent systems. In system synthesis, a desired specification is transformed into a system that is guaranteed to satisfy the specification. Methods for synthesizing concurrent programs are usually based on extracting a system that meets the specification from a constructive proof that the specification is satisfiable [EC82,MW80,MW84]. A method of synthesizing *k similar* sequential processes (which, as the authors write, can be automated) is presented in [AE98]. There are two possible approaches for synthesis of concurrent systems: (1) to use a synthesis procedure for a single process, and then decompose the processes [EC82, MW84], and (2) to construct the underlying processes directly [PR90]. We have mentioned only very few of the numerous papers that have been published on this subject. We conclude with a quote from [KV01]: "the real challenge that synthesis algorithms and tools face in the coming years is mostly not that of dealing with computational complexity, but rather of making automatically synthesized systems more practically useful''.

In [PS00], an automatic protocol generation approach, which is similar to our approach, is used to generate security protocols.

2 Tests and Results

The system was run to find mutual exclusion algorithms for two processes using 2, 3, 4, 5 or 6 shared bits. One should keep in mind that: (1) all generated algorithms adhere to the high-level generation language as defined in Section 3. The algorithms may be shorter when expressed in a different language; (2) It is relatively easy to prove that an algorithm discovered by the system is correct, but to formally prove that no other solution exists, requires proving that the entire system is correct. As already mentioned, it is assumed that initially, all shared variables are set to zero.

2.1 Two shared bits, simple conditions

A known solution, for *n* processes, is the one-bit algorithm [BL80,BL93,Lam86], which does not satisfy starvation-freedom. When porting it to our high-level language, and trimming it for just two processes, we get a short version (using two bits and simple conditions) with 9 entry commands and 1 exit command. Our system has found a shorter solution. The following table summarizes the results.

Conclusion *The shortest solution found for two bits and simple conditions has 7 entry and 1 exit commands.*

Comment: All the 66 algorithms do not satisfy starvation-freedom. Can it be proved analytically that for this case all correct algorithm satisfy only deadlock-freedom?

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct algorithms
4	1	27,372	0
4	2	44,340	0
5	1	925,389	0
5	2	1,235,778	0
6	1	28,522,988	0
7	1	846,712,059	66

Following are 3 sample solutions (for two processes). The identifiers are 0 and 1. Thus, me is in {0,1}, and next = 1-me.

2 bits, simple conditions, single-writer, deadlock-free		
1 a[me] = 1	1 a[me] = 1	1 a[me] = me
2 while a[next] = 1	2 while a[next] = 1	2 while a[0] = me
3 a[me] = next	3 while a[0] = me	3 a[me] = 1
4 while a[0] = me	4 a[me] = 0	4 while a[1] = next
5 endwhile	5 endwhile	5 a[me] = 0
6 a[me] = 1	6 a[me] = 1	6 endwhile
7 endwhile	7 endwhile	7 endwhile
8 critical section	8 critical section	8 critical section
9 a[me] = 0	9 a[me] = 0	9 a[me] = 0

Notice that in the third solution, process 0 does not initially set its flag bit to 1.

2.2 Three shared bits, simple conditions

A known solution for this setting is Dekker's starvation-free algorithm [Dij65]. When porting it to our language we get a version with 9 entry and 2 exit commands. Our system has found shorter solutions. The table summarizes the results.

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct algorithms
4	1	287,579	0
4	2	493,073	0
5	1	24,124,934	0
5	2	36,636,722	0
6	1	1,838,128,995	8 starvation-free 2 deadlock-free

Conclusion *The shortest solution found for three bits and simple conditions has 6 entry and 1 exit commands.*

Comment: Every starvation-free solution contains an “if” statement, while no deadlock-free solution contains an “if” statement.

3 bits, simple conditions starvation-free	3 bits, simple conditions deadlock-free
1 a[me] = next	1 a[me] = 1
2 if b = next	2 while b != a[next]
3 b = me	3 while a[b] = me
4 while b = a[next]	4 b = a[0]
5 endwhile	5 endwhile
6 endif	6 endwhile
7 critical section	7 critical section
8 a[me] = me	8 a[me] = 0

2.3 Four shared bits, simple conditions

The table summarizes the results.

Conclusion *The shortest deadlock free solution found for four bits and simple conditions has 5 entry and 1 exit commands.*

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct algorithms
4	1	915,350	0
4	2	1,270,897	0
5	1	129,190,403	56 deadlock-free
6	1	[*]900,000,000	80 starvation-free

Comment: None of the 56 deadlock-free algorithms satisfy starvation-freedom and none of them use only single-writer variables.

Conclusion *The shortest starvation-free solution found for four bits and simple conditions has 6 entry and 1 exit commands.*

Since no solution with 5 entry and 1 exit commands satisfies starvation-freedom, the system was run to check if a starvation-free solution exists when using more lines of code. When looking for solutions with 6 entry commands and 1 exit command, 80 solutions that satisfy starvation-freedom were found, and the execution was stopped before all possible algorithms were tested.

4 bits, simple conditions, deadlock-free	4 bits, simple conditions, starvation-free
1 a[me] = 1	1 a[me] = 1
2 b[0] = 0	2 b[me] = me
3 while b[me] != a[next]	3 a[1] = 0
4 b[next] = me	4 a[me] = a[0]
5 endwhile	5 while b[next] != a[1]
6 critical section	6 endwhile
7 a[me] = 0	7 critical section
	8 b[me] = next

[*] This run was stopped after a few solutions were found. Not all algorithms were tested.

2.4 Five and six shared bits, simple conditions

Most of the solutions are variants of the solution using four shared bits.

Conclusion *The shortest solution found for 5 and 6 bits and simple conditions has 5 entry and 1 exit commands.*

Conclusion *With simple conditions, having more than 4 bits does not help in finding a shorter algorithm.*

Parameters			Results	
# of bits	Entry comm.	Exit comm.	Tested algorithms	Correct alg.
5	4	1	6,987,284	0
5	4	2	10,729,817	0
5	5	1	*22,000,000	106
6	4	1	6,642,480	0
6	4	2	9,365,248	0
6	5	1	*70,000,000	96

2.5 Two shared bits, complex conditions

The last run, with 6 entry and 1 exit command, no solution was found. Solutions with 7 entry and 1 exit command using simple conditions are given in Section 2.1

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct alg.
4	1	1,264,266	0
5	1	69,400,411	0
6	1	7,196,536,269	0

Conclusion *Using complex conditions does not help in finding a shorter solution when two bits are used.*

2.6 Three shared bits, complex conditions

For this setting, a known solution is Peterson's algorithm [Pet81], shown on the right. An immediate question is whether it is the shortest possible. The table summarizes the results.

Peterson's algorithm for 2 processes

```
1  a[me] = 1
2  turn = next
3  while a[next] = 1 and turn = next
4  endwhile
5  critical section
6  a[me] = 0
```

Conclusion Peterson's algorithm is the shortest algorithm found!

Comment: All the 105 algorithms satisfy starvation-freedom and all are variants of Peterson's algorithm. One of them is Peterson's algorithm.

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct alg.
3	1	75,496	0
3	2	492,707	0
4	1	25,221,389	105 (all s.f.)

Conclusion *To get a weaker solution that (only) satisfies deadlock freedom but not starvation-freedom, one more bit or one more command is required.*

* This run was stopped after a few solutions were found. Not all algorithms were tested.

Define the contention-free (time) complexity (CFC), as the number times a process has to access shared variables in its entry code when it runs alone [AT96].

Comment: The CFC of each of the 105 starvation-free algorithms found is at least 3.

Comment: There is a deadlock-free solution where the contention-free (time) complexity is 2, which uses two bits, 7 entry and 1 exit commands (see Section 2.1).

Following are 3 examples (out of 105) of additional solutions found:

3 shared bits, complex conditions, starvation-free		
1 a[me] = 1	1 a[me] = 1	1 a[me] = 1
2 b = next	2 b = me	2 b = next
3 while a[1] = a[0] and	3 while a[next] = me	3 while me = 0 xor
b = next	xor b = 0	b != a[b]
4 endwhile	4 endwhile	4 endwhile
5 critical section	5 critical section	5 critical section
6 a[me] = 0	6 a[me] = 0	6 a[me] = 0

2.7 Four shared bits, complex conditions

For this setting there exists a variant of Peterson's algorithm, due to Kessels [Kes82], which uses only single-writer bits. Kessels' algorithm is not symmetric, and it uses "minus" (or negation). Porting the algorithm to our language would result an algorithm with over 15 entry commands. The following table summarizes the results.

Comment: Some of the 480 algorithms satisfy starvation-freedom, but none of these starvation-free algorithms use only single-writer bits.

Comment: 4 of the 480 algorithms use only single-writer bits.

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct alg-s
3	1	118,536	0
3	2	1,088,071	0
4	1	129,542,873	480

Conclusion *Using 4 bits instead of just 3 does not help in finding a shorter solution.*

Following are two solutions that were found:

4 shared bits, complex conditions, deadlock-free, single-writer	
1 a[me] = 1	1 a[me] = 1
2 while b[me] = me or b[1] != a[next]	2 while b[0] != a[next] or b[me] = next
3 b[me] = a[0]	3 b[me] = a[1]
4 endwhile	4 endwhile
5 critical section	5 critical section
6 a[me] = 0	6 a[me] = 0

3 Algorithm Generator

3.1 Generating all possible algorithms

Initially, an attempt has been made to generate all possible algorithms using the verification assembler-like language (see Section 4). This approach has two main disadvantages: (1) the number of generated algorithms is too large, because there are many ways one can write the same high-level language algorithm in a low-level language, and (2) it takes a long time to manually reverse-engineer an algorithm written in assembler into a high-level language to make it easier to understand. To overcome these problems, a simple high-level language was defined. This language was designed to be simple, so that the number of possible options for each command will be relatively small, but powerful enough so that with just a few lines of code, a solution can be written. The exact language specification is given in the next subsection.

Before running the tool, the following parameters must be specified (by the user):
1. The number of processes and the number of shared variables.
2. The number of entry and exit commands. A command is a single line of code, and the language defines exactly how statements are related to lines of code.
3. The maximal value for each shared variable. The minimum value is zero.
4. Are complex conditions allowed, and if so, with which relations (and, or, xor).
5. Are multi-writer variables allowed (or just single-write variables).

With these parameters set, the system enumerates all possible variables, assignments, conditions, and control structures. A control structure is the assignment of a command type to a line of code (e.g. line 1 is an assignment, line 2 is a while statement). Each algorithm is enumerated by the control-structure number, and for each line, the operand number for that line (which assignment or which condition). Once all language elements are enumerated, the first possible algorithm is generated. After verification, the last line of code is incremented to return the next possible command for this line, and so on. The generator compiles each algorithm into the verification language.

3.2 Language used to generate algorithms

The following table specifies the (high-level) language used. The line breaking within statements must be exactly as below. One of the parameters is the number of lines of code, hence, the language does not allow a single line with multiple statements.

Language elements

Constants	Integers, from zero to the highest allowed value for a variable.
Relative constants	Contain a process number: **me** (my process number, zero for first process), **next** (successor process number), **prev** (predecessor's number).
Simple variables	Integers, can have a value from zero to the highest allowed value for a variable.
Arrays	One-dimensional array of simple variables. The array size is as the number of processes.
Referencable variable	Either a simple variable, or an array variable with an index. The index can be a constant, a relative constant or a simple variable.
Simple conditions	A comparison between 2 variables or a variable and a constant. Comparison operators are = (equals) and != (not equal)
Complex conditions	2 simple conditions, related with **and**, **or** or **xor**

Statements

Assignment statement	*referencable-variable* = *constant* (e.g. v=0 or a[1] = 0) *referencable-variable* = *referencable-variable* (e.g. v=q or a[1]=b[v])			
if and while statements	**if** *condition* *statement(s)* **endif**	**if** *condition* *statement(s)* **else** *statement(s)* **endif**	**while** *condition* *statement(s)* **endwhile**	**while** *condi-tion* **endwhile**

4 Algorithm Verifier

Model checking is a technique for mechanically verifying finite state concurrent systems. The main challenge in model checking is to deal with the state space explosion problem. We have developed a special very fast verifier which is limited to verify only the correctness of mutual exclusion algorithms.

4.1 Verification language

To ensure that the verification is according to the model, a verification language was designed. This language is a low-level (assembler like) language, and contains only instructions that can be run atomically. Each instruction is represented by a triplet: an operation code, and 2 optional operands. Each process has 3 local variables that can be read or written only by the single process. These variables are: a program counter, a general purpose register and an index register. For lack of space, the complete instruction set of the verification language is omitted from this version.

4.2 Building the state transition graph

A *state* is the snapshot of all information of the system at a single point of time. In our implementation, state information is the contents of all global memory variables, and the contents of all local variables for each process (including program counters).

When a single verification language instruction is executed, the system moves to a new state. To build the entire state transition graph, the system starts with the initial state (i.e., all memory set to zero, and all program counters are before the first instruction) and iteratively adds states that are reachable, until all reachable states are added. Once this procedure is completed, the system has mapped all the reachable states and all possible state transitions, and can run validation checks on them. During the construction, various techniques are applied to reduce the size of the graph.

A key optimization used is the following: when an algorithm is verified, instead of generating the entire graph again, the verifier keeps the sub-graph that was generated for the previous algorithm and is still valid for the new algorithm. This is implemented as follows: comparing the two algorithms, the verifier finds the top-most verification command that was changed, and trims from the state transition graph all states and transitions that may be affected by the algorithm change, but keeps all the states that are not affected.

4.3 Checking if an algorithm satisfies the correctness properties

To be a valid solution, an algorithm must ensure mutual exclusion and deadlock-freedom. The test for mutual exclusion is done by looking at each reachable state and checking if two processes are in their critical section at that state. If there is a state where two processes are in their critical section, the algorithm is rejected. This test is actually done when adding each new state.

The test for deadlock-freedom is done by looking at all cycles in the state transition graph. A cycle represents an infinite loop. If there is a cycle where all active processes executed at least one instruction, and no process is in its critical section at any state of the cycle, then the algorithm does not satisfy deadlock-freedom, and it is rejected.

The test for starvation-freedom is also done by looking at all cycles in the graph. If there is a cycle where all active processes executed at least one instruction, and some process was not in its critical section during any state of the cycle, then the algorithm does not satisfy starvation-freedom, as the loop can be executed forever, and that process will be starved. If an algorithm satisfies mutual exclusion and deadlock-freedom but not starvation-freedom, it is not rejected. As done in other model checkers, Tarjan's algorithm is used for finding strongly connected components [Tar83].

5 Optimizations

One of the main challenges was to be able to process enough algorithms in a reasonable time, so that interesting results can actually be found. To achieve this, many optimizations were implemented. Following are the main optimizations used.

5.1 Optimizations during algorithm generation

1. *Do not generate syntactically incorrect algorithms*: An algorithm is syntactically correct if every open block (*if* or *while*) is properly closed (by *endif* or *endwhile*), and if each *if* command has no more than one *else* section. When the algorithm is not syntactically correct, the entire program structure is rejected, without generating all options for each condition or assignment.
2. *Do not generate equivalent conditions*: Equivalent conditions are generated and tested only once. For example, a=1 is equivalent to 1=a; and a=1 is equivalent to a!=0, when the variable is a single bit.[6]
3. *Do not generate constant conditions*: Some conditions are always true or always false. There is no point in generating them. An example is: (a=0 and a=1).
4. *Heuristics*: *do not allow consecutive assignments to the same variable.* 2 consecutive assignments should not have the same target variable.[7] Example, a=1 followed by a=0.

[6] Notice that the following conditions are *not* equivalent: (a=0 and b=0) is not equivalent to (b=0 and a=0). Since only a single read is atomic, the order of reading variables a and b could make a difference.

5. *Do not generate relative constant "prev" for 2 processes*: For only 2 processes relative constant *next* is exactly the same as *prev*, so only one of them is needed.

5.2 Optimizations during verification

6. *Incremental state graph construction*: Instead of building the graph for each algorithm from scratch, the largest valid sub-graph of the graph already built for the previously tested algorithm is used as a basis for building the new graph.
7. *Stop building state graph on first error*: The standard model-checking approach is to build first the entire state transition graph, and then verify all the requirements on this graph. The optimization is that while building the graph, every new state is checked if in it 2 processes are in the critical section. If more than one process is in the critical section, verification fails immediately, without building the rest of the graph. (Also called on-the-fly model checking.)
8. *Minimize number of states*: A new state is generated only if the last executed command accessed a (global) variable. If only local variables are accessed (change the program-counter or local registers) in the last executed command, another command is executed without generating a new state in the graph.

5.3 Interactive optimizations

The following optimizations use information exchanged between the verifier and the generator to skip the generation of some algorithms by the algorithm generator.

9. Sk*ip generation of alternatives options for statements that are not executed*: In some cases, the verifier determines that an algorithm is incorrect without executing all the instructions. For example, the algorithm to the right does not satisfy mutual exclusion. Since command 3 is never executed, no other assignment in line 3 can correct the algorithm. Therefore, there is no need to generate the alternative assignments for line 3.

```
1 a = 0
2 if a = 1
3    b = a
4 endif
5 critical section
6 a = 0
```

10. *Skip generation of alternatives for complex conditions that were not executed*: An optimization similar to the previous one applies to partially executed complex conditions. For example, if the algorithm contains the condition "if (a=1 or b=2)" and if during verification, the value of variable a is always evaluated as 1, then the right part of the condition (b=2) is never executed, and there is no need to generate other alternative conditions for the right part. For an "*and*" relation, the same applies when the first condition is always false.

[7] This restriction should be removed if expressions are added to the language. If the language is changed to contain operators like "+", then the following code should be made valid: a=1 followed by a=a+b.

5.4 Optimizations specific to mutual exclusion

11. *Check "solo" runs first*: We first check that, when there is no contention, a single process (when run alone) eventually enters its critical section. This simple optimization was found very important in improving the overall performance.
12. *Must have "while" in entry code*: In a correct algorithm a process must wait when another process is in its critical section. The only way to busy-wait in our model is by using a while loop. Therefore, an algorithm that does not contain a *while* in the entry code is incorrect and does not need to be verified.
13. *Must have an assignment statement in the entry code and in the exit code.* At least one assignment in the entry code must assign a value other than zero.

6 Discussion

We have implemented a tool that can automatically discover algorithms for the well-known problem of mutual exclusion. Using this tool our experiments have successfully discovered many new algorithms (and re-discovered some known ones). Immediate directions for further research are: to add optimizations and heuristics in order to reduce and search the huge algorithm space; and to apply a similar approach to other synchronization problems. The results mentioned in Section 2 leave some specific questions open: Why some sets of parameters have no starvation-free algorithms (or no single-writer algorithms), while others, not vastly different as defined by the parameters, do? Are there meaningful classes that solutions can be reduced to, which can help to distinguish between reordering of statements versus a radically different structure?

References

[AE98] P. Attie and E. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Tran. on Programming Languages and Systems* 20(1):51-115, 1998.

[AT96] R. Alur and G. Taubenfeld. Contention-free complexity of shared memory algorithms. *Information and Computation* 126(1):62-73,1996.

[BL80] J.N. Burns and N.A. Lynch. Mutual exclusion using indivisible reads and writes. In *18th annual allerton conf. on communication, control and computing,* 833-842, 1980.

[BL93] J.N. Burns and N.A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation,* 107(2):171-184, December 1993.

[CGP00] E.M. Clarke, O. Grumberg and D. Peled. *Model Checking.* The MIT Press, 2000.

[CW96] E.M. Clarke, J.M. Wing, et al. Formal Methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626-643, December 1996.

[EC82] E. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming* 2:241-266, 1982.

[Dij65] E.W. Dijkstra. Solutions of a problem in concurrent programming control. *Communications of the ACM,* 8(9):569, 1965.

[Kes82] J.L.W. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17(2):135-141, June 1982.

[Knu73] D.E. Knuth. *The art of computer programming*. Addison-Wesley, 2nd ed., 1973.

[KV01] O. Kupferman and M. Vardi. Synthesizing distributed systems. *Proc. of the 16th IEEE Symp. on logic in computer science*, 2001.

[Lam86] L. Lamport. The mutual exclusion problem: Part II -- Statement and Solutions. *Journal of the ACM,* 33:327-348, 1986.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems* 2(1):90-121, 1980.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Tran. on Programming Lang. and Systems* 6(1):68-93, 1984.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115-116, 1981

[PF77] G.L. Peterson and M.J. Fischer. Economical solutions for the critical section problem in a distributed system. *Proc. 9th ACM Symp. on Theory of Computing*, 91-97, 1977.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. *Proc. of the 31st FOCS*, 746-757, 1990.

[PS00] A. Perrig and D. Song. Looking for diamonds in the desserts: Automatic security protocol generation for three-party authentication and Key distribution. In *Proc. of the 13th IEEE Computer Security Foundation Workshop,* July 2000.

[Ray86] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallelisme, 1984.

[Tar83] R.E. Tarjan, *Data Structures and Network Algorithms.* Society for industrial and Applied Mathematics, 1983.