Searching for Mutual Exclusion Algorithms using BDDs

Koichi Takahashi and Masami Hagiya

National Institute of Advanced Industrial Science and Technology
and
University of Tokyo
k.takahashi@aist.go.jp
http://staff.aist.go.jp/k.takahashi/

Abstract. The impact of verification technologies would be much greater if they could not only verify existing information systems, but also synthesize or discover new ones. In our previous study, we tried to discover new algorithms that satisfy a given specification, by first defining a space of algorithms, and then checking each algorithm in the space against the specification, using an automatic verifier, i.e., model checker. Needless to say, the most serious problem of this approach is in search space explosion. In this paper, we describe case studies in which we employed symbolic model checking using BDD and searched for synchronization algorithms. By employing symbolic model checking, we could speed up enumeration and verification of algorithms. We also discuss the use of approximation for reducing the search space.

1 Introduction

Verification technologies have been successfully applied to guarantee the correctness of various kinds of information systems, ranging from abstract algorithms to hardware circuits. Among them is that of model checking, which checks the correctness of a state transition system by traversing its state space [2]. The success of model checking is mainly due to its ability to automatically verify a state transition system without human intervention.

However, the impact of verification technologies would be much greater if they could not only verify existing systems, but also synthesize or discover new ones.

In our previous study [6, 7], we tried to discover new algorithms that satisfy a given specification, by first defining a space of algorithms, and then checking each algorithm in the space against the specification, using a verifier, i.e., model checker. Note that this approach is possible only if the employed verifier is fully automatic. By the approach, we discovered new variants of the existing algorithms for concurrent garbage collection, and a new algorithm for mutual exclusion under some restrictions on parallel execution.

Perrig and Song have also taken a similar approach in the field of protocol verification [10]. They enumerated protocols for asynchronous and synchronous mutual authentication, and successfully discovered new ones by checking enumerated protocols using their protocol verifier, Athena [11].

Superoptimization in the field of compiler technologies is very close to the above mentioned approach, though it does not employ a complete verifier [9, 5]. It automatically synthesizes the code generation table of a compiler by enumerating and checking sequences of machine instructions.

Needless to say, the most serious problem of this approach is in search space explosion. The space of algorithms explodes if the size of enumerated algorithms is not carefully bounded. It is therefore vital for the approach to efficiently traverse the space of algorithms, employing various kinds of search heuristics.

In this paper, we investigate the possibility of simultaneously checking all algorithms in the algorithm space by a single execution of the verifier. Algorithms in the space are symbolically represented by a template containing parameters. Each instantiation of the parameters in the template corresponds to a specific algorithm. By symbolically verifying the template, we obtain constraints on the parameters for the template to be correct.

By symbolic verification, it is possible to share computations for verification among different algorithms, because computations with some parameters uninstantiated are shared by algorithms that correspond to instantiations of those parameters.

By symbolically representing a template of algorithms, it is also possible to apply approximation or abstraction [4] on the template. Before or during verification, algorithms can be approximated by another having similar but smaller representation. If abstraction could be applied on symbolic representation, it would greatly reduce the search space.

In this paper, we employ BDDs (binary decision diagrams) for symbolic representation and verification [1, 2]. As a case study, we take the problem of searching for synchronization algorithms for mutual exclusion without using semaphores, which was also taken in our previous study [6, 7].

In our case studies, a template of algorithms is represented by a sequence of pseudo-instructions including boolean parameters. We define two templates. The first template has the original version of Peterson's algorithm as an instance. The second template has both Peterson and Dekker's algorithms as instances. The predicate defining the initial state, that of error states, and the state transition relation are all represented by BDDs.

We checked the safety and liveness of the template using BDDs, and successfully obtained the constraints on the parameters. We compared the time required for verifying a single concrete algorithm — Peterson's algorithm (or Dekker's algorithm) — with that for checking the template, and gained speed-up of more than two hundred times.

Finally, we made an experiment in which we collapsed BDDs with a small Hamming distance. This is a first step towards using approximation and abstraction in the approach.

The rest of the paper is organized as follows. In the next section, we explain the original versions of Peterson and Dekker's algorithms for mutual exclusion, and their safety and liveness properties. In Sections 3 and 4, we describe the results of searching for variants of those algorithms by the above mentioned approach. In Section 5, we report the result of a small experiment in which we tried approximation during search. Section 5 is for conclusion.

2 Mutual Exclusion

The target of these case studies is to synthesize synchronization algorithms for mutual exclusion without using semaphores.

Peterson and Dekker's algorithms realize mutual exclusion among processes without semaphores. Figure 1 shows an instance of Peterson's algorithm for two processes. An instance of Dekker's algorithm is shown in Figure 2.

```
for (;;) {
    // beginning of the entry part
    flags[me] = true;
    turn = you;
    while (flags[you] == true) {
        if (turn != you) break;
    } // end of the entry part

    // the critical section

    // beginning of the finishing part
    flags[me] = false;
    // end of the finishing part

    // the idle part
}
```

Fig. 1. Peterson's algorithm.

In the figures, `me` denotes the number of the process that is executing the code (`1` or `2`), and `you` denotes the number of the other process (`2` or `1`). The entry part realizes mutual execution before entering the critical section, and the finishing part is executed after the critical section. The idle part represents a process-dependent task.

The safety property of mutual exclusion algorithm is:

> Two processes do not simultaneously enter the critical section.

Liveness is:

> There does not exist an execution path (loop) that begins and ends with the same state, at which one process is in its entry part, and satisfies the following conditions.

```
for (;;) {
    // beginning of the entry part
    flags[me] = true;
    while (flags[you] == true) {
       if (turn != me) {
           flags[me] = false;
           while (turn != me) {}
           flags[me] = true;
       }
    } // end of the entry part

    // the critical section

    // beginning of the finishing part
    turn = you;
    flags[me] = false;
    // end of the finishing part

    // the idle part
}
```

Fig. 2. Dekker's algorithm.

- The process stays in the entry part on the execution path, i.e., it does not enter the critical section.
- Both processes execute at least one instruction on the execution path.

The results of searching for variants of these algorithms are described in the next two sections.

3 Search for Variants of Peterson's Algorithm

In this section, we describe the first case study. We make a template which has Peterson's algorithm as an instance, and check the safety and liveness of the template using BDDs.

3.1 Pseudo-code

We represent Peterson's algorithm using pseudo-code consisting of pseudo-instructions. The pseudo-code may refer to three variables, each of which holds a boolean value.

- `FLAG1`: This variable corresponds to `flags[1]` in Figure 1.
- `FLAG2`: This variable corresponds to `flags[2]` in Figure 1.
- `FLAG0`: This variable corresponds to `turn` in Figure 1. `FLAG0=true` means `turn=2`, and `FLAG0=false` means `turn=1`,

Each instruction in the pseudo-code is in one of the following three forms.

```
SET_CLEAR($p$),$L$
IF_WHILE($p$),$L_1$,$L_2$
NOP,$L_1$,$L_2$
```

The operands L, L_1 and L_2 in the instructions denote addresses in the pseudo-code. In `SET_CLEAR(`p`)`,L, the operand L should point to the next address in the pseudo-code.

The operators `SET_CLEAR` and `IF_WHILE` have a three-bit parameter, denoted by p. Each value of the parameter results in a pseudo-instruction as defined in Figures 3 and 4 for each process. In the figures, b denotes `0` or `1`.

The instruction

```
NOP,$L_1$,$L_2$
```

jumps to either L_1 or L_2 nondeterministically.

	Process 1	Process 2
SET_CLEAR(b00),L	goto L	goto L
SET_CLEAR(b01),L	FLAG0 := b; goto L	FLAG0 := not(b); goto L
SET_CLEAR(b10),L	FLAG1 := b; goto L	FLAG2 := b; goto L
SET_CLEAR(b11),L	FLAG2 := b; goto L	FLAG1 := b; goto L

Fig. 3. SET_CLEAR

	Process 1	Process 2
IF_WHILE(b00),L_1,L_2	goto L_1	goto L_1
IF_WHILE(b01),L_1,L_2	IF FLAG0=b then goto L_1 else goto L_2	IF FLAG0=not(b) then goto L_1 else goto L_2
IF_WHILE(b10),L_1,L_2	IF FLAG1=b then goto L_1 else goto L_2	IF FLAG2=b then goto L_1 else goto L_2
IF_WHILE(b11),L_1,L_2	IF FLAG2=b then goto L_1 else goto L_2	IF FLAG1=b then goto L_1 else goto L_2

Fig. 4. IF_WHILE

The original version of Peterson's algorithm for mutual exclusion can be represented by the following pseudo-code.

```
0: SET_CLEAR(110),1
1: SET_CLEAR(101),2
2: IF_WHILE(111),3,4
3: IF_WHILE(001),4,2
4: NOP,5,5
5: SET_CLEAR(010),6
6: NOP,6,0
```

The first column of the pseudo-code denotes the address of each instruction. The fourth instruction, `4: NOP,5,5`, represents the critical section, and the sixth instruction, `6: NOP,6,0`, the part that is specific to each process. Each process is allowed to loop around the sixth instruction.

We then parameterize five instructions in Peterson's algorithm as in Figure 5. The safety and liveness of this parameterized code were verified as described in the next section.

```
0: SET_CLEAR($p_0$),1
1: SET_CLEAR($p_1$),2
2: IF_WHILE($p_2$),3,4
3: IF_WHILE($p_3$),4,2
4: NOP,5,5
5: SET_CLEAR($p_4$),6
6: NOP,6,0
```

Fig. 5. Template 1

3.2 Verification

The initial state of the state transition system is defined as a state that satisfies the following condition.

$\texttt{PC1} = \texttt{PC2} = 0$
$\texttt{FLAG0} = \texttt{FLAG1} = \texttt{FLAG2} = 0$

In the above condition, `PC1` and `PC2` denote the program counter of Process 1 and Process 2, respectively. Let $I(x)$ denote the predicate expressing the condition, where x ranges over states. $I(x)$ holds if and only if x is the initial state.

The safety of the state transition system is defined as unreachability of an error state that satisfies the following condition.

$\texttt{PC1} = \texttt{PC2} = 4$

In an error state, both processes enter their critical section simultaneously. The system is safe unless it reaches an error state from the initial state. Let $E(x)$ denote the predicate expressing the condition.

Let $T(x, y)$ mean that there is a one-step transition from state x to state y, and $T^*(x, y)$ denote the reflexive and transitive closure of $T(x, y)$. The safety of the system is then expressed by the following formula.

$$\neg\exists xy.\ I(x) \wedge T^*(x, y) \wedge E(y)$$

We can describe the liveness for Process 1 of the system as non-existence of an infinite path on which

$$0 \le \texttt{PC1} \le 3$$

is always satisfied though both processes are infinitely executed. The condition, $0 \le \texttt{PC1} \le 3$, means that Process 1 is trying to enter its critical section. $S(x)$ denote the predicate expressing the condition.

We verify the liveness as follows. Let $T_1(x, y)$ denote the one-step transition relation for Process 1. $T_1(x, y)$ holds if and only if state y is obtained by executing Process 1 for one step from x. Similarly, let $T_2(x, y)$ denote the one-step transition relation for Process 2. Note that $T(x, y)$ is equivalent to $T_1(x, y) \vee T_2(x, y)$. We then define the following three predicates.

$$\begin{aligned}
&T_1'(x, y) = T_1(x, y) \wedge S(x)\\
&T_2'(x, y) = T_2(x, y) \wedge S(x)\\
&T'(x, y) = T(x, y) \wedge S(x)
\end{aligned}$$

For any predicate $Z(x)$ on state x, and any binary relation $R(x, y)$ on states x and y, we define the relation, denoted by $Z \circ R$, as follows.

$$(Z \circ R)(x) = \exists y.\ Z(y) \wedge R(x, y)$$

$Z \circ R$ is also a predicate on states.

For verifying the liveness of the system, we compute the following limit of predicates.

$$\begin{aligned}
&S \circ T_1' \circ T'^* \circ T_2' \circ T'^* \circ\\
&T_1' \circ T'^* \circ T_2' \circ T'^* \circ\\
&T_1' \circ T'^* \circ T_2' \circ T'^* \circ \cdots
\end{aligned}$$

This limit always exists, because the sequence of predicates

$$\begin{aligned}
&S \circ T_1' \circ T'^*\\
&S \circ T_1' \circ T'^* \circ T_2'\\
&S \circ T_1' \circ T'^* \circ T_2' \circ T'^*\\
&S \circ T_1' \circ T'^* \circ T_2' \circ T'^* \circ T_1'\\
&S \circ T_1' \circ T'^* \circ T_2' \circ T'^* \circ T_1' \circ T'^*\\
&\cdots
\end{aligned}$$

is monotonically decreasing. For example, we can prove the second predicate is smaller than the first one as follows. $S \circ T'_1 \circ T'^* \circ T'_2 \subseteq S \circ T'_1 \circ T'^* \circ T' \subseteq S \circ T'_1 \circ T'^*$.

Let us denote this limit by S'. It expresses the beginning of an infinite path on which S always holds and both processes are infinitely executed, i.e., a state satisfies the limit if and only if there exists such an infinite path from the state. The liveness is then equivalent to unreachability from the initial state to a state satisfying the limit.

$$\neg \exists xy.\ I(x) \wedge T^*(x, y) \wedge S'(y)$$

The liveness for Process 2 can be symmetrically described as that for Process 1. The whole system satisfies the liveness if it holds for both processes.

Since there are 7 pseudo-instructions in the pseudo-code, the program counter of each process can be represented by three bits. Therefore, a state in the state transition system can be represented by nine boolean variables; three are used to represent the program counter of each process, and three for the three shared variables.

There are five parameters in the pseudo-code, so fifteen boolean variables are required to represent the parameters. Let p denote the vector of the fifteen boolean variables. All the predicates and relations introduced so far are considered indexed by p. For example, we should have written $T_p(x, y)$.

Predicates such as $I_p(x)$ and $S_p(x)$ contain 24 boolean variables, and relations such as $T_p(x, y)$ contain 33 boolean variables. All these predicates and relations are represented by OBDD (ordered binary decision diagrams)

We employed the BDD package developed by David E. Long and distributed from CMU [8]. The verification program was written in C by hand. The essential part of the program is shown in the appendix.

Variables in predicates and relations are ordered as follows.

- variables in x first,
- variables in y (if any) next, and
- variables in p last.

We have tried the opposite order but never succeeded in verification.

By the above order, predicates such as $S_p(x)$ are decomposed into a collection of predicates on p as follows.

```
if ...x... then ...p...
else if ...x... then ...p...
else ...
```

The value of x is successively checked by a series of conditions, and for each condition on x, a predicate on p is applied.

Similarly, relations such as $T_p(x, y)$ are decomposed as follows.

```
if ...x...y... then ...p...
else if ...x...y... then ...p...
else ...
```

3.3 Result

We checked the safety and liveness of the system as described in the previous section. The experiments are made by Vectra VE/450 Series8, from Hewlett-Packard.

- It took 0.853 seconds to check the original version of Peterson's algorithm. The result was, of course, true.
- It took 117.393 seconds to check the template of algorithms containing a fifteen-bit parameter.

Since the template contains fifteen boolean variables, checking the template amounts to checking 2^{15} instances of the template simultaneously. If we simply multiply 0.853 by 2^{15} and compare the result with 117.393, we found speed up of about 238 times.

The size of the BDD that represents the constraints on the parameters is 93. Only 16 instances of parameters satisfy the constraints. So, we found 15 variants of Peterson's algorithm. But they are essentially equivalent to Peterson's algorithm. In the variants, the interpretation of some shared variables is simply reversed.

4 Search for Variants of Peterson's or Dekker's Algorithm

In this section, we show the second case study. We apply the method described in the previous section to another template. Since Dekker's algorithm is very similar to Peterson's, we make a template which cover both algorithms.

4.1 Pseudo-code

We define another template which has both Dekker and Peterson's algorithms as instances. In order to define such a template with a small number of parameters, we modify instructions as follows.

```
SET_CLEAR($p$),$L$
JUMP_IF_TURN($p$),$L_{00}$,$L_{01}$,$L_{10}$,$L_{11}$
JUMP_IF_FLAG_YOU($p$),$L_{00}$,$L_{01}$,$L_{10}$,$L_{11}$
NOP,$L_1$,$L_2$
```

The instruction NOP is the same as in the previous section.

The operator SET_CLEAR is slightly changed to reduce parameters. The parameter p of SET_CLEAR(p) has two-bit value. Each value of the parameter results in a pseudo-instruction as defined in Figure 6.

The operators JUMP_IF_TURN and JUMP_IF_FLAG_YOU have a three-bit parameter and four operands. The operands denote addresses in the pseudo-code. The intuitive meaning of the operands is that L_{00} points to the next address, L_{01} points to the address of the critical section, L_{10} points to the address of the beginning of the loop of the entry part, and L_{10} points to the current address. Each

	Process 1	Process 2
`SET_CLEAR(`$b0$`)`,L	`FLAG0 :=` b`;` `goto` L	`FLAG0 := not(`b`);` `goto` L
`SET_CLEAR(`$b1$`)`,L	`FLAG1 :=` b`;` `goto` L	`FLAG2 :=` b`;` `goto` L

Fig. 6. Modified `SET_CLEAR`

	Process 1	Process 2
`JUMP_IF_TURN(`$b00$`)`,L_{00},L_{01},L_{10},L_{11}	`IF FLAG0 =` b `THEN goto` L_{00} `ELSE goto` L_{00}	`IF FLAG0 = not(`b`)` `THEN goto` L_{00} `ELSE goto` L_{00}
`JUMP_IF_TURN(`$b01$`)`,L_{00},L_{01},L_{10},L_{11}	`IF FLAG0 =` b `THEN goto` L_{01} `ELSE goto` L_{00}	`IF FLAG0 = not(`b`)` `THEN goto` L_{01} `ELSE goto` L_{00}
`JUMP_IF_TURN(`$b10$`)`,L_{00},L_{01},L_{10},L_{11}	`IF FLAG0 =` b `THEN goto` L_{10} `ELSE goto` L_{00}	`IF FLAG0 = not(`b`)` `THEN goto` L_{10} `ELSE goto` L_{00}
`JUMP_IF_TURN(`$b11$`)`,L_{00},L_{01},L_{10},L_{11}	`IF FLAG0 =` b `THEN goto` L_{11} `ELSE goto` L_{00}	`IF FLAG0 = not(`b`)` `THEN goto` L_{11} `ELSE goto` L_{00}

Fig. 7. `JUMP_IF_TURN`

	Process 1	Process 2
`JUMP_IF_FLAG_YOU(`$b00$`)`,L_{00},L_{01},L_{10},L_{11}	`IF FLAG2 =` b `THEN goto` L_{00} `ELSE goto` L_{00}	`IF FLAG1 =` b `THEN goto` L_{00} `ELSE goto` L_{00}
`JUMP_IF_FLAG_YOU(`$b01$`)`,L_{00},L_{01},L_{10},L_{11}	`IF FLAG2 =` b `THEN goto` L_{01} `ELSE goto` L_{00}	`IF FLAG1 =` b `THEN goto` L_{01} `ELSE goto` L_{00}
`JUMP_IF_FLAG_YOU(`$b10$`)`,L_{00},L_{01},L_{10},L_{11}	`IF FLAG2 =` b `THEN goto` L_{10} `ELSE goto` L_{00}	`IF FLAG1 =` b `THEN goto` L_{10} `ELSE goto` L_{00}
`JUMP_IF_FLAG_YOU(`$b11$`)`,L_{00},L_{01},L_{10},L_{11}	`IF FLAG2 =` b `THEN goto` L_{11} `ELSE goto` L_{00}	`IF FLAG1 =` b `THEN goto` L_{11} `ELSE goto` L_{00}

Fig. 8. `JUMP_IF_FLAG_YOU`

value of the parameter results in a pseudo-instruction as defined in Figures 7 and Figures 8.

We define a template as in Figure 9. Note that we fixed the values of some parameters to reduce the search space. In the template, each parameter has two-bit value. So the template contains sixteen boolean variables.

```
 0: SET_CLEAR(11),1
 1: SET_CLEAR($p_1$),2
 2: JUMP_IF_FLAG_YOU(0$p_2$),3,8,2,2
 3: JUMP_IF_TURN(1$p_3$),4,8,2,3
 4: JUMP_IF_TURN(0$p_4$),5,8,2,4
 5: SET_CLEAR($p_5$),6
 6: JUMP_IF_TURN(1$p_6$),7,8,2,6
 7: SET_CLEAR($p_7$),2
 8: NOP,9,9
 9: SET_CLEAR($p_8$),10
10: SET_CLEAR(01),11
11: NOP,0,11
```

Fig. 9. Template 2

The template contains both algorithms as instances. Dekker's algorithm can be represented by the following pseudo-code.

```
0: SET_CLEAR(11),1
1: SET_CLEAR(11),2
2: JUMP_IF_FLAG_YOU(001),3,8,2,2
3: JUMP_IF_TURN(100),4,8,2,3
4: JUMP_IF_TURN(010),5,8,2,4
5: SET_CLEAR(01),6
6: JUMP_IF_TURN(111),7,8,2,6
7: SET_CLEAR(11),2
8: NOP,9,9
9: SET_CLEAR(10),10
10: SET_CLEAR(01),11
11: NOP,0,11
```

The Peterson's algorithm can be represented by the following pseudo-code.

```
0: SET_CLEAR(11),1
1: SET_CLEAR(10),2
2: JUMP_IF_FLAG_YOU(001),3,8,2,2
3: JUMP_IF_TURN(110),4,8,2,3
4: JUMP_IF_TURN(001),5,8,2,4
5: SET_CLEAR(00),6
6: JUMP_IF_TURN(100),7,8,2,6
7: SET_CLEAR(00),2
8: NOP,9,9
```

```
9: SET_CLEAR(01),10
10: SET_CLEAR(01),11
11: NOP,0,11
```

4.2 Verification

Verification of the template goes almost in the same manner as in the previous section.

In this verification, the condition of an error state is

$$\texttt{PC1} = \texttt{PC2} = 8.$$

For the liveness, we modify the condition of the starvation loop of Process 1 as follows.

$$0 \leq \texttt{PC1} \leq 7$$

Since there are twelve pseudo-instructions in the pseudo-code, the representation of the program counter of each process requires four bits. Therefore, a state in the state transition system can be represented by eleven boolean variables. Sixteen boolean variables are required to represented the parameters.

4.3 Result

We checked the safety and liveness of the system on the same machine as in the previous section.

- It took 10.195 seconds to check the Peterson's algorithm. The result was, of course, true.
- It took 19.506 seconds to check the Dekker's algorithm. The result was, of course, true.
- It took 741.636 seconds to check the template of algorithms containing a sixteen-bit parameter.

We found speed up of about 900 times at least.

The size of the BDD that represents the constraints on the parameters is 62. There are about 400 solutions of the constraints. We found that just one solution represents the original Dekker's algorithm, and the remaining solutions essentially represent Peterson's algorithm.

5 Approximation

Remember that according to the variable ordering we adopted in BDDs, a predicate on states indexed by p is represented as a collection of sub-predicates on p as follows.

```
**if** $C_0(x)$ **then** $P_0(p)$
**else if** $C_1(x)$ **then** $P_1(p)$
**else if** $C_2(x)$ **then** $P_2(p)$
**else** ...
```

The sub-predicates, $P_0(p), P_1(p), P_1(p), \cdots$, occupy the branches of the entire predicate. We tried to reduce the size of such a predicate by collapsing some of the sub-predicates on p with a small Hamming distance. This is considered a first step towards using approximation in our approach.

The Hamming distance between two predicates $P_i(p)$ and $P_j(p)$ is the fraction of assignments to p that make $P_i(p)$ and $P_j(p)$ different. For example, the Hamming distance between $p_0 \wedge p_1$ and $p_0 \vee p_1$ is 1/2, since they agree on the assignments ($p_0 = 0,\ p_1 = 0$) and ($p_0 = 1,\ p_1 = 1$), while they do not agree on the assignments ($p_0 = 0,\ p_1 = 1$) and ($p_0 = 1,\ p_1 = 0$).

Let θ be some threshold between 0 and 1. By collapsing sub-predicates $P_i(p)$ and $P_j(p)$, we mean to replace both $P_i(p)$ and $P_j(p)$ with $P_i(p) \vee P_j(p)$, provided that their Hamming distance is less than θ. After $P_i(p)$ and $P_j(p)$ are replaced with $P_i(p) \vee P_j(p)$, the size of the entire predicate is reduced because the BDD node representing $P_i(p) \vee P_j(p)$ is shared. We note that this collapsed predicate R' by replacing the disjunction is bigger than the original predicate R, i.e. $\forall x.\ R(x) \Rightarrow R'(x)$.

We made the following experiment. We first computed the reachability predicate $R_p(y)$ defined as follows.

$$\exists x.\ I_p(x) \wedge T_p^*(x, y)$$

We then collapsed $R_p(y)$ according to some threshold θ, and continued verification using the collapsed $R'_p(y)$.

The discovered algorithms by using $R'_p(y)$ always satisfy the safety and the liveness. The expression of safety is $\neg\exists y.\ R_p(y) \wedge E(y)$. Because the collapsed predicate is bigger, $\forall p.\ (\neg\exists y.\ R'_p(y) \wedge E(y)) \Rightarrow (\neg\exists y.\ R_p(y) \wedge E(y))$. If the algorithm with parameter p satisfies the safety under the collapsed $R'_p(y)$, it satisfies the safety. This discussion can be used for the liveness, because the expression of the liveness is $\neg\exists y.\ R_p(y) \wedge S'(y)$.

We successfully obtained constraints on the parameters. We summarize the results in Figure 10. In the figure, if the size of the final result is marked as '$-$', it means that verification failed, i.e., no instantiation of parameters could satisfy the safety and liveness.

Using abstract BDDs [3] is another approach to reduce the size of BDDs. It is similar to ours in that abstract BDDs are obtained by merging BDD nodes whose abstract values coincide. Abstract values of BDD nodes are given in advance. In our case, since it is difficult to define such abstraction before the search, we dynamically collapse the algorithm space according to the Hamming distance of BDDs.

Template	Threshold	Size of $R_p(y)$		Size of final result	
	θ	w/ collapse	w/o collapse	w/ collapse	w/o collapse
Template 1	0.03	175	278	38	93
Template 1	0.05	145	278	–	93
Template 2	0.15	50	429	30	62
Template 2	0.20	39	429	–	62

Fig. 10. Results of collapsing

6 Conclusion

We searched for mutual exclusion algorithms using BDDs. Since the algorithm space was given by a template, we could simultaneously check all algorithms in the algorithm space by a single execution of the verifier. We gained speed-up of several hundred times compared with verifying all algorithms one by one.

The result of the second case study might suggest that Peterson's algorithm could be discovered as a variant of Dekker's algorithm. In the previous study [7], we searched for variants of Dekker's algorithm, but we could not find Peterson's. This was because we fixed the finishing part as that of the original version of Dekker's. Designing an algorithm space that contains interesting algorithms is not easy. In these case studies, we found only known algorithms.

In this case study, we could not find new algorithms. Algorithm space we used may be too closed to the original algorithms. The construction of effective algorithm space is a problem. In superoptimizer, the candidates are generated independent to the original code. The independency to the original one may be a key to find out essential new algorithms.

Analysis of the resulting constraints obtained by verification was also a difficult task. We examined all solutions of the constraints by hand, and we found out the discovered algorithms are equivalent to the original algorithms. Automatic check of equivalence of algorithms greatly help our method. We should develop automatic equivalence checker.

Acknowledgments

The work reported in this paper was supported, in part, by the Ministry of Education, Science, Sports and Culture, under the grants 11480062 and 10143105.

References

1. R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers,* Vol.C-35, No.8, pp.677–691, 1986.
2. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking,* The MIT Press, 1999.

3. Edmund M. Clarke, Somesh Jha, Yuan Lu, and Dong Wang. Abstact BDDs: A Technique for Using Abstraction in Model Checking. *Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science, Vol.1703, pp.172–186, 1999.
4. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the 4th ACM Symposium on Principles of Programming Languages,* pp.238–252, 1977.
5. Torbjörn Granlund and Richard Kenner. Eliminating Branches using a Superoptimizer and the GNU C Compiler. *PLDI'92, Proceedings of the conference on Programming language design and implementation,* pp.341–352, 1992.
6. Masami Hagiya. Discovering Algorithms by Verifiers. Programming Symposium, Information Processing Society of Japan, pp.9–19, 2000, in Japanese.
7. Masami Hagiya and Koichi Takahashi. Discovery and Deduction, *Discovery Science, Third International Conference, DS 2000* (Setsuo Arikawa and Shinichi Morishita Eds.), Lecture Notes in Artificial Intelligence, Vol.1967, pp.17–37, 2000.
8. David E. Long. bdd - a binary decision diagram (BDD) package, 1993. `http://www.cs.cmu.edu/~modelcheck/code.html`
9. Henry Massalin. Superoptimizer: A Look at the Smallest Program. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS II,* pp.122–126, 1987.
10. Adrian Perrig and Dawn Song. A First Step on Automatic Protocol Generation of Security Protocols. *Proceedings of Network and Distributed System Security,* 2000 Feb.
11. Dawn Xiaodong Song. Athena: a New Efficient Automatic Checker for Security Protocol Analysis, *Proceedings of the 12th IEEE Computer Security Foundations Workshop,* pp.192–202, 1999.

Appendix

In this appendix, we show the essential part of the verification program of Section 3.2. It employs the BDD package developed by David E. Long and distributed from CMU [8]. Functions beginning with "`bdd_`" are from the bdd package. They include, for example, the following functions.

- `bdd_not`: returns the negation of a BDD.
- `bdd_and`: returns the conjunction of BDDs.
- `bdd_or`: returns the disjunction of BDDs.

Their first argument is the bdd manager as defined in the bdd package [8] and is of the type `bdd_manager`. The second and third arguments are BDDs of the type `bdd`.

The function `bdd_exists` computes the existential quantification of a given BDD. Before calling `bdd_exists`, the array of variables to be quantified should be specified as follows.

```
bdd_temp_assoc(bddm, y, 0);
bdd_assoc(bddm, -1);
```

The function `bdd_rel_prod` computes the relational product of given BDDs. Semantically, it first computes the conjunction of the second and third arguments and then computes the existential quantification as `bdd_exists` does.

We also defined some auxiliary functions described below.

The C function `bdd_closure_relation` takes the following seven arguments and computes the closure of a given relation.

- `bdd_manager bddm`: the bdd manager.
- `int AND`: the flag specifying whether the closure is computed by conjunction (if it is 1) or disjunction (if it is 0).
- `bdd g`: the initial predicate.
- `bdd f`: the relation whose closure is computed. It contains variables in `x` and `y`.
- `bdd *x`: the array of the variables in `f` and `g`.
- `bdd *y`: the array of the variables in `f`.
- `bdd *z`: the array of temporary variables.

Let x denote the vector of variables in `x`, y the vector of variables in `y`. It is assumed that `g` is a predicate on x and `f` is a relation between x and y. If the value of the flag `AND` is 0, then the above function computes and returns the following predicate on y.

$$\exists x.\ \mathtt{g}(x) \wedge \mathtt{f}^*(x, y)$$

The return value of the function is of the type `bdd`.

Following are some more auxiliary functions.

- `bdd_rename`: renames variables in a BDD and returns the resulting BDD.
- `bdd_equal_int`: constrains an array of variables as a binary number. For example,

  ```
  bdd_equal_int(bddm, 3, pc11, 4)
  ```

  returns the bdd expressing the following condition.
  $\mathtt{pc11[0]} = \mathtt{pc11[1]} = 0 \wedge$
  $\mathtt{pc11[2]} = 1$
- `bdd_and3`: returns the conjunction of three BDDs.
- `bdd_or4`: returns the disjunction of four BDDs.

The function `smash` makes approximation as described in Section 5. It takes the bdd manager, the BDD to be collapsed, and the threshold value. The identifier `smash_threshold` must have been defined as a C macro.

Following is the main function of the verification program. In the initialization part, omitted from the following code, the transition relations are initialized as follows.

- `t`: the one-step transition relation.
- `t1`: the one-step transition relation for Process 1.
- `t2`: the one-step transition relation for Process 2.

States are represented by nine variables. In the main function, the arrays x, y and z store nine variables representing states. Since they are null terminated, their size is 10. x denotes the state before a transition and y the state after a transition. z consists of temporary variables. The pointers pc01, pc02, pc11, and pc12 point to the program counters in x and y.

```
int main()
{
  bdd_manager bddm;
  bdd x[10];
  bdd *pc01 = &x[0];
  bdd *pc02 = &x[3];
  bdd *vs0 = &x[6];
  bdd y[10];
  bdd *pc11 = &y[0];
  bdd *pc12 = &y[3];
  bdd *vs1 = &y[6];
  bdd z[10], *p, *q;
  bdd t, t1, t2,
      initial, reachable, conflict;
  bdd starving, starving_t,
      starving_t1, starving_t2;
  bdd starvation, s, s0;
  bdd e;

  bddm = bdd_init();

  ...
  /* initialization */
  ...

  initial = bdd_equal_int(bddm, 9, y, 0);
  reachable = bdd_closure_relation(bddm, 0,
                                   initial,
                                   t, x, y, z);

#ifdef smash_reachable
  reachable = smash(bddm, reachable,
                    smash_reachable);
#endif

  conflict = bdd_and3(bddm, reachable,
                      bdd_equal_int(bddm, 3,
                                    pc11, 4),
                      bdd_equal_int(bddm, 3,
                                    pc12, 4));
  bdd_temp_assoc(bddm, y, 0);
  bdd_assoc(bddm, -1);
  conflict = bdd_exists(bddm, conflict);

  starving = bdd_or4(bddm,
                     bdd_equal_int(bddm, 3,
                                   pc01, 0),
                     bdd_equal_int(bddm, 3,
                                   pc01, 1),
                     bdd_equal_int(bddm, 3,
                                   pc01, 2),
                     bdd_equal_int(bddm, 3,
                                   pc01, 3));

  starving_t = bdd_and(bddm, starving, t);
  starving_t1 = bdd_and(bddm, starving, t1);
  starving_t2 = bdd_and(bddm, starving, t2);

  s = starving;
  do {
    printf("iteration for fairness\n");
    s0 = s;

    s = bdd_rename(bddm, s, x, y);
    bdd_temp_assoc(bddm, y, 0);
    bdd_assoc(bddm, -1);
    s = bdd_rel_prod(bddm, starving_t1, s);
    s = bdd_closure_relation(bddm, 0, s,
                             starving_t,
                             y, x, z);

    s = bdd_rename(bddm, s, x, y);
    bdd_temp_assoc(bddm, y, 0);
    bdd_assoc(bddm, -1);
    s = bdd_rel_prod(bddm, starving_t2, s);
    s = bdd_closure_relation(bddm, 0, s,
                             starving_t,
                             y, x, z);

  } while (s0 != s);

  s = bdd_rename(bddm, s, x, y);
  bdd_temp_assoc(bddm, y, 0);
  bdd_assoc(bddm, -1);
  starvation = bdd_rel_prod(bddm, reachable, s);

  e = bdd_and(bddm,
              bdd_not(bddm, conflict),
              bdd_not(bddm, starvation));
}
```