VeriSHIM
A BDD Verifier for SHIM

Nalini Vasudevan
Department of Computer Science
Columbia University
New York, USA
naliniv@cs.columbia.edu

Abstract

Keywords: Concurrency, SHIM, Deadlock, BDD, Reachability

Concurrent programming languages have become more popular with the advent of multi-core systems. Shared memory is read or written atomically, by concurrent processes to prevent races. A typical mechanism is the use of locks. However, if locks are not acquired and released in the correct order, the result is a deadlock.

In this paper, we propose a deadlock detection tool for a deterministic, concurrent language, SHIM. SHIM is race free but not deadlock free. A deadlock can easily be detected while the program is running, but here we use static approaches to find deadlocks in a program. Enumerating all possible states is not feasible because there will be explosion of states. We exploit the deterministic property of SHIM and do a BDD based reachability analysis after abstracting parts of the program. We run the deadlock detector on the the JPEG decoder and report the results.

1 Introduction

Concurrent programs are known for errors that are difficult to diagonize and correct. One such class of errors is deadlock. A deadlock is a state in a program, where two or more processes wait for each other to finish a task, and neither ever finish resulting in indefinite blocking. Deadlock detection and hence, removal of deadlock states are two important aspects in concurrent programming. In this paper, we focus on the deadlock diagnosis in a concurrent programming language called SHIM.

SHIM [5] was designed to provide a common programming environment for both hardware and software. It is deterministic i.e it is guaranteed to behave the same irrespective of the scheduling of the concurrent blocks in the program. Data can be read or written into shared memory in a restricted fashion. The model combines the functional determinism of Kahn networks [8] with the rendezvous of Hoare's CSP [7]. To achieve determinism, a lock based approach is used. Although, SHIM guarantees determinism, a program may not be deadlock free. SHIM's philosophy is that it is easier to detect deadlocks than to find data races in the program. One important aspect of SHIM is that even the deadlocks are deterministic, i.e. the program deadlocks at the same place given a particular input sequence.

In static analysis, the programs are not executed. A naive approach is to find all possible

paths in the program by an exhaustive state space search. This is almost impractical because of the large state spaces of programs. SHIM [12], being deterministic, ensures that the output of a program is independent of the scheduling of parallel threads. Therefore, we can restrict the state space to any one schedule rather than explore all possible schedules.

Another approach, is to restrict the state space by doing some kind of abstraction. If the tool reports that there is no deadlock, there will be no reachable deadlock state in the program. On the other hand, the tool may report a possibility of deadlock. There is some path in the program that is not deadlock free and this path may be executed only for some particular input sequence. Reducing the state space by some approximations leads to false positives. In other words, there are some programs that will never deadlock for any input sequence, but yet the tool reports a possibility of deadlock.

2 Related Work

The two main problems in concurrent programming are data races and deadlocks. Boyapati [1] talks about ownership types for safe programming to prevent data races and deadlocks. Corbett [4] evaluates deadlock detection methods for concurrent software. In this paper, we address the problem of deadlocks. Deadlocks can be either detected dynamically or statically. Generally, dynamic deadlock detection [9] can be done easily at run time, while static deadlock detection is difficult.

There has been considerable work done in verifying the C and Java programming language. Abstraction and counter-example guided refinement are popular approaches to model checking. Chaki [3] [2] uses these methods to verify C programs. RacerX [6] is another tool that detects deadlocks and data races in C. Jlint is a static deadlock detector for Java that uses lock-order-graph. Java Pathfinder [13] finds execution paths and it also reports if a program is susceptible to deadlocks.

SHIM is a multi-way rendezvous language, and it is quite different from C and Java. SHIM is guaranteed to be race-free but it is susceptible to deadlocks. The deadlock vulnerability arises out of rendezvous. Ada is one of the programming languages that uses rendezvous. Masticola [11] describes an algorithm for detecting deadlocks in some constructs of the Ada Programming Language. Ada supports 2-way rendezvous while SHIM supports multi-way rendezvous and hence it makes the task difficult.

In this paper, we devise an algorithm for a multi-way rendezvous language called SHIM. We use BDDs as a reachability analysis tool in our algorithm.

3 Background: The SHIM Programming Language

SHIM [12] is a C like language with additional constructs: *p par q* runs statements *p* and *q* in parallel, waiting for both to terminate before proceeding; *next c* is a blocking communication operator that synchronizes on channel *c* and either sends or receives data depending on which side of = the *next* appears, SHIM uses multi-way rendezvous to communicate with peer threads. We use the terms process and threads interchangeably in this paper, and they both mean the same in this context.

In Figure 1, two peer threads communicate on channels *a* and *b*. Threads *1* and *2* are executed in parallel. SHIM interprets the *next a* in Thread *1* as a send because it is on the left hand side of the assignment. The *next a* in Thread *2* is a receive. The *next a* in Thread *1* waits for Thread *2* to receive the value. The threads therefore rendezvous at their *next*s,

```
void main (int8 & cout)
{
 chan int a, b;
  {
   /* Thread 1 */
        next a = 5;
/* Sets a and send (wait for thread 2) */
        /* a now 5 */
        next b;
/* Receive a (wait for thread 2) */
        /* b now 10 */
  }
  par
  /* Thread 2 */
  {
        next a;
/* Receive a (wait for thread 1) */
        /* a now 5 */
        next b = 10;
/* Sets b and send (wait for thread 1) */
        /* b now 10 */
  }

}
```

Figure 1: A SHIM program

then continue to run after the communication takes place. In the next step, the two threads rendezvous at *next b*. In this case, Thread *1* is the receiver while Thread *2* tries to send.

Consider a modification of the above program in Figure 2.

```
void main (int8 &cout)
{
 chan int a, b;
  {
   /* Thread 1*/
        next a = 5;
        next b = 10;
  }
  par
  /* Thread 2 */
  {
        next b;
        next a;
  }
}
```

Figure 2: A SHIM program with deadlock

In Figure 2, two peer threads try to communicate on channels *a* and *b*. Threads *1* and *2* are executed in parallel. SHIM interprets the *next a* in Thread *1* as a send. The *next b* in Thread *2* is a receive. Thread *1* waits for Thread *2* to receive the value on channel *a*. Thread *2* waits for Thread *1* to send a value on channel *b*. Both threads wait for each other on different channels indefinitely, resulting in a deadlock.

This is a simple illustration of a deadlock in SHIM. In the further sections, we will be seeing the different approaches used to detect deadlocks in SHIM.

```
current_states := initial_state
explored_states := nil
while (current_states not in explored_states)
    do
        explored_states := explored_states + current_states
        next_states := get_next_states (current_states)
        if (next_states has a deadlock state)
            then report ("Deadlock state found")
        current_states := next_states
    end while
```

Figure 3: Deadlock Reachability Analysis

4 Explicit State Space Exploration

Irrespective of the approach, we maintain only the synchronization skeleton of the program. In the explicit state space exploration method (Figure 3), states are analyzed one after the other. We start at the initial state, get all the possible successors of the state. We see if there is some successor that represents a deadlock state. If yes, we report it else we explore the successors state. We iterate this procedure until all states have been explored.

The SHIM compiler dismantles a program into statement lists. A SHIM program and its dismantled version are shown in Figure 4 and Figure 6 respectively. It is evident that if $n > 10$, the first thread tries to communicate on a, while the second thread tries to communicate on b resulting in a deadlock. On the other hand, if $n <= 10$, the first thread

will want to communicate on b and the second thread will also want to communicate on b and therefore they rendezvous and exchange values, and proceed to *next a*. Again here, the communication is successful because both threads are willing to communicate on a, resulting in successful termination of both threads. The program for a given schedule, has only two paths, one when the condition is true and the other when it is false. We do not care about the details of *some_number*. All that we care and want is that it returns an integer.

```
void main (int &cout)
{
 chan int a, b;
 int n;

  {
   /* Thread 1 */
        n = some_number();
        if (n > 10)
                next a = 5;
        else
                next b = 5;
        next a = 10;
  }
  par
  /* Thread 2 */
  {
         next b;
         next a;
  }
}
```

Figure 4: Snippet of a SHIM program with conditional statements

We are at the moment, not doing any kind of predicate analysis. Therefore, we assume equal probability of execution of the *if* and *else* part of the conditional statement. Therefore Figure 4 can be rewritten as Figure 5.

When we are exploring the state space of a SHIM program, we do not have to consider all possible inter-leavings of the SHIM program. Since SHIM is deterministic, and also that deadlocks happen deterministically, it is sufficient to explore any one schedule of processes

```
void main (int &cout)
c chan int a, b;
 int n;

  {
   /* Thread 1 */
        n = some_number();
        if (*) /* We do not analyze the condition*/
                next a = 5;
        else
                next b = 5;
        next a = 10;
  }
  par
  /* Thread 2 */
  {
         next b;
         next a;
  }
}
```

Figure 5: Replacing conditions by wild-card: Giving equal probability to branch statements

```
main(int8 &cout)
   channel int32 a
   channel int32 b
   local int32 n
      main_1(a, b, n) : main_2(a, b);

main_1(chan int32 &a, chan int32 &b, int32 &n)
   local int32 _tmp0
   local int32 _tmp1
     some_number(_tmp0);
     n = _tmp0
     _tmp1 = n > 10
     ifnot _tmp1 goto _else3
     a = 5
     send a
     goto _endif4
     _else3:
     b = 5
     send b
     _endif4:
     a = 10
     send a

main_2(chan int32 a, chan int32 b)
     recv b
     recv a
```

Figure 6: Dismantled version of Figure 4

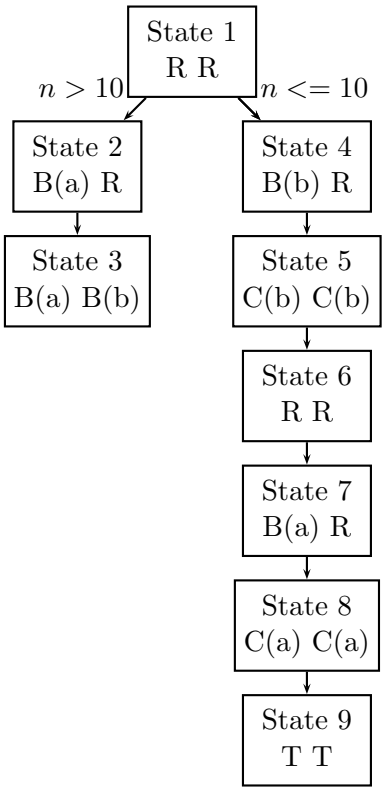

Figure 7: Automaton for Figure 4. R = Runnable, B(x) = Blocked on Channel x, C(x) = Communicating on Channel x, T = Terminated.

and not necessarily explore all possible interleavings. This approach is something similar to *Improviso* [10] algorithm that uses partial order reduction. Figure 7 shows the automaton representing the program in Figure 2. We use the reachability analysis (Figure 3) to find if there is any path in the program that will deadlock. Each state represents the combined state of two threads. Both are runnable at the start, and depending on the conditional statement, either both get blocked and end up in a deadlock state, or both terminate successfully. While expanding states, we pick any one possible schedule. In this case we pick the first runnable process in left-to-right-order. If a state has more than one outgoing transition, it is due to a conditional statement at that point. *State*3 is the deadlock state in the program, because both the threads are blocked on different channels.

Note that we are ignoring all other statements such as simple assignments etc., that do not contribute to parallelism or communication.

5 The BDD Approach

Explicit state space approach guarantees the right answer but it is only suitable for small programs. With programs with very large state spaces, we either run out of memory or it takes a very long time to execute. Hence we use a symbolic approach. In this method, we encode states as bits. If we represent a state as n bits, the state space is restricted to 2^n bits. Software processes generally have complicated data structures and complex functionality such as recursion. Therefore it is hard to restrict the state space which makes them difficult to be encoded as bits. Implicit approach is generally used for modeling hardware design. The number of bits represented by a state can be reduced by doing some kind of abstraction. We discuss it

in detail in this section.

5.0.1 Abstracting the program

As a first step of the program, we remove all irrelevant information such as simple assignment statements etc., and we only project the part of the program that has parallel and communication constructs in it. We maintain the conditional statements, because they define paths in the program. This abstract interpretation of the program enables us to concentrate on the communication part of the program rather than the computation part. We keep the conditionals, but treat them as wild cards giving equal probability to the if and else branches. Note that *while*, *for* and other looping statements can be converted to a combination of *if* and *goto* statements. We treat every *if* as a wild-card. We do this for both implicit and explicit approach.

In the implicit approach, we would like to minimize the number of states as possible to maintain n as small as possible. Increasing n would increase the time and space complexity exponentially in the worst case. Therefore to decrease n, we do some kind of abstraction. In our approach, we use the following approximation.

Consider the code segment of *Thread 1* in Figure 4. Lets rewrite the code with program counters as shown in Figure 8. The automaton for the thread in isolation is shown in Figure 9. We see that *State 2* and *State 4*, hold the same state value, but their program counters are different i.e. their input or output transitions are different. However, to reduce the state space, we merge the two states into one as shown in Figure 10. We have eliminated the need of pc. By doing this we are inducing a new property into the merged states. A state can possibly have more output transitions in the abstracted automaton compared to the original automaton. By this kind of approximation, we are loosing precision in the accuracy of output, which leads to false positives. In other words, our tool may report a deadlock even when there is no possibility of a deadlock in the program, but the probability of a false positive is very low.

```
{
 /* Thread 1 */
1 n = some_number();
2 if (n > 10)
3 next a = 5;
4 else
5 next b = 5;
6 next a = 10;
}
```

Figure 8: *Thread 1* of Figure 4 with program counters enumerated

We are at the moment, not doing any kind of predicate analysis. Therefore, we assume equal probability of execution of the *if* and *else* part of the conditional statement. Therefore Figure 4 can be rewritten as Figure 5.

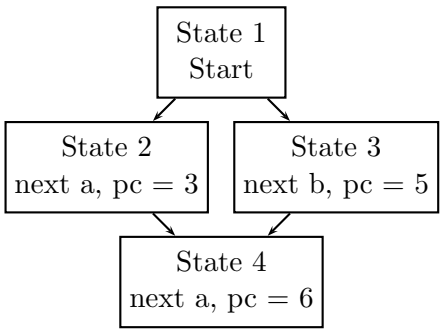

Figure 9: Automaton for *Thread 1* of Figure 4 in isolation. Program counters are enumerated in Figure 8

When is this kind of abstraction actually useful? Consider the automaton shown inFigure 11 with program counter not abstracted. The equivalent automaton with pc abstracted is shown in Figure 12. In this automaton, we

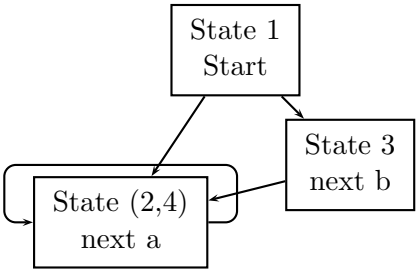

Figure 10: Abstraction of Figure 9

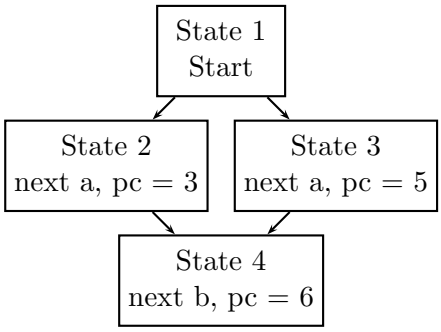

Figure 11: Another simple automaton for example

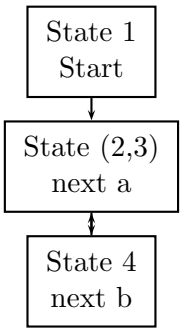

Figure 12: Abstraction of Figure 11

are not inducing any false path and we are reducing the number of states too.

In any case, the number of states can only reduce after pc abstraction, but the false paths may be induced or not, depending on the structure of the program.

In addition to the abstraction, we have made the following assumptions. We assume that the program does not have recursion. Also, we assume that all functions are unique, that is a function is never called from two different places. To do this, we preprocess the input, by simply copying the functions and renaming them.

5.0.2 The BDD Algorithm

Let us remember that we are assuming the program has no recursion or the depth of recursion is restricted to some level that we can unroll it. This enables us to statically look at the program, and give a count of the number of channels and threads used. We can also statically look at the program and see in what functions does a channel occur.

In general, a state is represented by a tuple (p1, p2, p3, p4,....., pn) where each component represents the state of process *pi*. We can statically determine the number of processes in the program. Note that we are using the terms *threads* and *processes* interchangeably. Both mean the same in this context. A state can be in any one of the following states:

- Not Running: The process has not yet started running.
- Par called: The process has reached its par-statement. It has not yet instantiated its children.
- Children running: A process is waiting for its running children to terminate.
- Blocked on channel ch: The process is waiting to communicate on channel ch

- Terminated: The process has terminated.

We use the following bit convention to encode the states as bits

- Not Running: 000
- Par Called: 100
- Children running: 110
- Blocked on a channel: 111
- Terminated: 001

If there are m channels in the program, then the number of bits needed to encode a channel is equal to

$$channel\_bits = log(m-1) + 1$$

We require 3 bits to represent the state and *channel_bits* to represent the channel if $state = blocked$. Therefore the state of a particular thread can be represented in $3 + channel\_bits$. If there are n threads in the program, then the entire state of a program can be represented in $n * (3 + channel\_bits)$. For example, consider a program with two channels, c and d and 4 threads. If the state of the program is

$$(blocked\_c, blocked\_d, not\_running, not\_running)$$

it implies that *process 1* is blocked on *channel c*, *process 2* on *channel d* and the other two processes have not yet started running. So the state of the program in terms of bits is

$$(1110, 1111, 000X, 000X)$$

We need one bit to represent the channel and 3 bits to represent the process state. So we can represent the state of a thread with 4 bits. The last bit of the 4 bits represents the channel if the state of the channel is blocked. It is dont-care-X when the 4th bit is not used. Now if we want to build a BDD out of this, we need 16 variables for 4 threads, where each thread is represented by 4 variables.

As a next step, we construct the transition function for the SHIM program as a BDD. Given a pair $(curr\_state, next\_state)$, the transition function returns true if the transition is possible and false if it is not. Both the *curr_state* as well as the *next_state* in isolation represent the state of the entire program, with all possible threads in it. We construct this transition function as a BDD and use this in reachability analysis.

Before we see the procedure to build the transition function, first let us see how to perform reachability analysis using BDDs. BDDs are typically used for set manipulation. If P and Q are two sets, and if we can represent them as P1 and Q1 that are BDDs, then we can use the equivalent operations in Table 1 to manipulate sets using BDDS.

Set operation	BDD operation
P ∪ Q	BDD_OR (P1, Q1)
P ∩ Q	BDD_AND (P1, Q1)
P'	BDD_NOT (P1)
P ⊂ Q	BDD_CONTAINS (Q1, P1)

Table 1: Using BDDs to manipulate Sets

Figure 13 shows the state space exploration using BDDs. We precompute *par_transitions*, *termination_transitions* and *communication_transitions* that are specific to SHIM. We then find successors of new states using these transitions. The BDD *allowed_transitions* represents represents the transition function of the program. Given the current states, we use BDD_EXISTS, to get the next possible states from *allowed_transitions*.

Let us now precompute all deadlock states (*deadlock_states*). If there are two processes p and q and both are owning i and j, then

```
current_states := initial_state
explored_states := nil
allowed_transitions := BDD_OR (par_transitions,
                               termination_transitions,
                               communication_transitions);
while (not BDD_CONTAINS (explored_states,
                         current_states))
    do
        explored_states := BDD_OR (explored_states,
                                   current_states);
        next_states := BDD_EXISTS (current_states,
                                   allowed_transitions);
        if (BDD_AND (next_states, deadlock_states))
            then report ("Deadlock state found")
        current_states := next_states
    end while
```

Figure 13: Deadlock Reachability using BDDs

```
for every channel pair (i,j)
    for every process pair (p, q) where both
                  p and q own i and j, do
        if ((p blocked on i) and (q blocked on j))
           or ((p blocked on j) and (q blocked on i))
                then "Deadlock state"
        end if
    end for
end for
```

Figure 14: Calculating Deadlock States

the processes are in a deadlock state if p is trying to communicate on i and q is trying to communicate on j simultaneously or vice verse. In both cases both the processes wait for each other indefinitely causing a deadlock. To make the definition of *own* clear, we say that a process p owns a channel c, if c is used in the code section of p.

For the program shown in Figure 2, we have two channels, a and b. So we require 1 bit for representing a channel. There are two threads $T1$ and $T2$ and the main thread T. A deadlock happens when $T1$ is blocked on channel a and $T2$ is blocked on channel b at the same time, or when $T1$ is blocked on channel b and $T2$ is blocked on channel a simultaneously. If we represent the states of $T1$, $T2$ and T as a tuple ($T1$, $T2$, T), then a deadlock state in terms of bits would be (1110, 1111, XXXX) or the other way round (1111, 1110, XXXX). Now we construct a BDD that represents all possible deadlock states in the program. The BDD will have 12 variables, 4 representing each thread. The BDD will return true when either of the following conditions hold:

- 11101111XXXX → True
- 11111110XXXX → True
- False otherwise

Now let us get the BDD for *par_transitions*. Whenever a process p calls par, we instantiate all its children, i.e. the children advance themselves from the not-running state and move to some other state. Finally the state of process p is set to children-running. Using the algorithm in

```
if p has just called par
    for every child q of process p
        instantiate q by advancing
                  state of q from not-running
    set state of p to children-running
end if
```

Figure 15: Instantiating child processes

Figure 16, we can build a transition BDD, that contains the current state and next state, where the current state has a process that has called *par*.

Let us now write algorithms for state advancement i.e. when a communication can proceed. We encode these transitions as *communication_transitions*. Suppose a process p, is blocked on channel i, then it can move ahead only if all the processes that hold channel i are trying to communicate on channel i However, there are a few exceptions. We do not have to wait for a process that has already terminated. We also do not have to wait for a process that has its children executing. The algorithm for communication synchronization is shown in Figure 16.

```
if (p blocked on i) then
    if (for every process q, holding i
                              and q != p)
       (q is (blocked on i) or (terminated)
        or (has children-running))
    then unblock and advance p;
end if
```

Figure 16: Advancing from a blocked state

Finally when every child of a process p terminates, we advance the state of p i.e we allow p to execute the statement after the *par* statement. We also set the states of the children processes to not running (Figure 17). The *termination_transitions* refer to these transitions.

```
for every child q of process p
    if q has terminated
        change state of q to not-running
advance state of p
end if
```

Figure 17: Children terminate, parent advances

We now have BDDs for *par_transitions*, *communication_transitions* and *termination_transitions*. We can combine all transitions into one BDD named *allowed_transitions*. This sole BDD represents the entire SHIM program. All we require is to manipulate this BDD using reachability analysis algorithm in Figure 13.

6 Results

6.1 Experimental Analysis

We ran the deadlock detector on a parallel jpeg decoder written in SHIM and we report the statistics here.

The jpeg decoder is one of SHIM's biggest applications, in terms of number of lines of code. With the jpeg decoder, the explicit detector runs out of memory. The BDD approach with pc abstracted verifies that the program is deadlock free in less than 2 minutes. As it can be seen in Table 2, the BDD version with program counter abstracted runs faster when compared to the explicit version. Our tool also used a reasonable amount of memory on the jpeg decoder. As future work, we would like to write a BDD version without abstracting the program counter, and see if program counter abstraction really contributes to the speed up.

6.2 Program Termination

The solution to the deadlock problem partially solves the termination problem. Let us not abstract the program. If there is a path in the program that deadlocks, then it means there is also a path in a program that will never terminate. If there are no loops or recursive calls in the program, and there is no deadlock path, then we can certainly say that the program will terminate irrespective of the dynamic parameters of the program. If there are loops in the program, we cannot guarantee the termination of the program, because we do no kind of conditional statement analysis.

7 Conclusion

In this paper, we have addressed the verification issues in SHIM. We have provided a BDD deadlock detecting tool for the SHIM language. We have compared the approach with the explicit method and provided the results. We have also touched upon program termination.

From the results section, we see that the biggest application of SHIM is verified in less than a minute. Since the verifier is in its early stages of development, with further more optimization, we believe that we can bring down the running time of the verifier by a great extent. We also wish to integrate the deadlock detector with the compiler. In other words,

Attribute	Explicit Approach	BDD (without pc)
Output	-	No Deadlock
Time	> 2 hours	1 min 58 secs
Memory(RSS)	Ran out of stack	17MB

Table 2: JPEG Decoder - 1020 lines of code, 7 channels, 59 processes

we want to always check for deadlocks, while compiling a SHIM program.

References

[1] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[2] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

[3] Sagar Chaki, Edmund Clarke, Joel Ouaknine, and Natasha Sharygina. Automated, compositional and iterative deadlock detection. In *Proceedings of Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2004)*, 2004.

[4] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, March 1996.

[5] Stephen A. Edwards. SHIM: A language for hardware/software integration. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Electronic Notes in Theoretical Computer Science, Edinburgh, Scotland, April 2005.

[6] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks, 2003.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

[8] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[9] Dongmyun Lee and Myunghwan Kim. A distributed scheme for dynamic deadlock detection and resolution. *Inf. Sci.*, 64(1-2):149–164, 1992.

[10] Flavio Lerda, Nishant Sinha, and Michael Theobald. Symbolic model checking of software. In Byron Cook, Scott Stoller, and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[11] Stephen P. Masticola and Barbara G. Ryder. A model of Ada programs for static deadlock detection in polynomial times. In *Proceedings of the Workshop on Parallel and Distributed Debugging (PADD)*, pages 97–107, New York, NY, USA, May 1991. ACM.

[12] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, October 2006.

[13] Willem Visser, Klaus Havelund, Guillaume Brat, and Seungjoon Park. Model checking programs. In *Proceedings of Automated Software Engineering (ASE)*, pages 3–11, Grenoble, France, September 2000.

Code Listing

```
/******************
*
* BDD Deadlock Detector
* Nalini Vasudevan
*
******************/

#define DD_DEBUG
#include "headers/util.h"
#include "headers/cudd.h"
#include <math.h>
#define BUF_SIZE 1024



char** channel_details ;
char** children_details ;
char*** individual_process_transitions = NULL;
int* individual_num_transitions ;
int num_processes ;
int num_channels ;
int channel_bits ;
int process_bits ;
int domain_bits ;
int total_bits ;



DdNode** process_bdds;
DdNode* init_state;

/*Converts 'number' to a string of 0s and 1s (the equivalent binary representation)*/
char to_bit_string(int number, int length, char* str)
{
        int i;
        for (i = length - 1; i >= 0; i--)
        {
                if(number % 2 == 1)
                        str[i] = '1';
                else str[i] = '0';
                number /= 2;
        }
        str[length] = '\0';

}

/*Parses the file spit out of the  compiler*/
int read_file (char *filename)
{
        int i, j, ch;
        char c;
        FILE *fp, *fp1;
        char str [BUF_SIZE], tmp[BUF_SIZE];
        fp = fopen(filename, "r");



        fscanf (fp, "%d", &num_processes);
        fscanf (fp, "%d", &num_channels);
        //printf("%d\n", num_processes);
        //printf("%d\n", num_channels);
        channel_bits = num_channels > 1? log (num_channels - 1) / log (2) + 1 : 1;
        process_bits = 3 + channel_bits;
        domain_bits =  num_processes * process_bits;
        total_bits = 2 * domain_bits;

        ///*Read children details*/
        children_details = (char **) malloc (num_processes * sizeof (char*));
        for (i = 0; i < num_processes; i++)
        {
                children_details[i] = (char*)malloc((num_processes + 1) * sizeof(char));
                fscanf (fp, "%s", children_details[i]);
                //printf ("%s\n", children_details[i]);
        }
```

```
        ///*Read channel details*/
        channel_details = (char **) malloc (num_channels * sizeof (char*));
        for (i = 0; i < num_channels; i++)
        {
                channel_details[i] = (char*)malloc((num_processes + 1) * sizeof(char));
                fscanf (fp, "%s", channel_details[i]);
                //printf ("%s\n", channel_details[i]);
        }

        individual_process_transitions = (char***) malloc (num_processes * sizeof(char**));
        individual_num_transitions = (int*) malloc (num_processes * sizeof(int*));
        fscanf (fp, "%s", str); /*Discard first $*/
        for ( i = 0 ; i < num_processes ; i++)
        {
                j = 0;
                while (1)
                {
                        fscanf (fp, "%s", str); /*Discard first $*/
                        if (str[0] == '$') break;
                        switch (str[0])
                        {
                                case 'N' : strcpy(tmp,"000"); break;
                                case 'C' : strcpy(tmp,"111"); break;
                                case 'P' : strcpy(tmp,"100"); break;
                                case 'T' : strcpy(tmp,"001"); break;
                        }
                        fscanf (fp, "%d", &ch); /*Read the channel*/
                        to_bit_string (ch, channel_bits, str);
                        strcat (tmp, str);

                        fscanf (fp, "%s", str); /*Discard ->*/
                        fscanf (fp, "%s", str);
                        switch (str[0])
                        {
                                case 'N' : strcat(tmp,"000"); break;
                                case 'C' : strcat(tmp,"111"); break;
                                case 'P' : strcat(tmp,"100"); break;
                                case 'T' : strcat(tmp,"001"); break;
                                default  : break; // printf("%s", str);
                        }

                        fscanf (fp, "%d", &ch); /*Read the channel*/
                        to_bit_string (ch, channel_bits, str);
                        strcat (tmp, str);


                        individual_process_transitions[i] = (char**) realloc (individual_process_transitions[i],
                                                                              (j+1) * sizeof(char*));
                        individual_process_transitions[i][j] = (char*) malloc ((process_bits * 2 + 1) * sizeof(char));
                        strcpy (individual_process_transitions[i][j], tmp);
                        //printf("%s \n", individual_process_transitions[i][j]);

                        j++;

                }
        individual_num_transitions[i] = j;
        //printf ("\n");

        }


}

/*Converts a bit string to number*/
int to_number (char *str)
{
        int i, num = 0;
        for (i = 0 ; i < strlen(str); i++)
                if (str[i] == '0')
                        num = num * 2;
                else
                        num = num * 2 + 1;
        return num;
}


/*Encodes a specific number of bits starting from the offset*/
DdNode* encode (DdManager *manager, int num_of_bits, int transition_state, int offset)
```

```
{
        int i;
        DdNode *f, *pos, *temp, *g;
        f = Cudd_ReadOne(manager);
        Cudd_Ref (f);
        for(i = offset + num_of_bits - 1; i >= offset ; i--)
        {

                    pos = Cudd_bddIthVar(manager, i);

                if (transition_state%2 == 1)
                        temp = pos;
                else
                        temp = Cudd_Not(pos);
                    Cudd_Ref (temp);

                g = f;

                f = Cudd_bddAnd (manager, g, temp);
                Cudd_Ref (f);
                   Cudd_RecursiveDeref (manager, temp);
                   Cudd_RecursiveDeref (manager, g);

                transition_state /= 2;
        }
        return f;

}

/*All set to 1*/
DdNode* get_domain_bdd (DdManager *manager, int n)
{
        DdNode *f, *pos, *g;
        int i;
        f = Cudd_ReadOne(manager);
        Cudd_Ref (f);
        for(i = 0; i <= n - 1 ; i++)
        {
                    pos = Cudd_bddIthVar(manager, i);
                    Cudd_Ref (pos);

                g = f;

                f = Cudd_bddAnd (manager, g, pos);
                Cudd_Ref (f);
                   Cudd_RecursiveDeref (manager, pos);
                   Cudd_RecursiveDeref (manager, g);
        }
        return f;
}

/* Flag = 0, first half */
/* Flag = 1,  next half  */
int permute_list (int p [], int p_count, int flag)
{
        int p_start, i, index;
        p_start = (process_bits * num_processes) * flag + p_count * process_bits;
        for (i = 0; i <= total_bits -1; i++)
                {
                        p[i]  = (i + p_start - (process_bits)* flag) % (2 * domain_bits);
                }
}

/*Get the current state as*/
DdNode* get_curr_state (DdManager *manager, int p_count, char *status_str, int c_count)
{
        int p_start;
        DdNode* status_bdd, *channel_bdd, *temp, *curr_state;
        int *p = (int *) malloc (total_bits * sizeof (int));
        p_start  = p_count * num_processes;
        status_bdd = encode (manager, 3, to_number(status_str), 0);
        channel_bdd = encode (manager, channel_bits, c_count, 3);
        Cudd_Ref (status_bdd);
        Cudd_Ref (channel_bdd);
        temp  = Cudd_bddAnd (manager, status_bdd, channel_bdd);
        Cudd_Ref (temp);
        Cudd_RecursiveDeref (manager, channel_bdd);
        Cudd_RecursiveDeref (manager, status_bdd);
```

```
        permute_list (p, p_count, 0);
        curr_state = (Cudd_bddPermute (manager, temp, p));
        Cudd_Ref (curr_state);
        Cudd_RecursiveDeref (manager, temp);
        free (p);
        return (curr_state);
}


/* Get next state */
DdNode* get_next_state (DdManager *manager, int p_count, char *status_str, int c_count)
{
        int p_start;
        DdNode* status_bdd, *channel_bdd, *temp1, *temp2, *domain, *curr_state, *q1, *q2;
        int *p = (int *) malloc (total_bits * sizeof (int));
        p_start  = p_count * num_processes;
        status_bdd = encode (manager, 3, to_number(status_str), 0);
        channel_bdd = encode (manager, channel_bits, c_count, 3);
        temp1 = Cudd_bddAnd (manager, status_bdd, channel_bdd);
        Cudd_Ref (temp1);
        Cudd_RecursiveDeref (manager, channel_bdd);
        Cudd_RecursiveDeref (manager, status_bdd);
        domain = get_domain_bdd (manager, process_bits);
        q1 = Cudd_bddAnd (manager, process_bdds[p_count], temp1);
        Cudd_Ref(q1);
        Cudd_RecursiveDeref (manager, temp1);
        q2 = Cudd_Support (manager, domain);
        Cudd_Ref(q2);
        Cudd_RecursiveDeref (manager, domain);
        temp2 =  Cudd_bddExistAbstract (manager, q1, q2);
        Cudd_Ref (temp2);
        Cudd_RecursiveDeref (manager, q1);
        Cudd_RecursiveDeref (manager, q2);
        permute_list (p, p_count, 1);
        curr_state = (Cudd_bddPermute (manager, temp2, p));
        Cudd_Ref (curr_state);
        Cudd_RecursiveDeref (manager, temp2);
        free (p);
        return (curr_state);
}

/* We know what the next state is */
DdNode* get_next_state_as (DdManager *manager, int p_count, char *status_str, int c_count)
{
        int p_start;
        DdNode* status_bdd, *channel_bdd, *temp, *curr_state;
        int *p = (int *) malloc (total_bits * sizeof (int));
        p_start  = p_count * num_processes;
        status_bdd = encode (manager, 3, to_number(status_str), process_bits);
        channel_bdd = encode (manager, channel_bits, c_count, process_bits + 3);
        Cudd_Ref (status_bdd);
        Cudd_Ref (channel_bdd);
        temp  = Cudd_bddAnd (manager, status_bdd, channel_bdd);
        Cudd_Ref (temp);
        Cudd_RecursiveDeref (manager, channel_bdd);
        Cudd_RecursiveDeref (manager, status_bdd);
        permute_list (p, p_count, 1);
        curr_state = (Cudd_bddPermute (manager, temp, p));
        Cudd_Ref (curr_state);
        Cudd_RecursiveDeref (manager, temp);
        free (p);
        return (curr_state);
}

/* The processes remain in the same state */
DdNode* replicate_domain_as_range (DdManager *manager, int p_count)
{
        int p_start, i,  j;
        char str [BUF_SIZE] ;
        DdNode *f, *g, *var, *temp, *p, *p_domain, *p_range;
        p_start = p_count * process_bits;
        f = Cudd_ReadLogicZero(manager);
        Cudd_Ref(f);

        /*Get all the transitions and replicate as next state*/
        for (i = 0; i <= individual_num_transitions[p_count]+1; i++)
        {
                if ( i == individual_num_transitions[p_count])
```

```
                        {
                                strcpy (str, "001"); /* Terminated case */
                                for ( j= 0; j < channel_bits; j++)
                                        strcat (str, "0");
                        }

                else if ( i == individual_num_transitions[p_count] + 1)
                        {
                                strcpy (str, "110"); /* Par with children case */
                                for ( j= 0; j < channel_bits; j++)
                                        strcat (str, "0");
                        }
                else
                strncpy (str, individual_process_transitions [p_count][i], process_bits);
                str[process_bits] = '\0';
                p_domain  = encode (manager, process_bits, to_number(str), p_start);
                p_range  = encode (manager, process_bits, to_number(str), domain_bits + p_start);
                temp = Cudd_bddAnd (manager, p_domain, p_range);
                Cudd_Ref (temp);
                Cudd_RecursiveDeref (manager, p_domain);
                Cudd_RecursiveDeref (manager, p_range);
                p = f;

                f = Cudd_bddOr (manager, temp, p);
                Cudd_Ref (f);
                Cudd_RecursiveDeref (manager, p);
                Cudd_RecursiveDeref (manager, temp);
        }
/*
        for (j = p_start; j < p_start + process_bits; j++)
        {
                `temp =  Cudd_bddXnor (manager, Cudd_bddIthVar (manager, j), Cudd_bddIthVar (manager, j + domain_bits));
                Cudd_Ref (temp);
                g = f;
                f = Cudd_bddAnd(manager, temp, g);
                Cudd_Ref (f);
                Cudd_RecursiveDeref(manager, temp);
                Cudd_RecursiveDeref(manager, g);
        }
*/
        return f;

}

/* Check if the state of atleast one process has changed*/
DdNode* get_non_replica (DdManager *manager)
{
        int i;
        DdNode *temp, *tmp2, *f, *g;
        f = Cudd_ReadOne(manager);
        Cudd_Ref (f);
        for (i = 0; i < domain_bits; i++)
        {
                temp =  Cudd_bddXor (manager, Cudd_bddIthVar (manager, i), Cudd_bddIthVar (manager, i + domain_bits));
                Cudd_Ref (temp);
                g = f;
                f = Cudd_bddAnd(manager, temp, g);
                Cudd_Ref (f);
                Cudd_RecursiveDeref(manager, temp);
                Cudd_RecursiveDeref(manager, g);
        }
        return f;

}

/* Precompute Deadlock states */
DdNode* get_deadlock_states (DdManager *manager)
{

int i, j, p, q;
DdNode *f, *g, *tmp1, *tmp2, *tmp3, *tmp4, *p_domain1, *p_domain2, *p_range1, *p_range2;
        f = Cudd_ReadLogicZero(manager);
        Cudd_Ref(f);
        for ( i = 0; i < num_processes; i++)
                for (j = i + 1; j < num_processes; j++)
                        for (p = 0 ; p < num_channels; p++ )
                                for (q = p + 1 ; q < num_channels; q++ )
                                        if (channel_details[p][i] == '1'
```

```
                                        && channel_details[p][j] == '1'
                                        && channel_details[q][i] == '1'
                                        && channel_details[q][j] == '1')
                                        {
                                                p_domain1 = get_curr_state (manager, i, "111", p);
                                                p_range1 =  get_curr_state (manager, j, "111", q);
                                                tmp1 = Cudd_bddAnd(manager, p_domain1, p_range1);
                                                Cudd_Ref(tmp1);
                                                Cudd_RecursiveDeref (manager, p_domain1);
                                                Cudd_RecursiveDeref (manager, p_range1);
                                                p_domain1 = get_curr_state (manager, j, "111", p);
                                                p_range1 =  get_curr_state (manager, i, "111", q);
                                                tmp2 = Cudd_bddAnd(manager, p_domain1, p_range1);
                                                Cudd_Ref(tmp1);
                                                Cudd_RecursiveDeref (manager, p_domain1);
                                                Cudd_RecursiveDeref (manager, p_range1);
                                                tmp3 =  Cudd_bddOr(manager, tmp1, tmp2);
                                                Cudd_Ref(tmp3);
                                                Cudd_RecursiveDeref (manager, tmp1);
                                                Cudd_RecursiveDeref (manager, tmp2);
                                                g = f;
                                                f = Cudd_bddOr(manager, g, tmp3);
                                                Cudd_Ref(f);
                                                Cudd_RecursiveDeref (manager, g);
                                                Cudd_RecursiveDeref (manager, tmp3);
                                        }

return f;
}



/*Adding communication constraints*/
DdNode* communication_constraints (DdManager *manager)
{
        int c_count = 0, i , j;
        DdNode * p_domain1, *p_domain2, *p_domain3, *p_domain4, *p_range1, *p_range2, *p_range3, *p_range4,
                                                *tmp1, *tmp2, *tmp3, *tmp4, *tmp5, *tmp6, *f, *g, *h, *p, *q;
        h = Cudd_ReadLogicZero(manager);
        Cudd_Ref(h);
        for (i = 0; i < num_channels; i++)
        {
                g = Cudd_ReadOne(manager);
                Cudd_Ref(g);
                for (j = 0; j < num_processes; j++)
                {
                        if (channel_details[i][j] == '1')
                        {
                                //printf("\n If true \n");
                                p_domain1 = get_curr_state (manager, j, "111", i);
                                p_range1 =  get_next_state (manager, j, "111", i);
                                p_domain2 = get_curr_state (manager, j, "110", 0);
                                p_range2 =  get_next_state_as (manager, j, "110", 0);
                                p_domain3 = get_curr_state (manager, j, "001", 0);
                                p_range3 =  get_next_state_as (manager, j, "001", 0);
                                p_domain4 = get_curr_state (manager, j, "000", 0);
                                p_range4 =  get_next_state_as (manager, j, "000", 0);
                                tmp1 = Cudd_bddAnd(manager, p_domain1, p_range1);
                                tmp2 = Cudd_bddAnd(manager, p_domain2, p_range2);
                                tmp3 = Cudd_bddAnd(manager, p_domain3, p_range3);
                                tmp4 = Cudd_bddAnd(manager, p_domain4, p_range4);
                                Cudd_Ref(tmp1);
                                Cudd_Ref(tmp2);
                                Cudd_Ref(tmp3);
                                Cudd_Ref(tmp4);
                                Cudd_RecursiveDeref (manager, p_domain1);
                                Cudd_RecursiveDeref (manager, p_range1);
                                Cudd_RecursiveDeref (manager, p_domain2);
                                Cudd_RecursiveDeref (manager, p_range2);
                                Cudd_RecursiveDeref (manager, p_domain3);
                                Cudd_RecursiveDeref (manager, p_range3);
                                Cudd_RecursiveDeref (manager, p_domain4);
                                Cudd_RecursiveDeref (manager, p_range4);
                                tmp5 =  Cudd_bddOr(manager, tmp1, tmp2);
                                Cudd_Ref(tmp5);
                                Cudd_RecursiveDeref (manager, tmp1);
```

```
                                Cudd_RecursiveDeref (manager, tmp2);
                                tmp6 =  Cudd_bddOr(manager, tmp3, tmp4);
                                Cudd_Ref(tmp6);
                                Cudd_RecursiveDeref (manager, tmp3);
                                Cudd_RecursiveDeref (manager, tmp4);
                                f = Cudd_bddOr (manager, tmp5, tmp6);
                                Cudd_Ref(f);
                                Cudd_RecursiveDeref (manager, tmp5);
                                Cudd_RecursiveDeref (manager, tmp6);
                        }
                        else
                        {
                                f = replicate_domain_as_range (manager, j);
                        }
                        p = g;
                        //printf("\nSize = %ld", Cudd_ReadSize (manager));
                        //printf("i = %d, j = %d\n", i , j);
                        fflush (stdout);
                        g = Cudd_bddAnd (manager, f, p);
                        Cudd_Ref(g);
                        Cudd_RecursiveDeref (manager, f);
                        Cudd_RecursiveDeref (manager, p);

                }
        q = h;
        h = Cudd_bddOr(manager, g, q);
        Cudd_Ref(h);
        Cudd_RecursiveDeref (manager, g);
        Cudd_RecursiveDeref (manager, q);
        }
        /*printf("Communication Constraints\n");
        Cudd_PrintMinterm (manager, h);*/
        return h;
}


/*Adding par constraints*/
DdNode* par_constraints (DdManager *manager)
{
        int i, j;
        DdNode *p_domain, *p_domain2, *p_domain3, *p_domain4, *p_range, *p_range2, *p_range3, *p_range4,
                                                        *tmp1, *tmp2, *tmp3, *tmp4, *f, *g, *h, *p, *q;
        h = Cudd_ReadLogicZero(manager);
        Cudd_Ref(h);
        for ( i = 0; i < num_processes ; i++)
        {
               int flag = 0; /*Does it have atleast one child?*/

                p_domain = get_curr_state (manager, i, "100", 0);
                p_range =  get_next_state_as (manager, i, "110", 0);
                g = Cudd_bddAnd (manager, p_domain, p_range);
                Cudd_Ref (g);
                Cudd_RecursiveDeref (manager, p_domain);
                Cudd_RecursiveDeref (manager, p_range);
                for (j = 0; j < num_processes; j++)
                        if (children_details[i][j] == '1')
                        {
                                flag = 1;
                                break;
                        }
                if (flag == 1)
                {
                        for  (j = 0; j < num_processes; j++)
                        {
                                if (i != j)
                                {
                                        if (children_details[i][j] == '1')
                                        {
                                                p_domain = get_curr_state (manager, j, "000", 0);
                                                p_range =  get_next_state (manager, j, "000", 0);
                                                f = Cudd_bddAnd (manager, p_domain, p_range);
                                                Cudd_Ref (f);
                                                Cudd_RecursiveDeref (manager, p_domain);
                                                Cudd_RecursiveDeref (manager, p_range);
                                                flag = 1;

                                        }
                                        else
```

```
                                {
                                        f = replicate_domain_as_range (manager, j);
                                }
                        /*printf("f\n");
                        Cudd_PrintMinterm (manager, f);*/
                                p = g;
                        /*        printf("\nSize = %d", Cudd_ReadSize (manager));
                                printf("i = %d, j = %d", i , j);
                                fflush (stdout); */
                                g = Cudd_bddAnd (manager, f, p);
                                Cudd_Ref (g);
                                Cudd_RecursiveDeref (manager, f);
                                Cudd_RecursiveDeref (manager, p);

                                }
                        }

                         q = h;
                        h = Cudd_bddOr(manager, g, q);
                        Cudd_Ref(h);
                        Cudd_RecursiveDeref (manager, g);
                        Cudd_RecursiveDeref (manager, q);
                }
        }
        /*printf("Par Constraints\n");
        Cudd_PrintMinterm (manager, h);*/
        return h;
}


/*Adding termination constraints*/
DdNode* termination_constraints (DdManager *manager)
{
        int i, j;
        DdNode *p_domain, *p_domain2, *p_domain3, *p_domain4, *p_range, *p_range2, *p_range3, *p_range4,
                                                                *tmp1, *tmp2, *tmp3, *tmp4, *f, *g, *h, *p, *q;
        h = Cudd_ReadLogicZero(manager);
        Cudd_Ref(h);
        for ( i = 0; i < num_processes; i++)
        {
               int flag = 0; /*Does it have atleast one child?*/

                p_domain = get_curr_state (manager, i, "110", 0);
                p_range =  get_next_state (manager, i, "100", 0);
                g = Cudd_bddAnd (manager, p_domain, p_range);
                Cudd_Ref(g);
                Cudd_RecursiveDeref (manager, p_domain);
                Cudd_RecursiveDeref (manager, p_range);
                for  (j = 0; j < num_processes; j++)
                {
                        if (i != j)
                        {
                                if (children_details[i][j] == '1')
                                {
                                        p_domain = get_curr_state (manager, j, "001", 0);
                                        p_range =  get_next_state_as (manager, j, "000", 0);
                                        f = Cudd_bddAnd (manager, p_domain, p_range);
                                        Cudd_Ref(f);
                                        Cudd_RecursiveDeref (manager, p_domain);
                                        Cudd_RecursiveDeref (manager, p_range);
                                        flag = 1;

                                }
                                else f = replicate_domain_as_range (manager, j);
                                p = g;
                                g = Cudd_bddAnd (manager, f, p);
                                Cudd_Ref(g);
                                Cudd_RecursiveDeref (manager, f);
                                Cudd_RecursiveDeref (manager, p);
                        }
                }
                if (flag == 1)
                {
                        q = h;
                        h = Cudd_bddOr(manager, g, q);
                        Cudd_Ref(h);
                        Cudd_RecursiveDeref (manager, g);
                        Cudd_RecursiveDeref (manager, q);
```

```
                }
        }
        /*printf("Termination Constraints\n");
        Cudd_PrintMinterm (manager, h);*/
        return h;
}


/*Get a BDD that represents terminal state */
DdNode* get_terminal_states (DdManager *manager)
{
        int i;
        DdNode * var, *f, *p;
        f = Cudd_ReadOne(manager);
        Cudd_Ref (f);
        for( i= 0; i <domain_bits; i++)
        {
                var = Cudd_bddIthVar (manager, i);
                if ( i != 2) /* Terminal state of process 1*/
                        var = Cudd_Not (var);
                Cudd_Ref(var);
                p = f;
                    f = Cudd_bddAnd (manager, var, p);
                Cudd_Ref(f);
                       Cudd_RecursiveDeref (manager, var);
                       Cudd_RecursiveDeref (manager, p);

        }

return f;
}

/* Dummy test function */
int test_function (DdManager *manager)
{
        DdNode *f, *g, *var, *temp;
        DdNode * var_50;
        int i;
        f = Cudd_ReadOne(manager);
        Cudd_Ref (f);
        for (i = 0; i < 100 ; i++)
        {
            var = Cudd_bddIthVar (manager,i);
            var_50 = Cudd_bddIthVar (manager, i + 100);
            temp = Cudd_bddXnor (manager, var, var_50);
           // Cudd_PrintMinterm (manager, temp);
            Cudd_Ref (temp);
            g = Cudd_bddAnd (manager, temp, f);
            Cudd_Ref (g);
            Cudd_RecursiveDeref (manager,f);
            f = g;
        }
             //printf("%d", Cudd_DagSize (f));

}

/* BDD reachability analysis */
int explore_states (DdManager *manager)
{
        int deadlock = 0;
        DdNode *deltac, *deltap, *deltat, *deltacp, *curr_states, *next_states, *tmp_next_states,
                        *explored_states, *domain_bdd, *terminal_state, *q, *r, *tmp1, *tmp2, *deadlock_states;
        int* p = (int*) malloc(total_bits * sizeof(int));
        int i;

        /*Get the permute array*/
        for (i = 0 ; i < total_bits; i++)
                p[i] = (i + domain_bits) % total_bits;

        /*Get the different constraint bdds*/
        deltac = communication_constraints (manager);
        deltap = par_constraints (manager);
        deltat = termination_constraints (manager);
        terminal_state = get_terminal_states(manager);
        deadlock_states = get_deadlock_states(manager);
        //printf("deadlock states $$$- \n");
        //Cudd_PrintMinterm (manager, deadlock_states);
```

```
/*Initialize current and explored states*/
curr_states = init_state;
explored_states = init_state;
Cudd_Ref(explored_states);

//printf("curr states $$$- \n");
//Cudd_PrintMinterm (manager, curr_states);
int step = 0;
domain_bdd = get_domain_bdd (manager, domain_bits);
do
{
        //printf("******************Step number = %d\n", step++);
        fflush(stdout);
        /* Add the curr_states to expored states*/
        q = explored_states;
        explored_states = Cudd_bddOr(manager, q, curr_states);
        Cudd_Ref(explored_states);
               Cudd_RecursiveDeref (manager, q);


        tmp_next_states = curr_states;

        /* Go as far as possible in terms of termination states */
         while         (!Cudd_IsConstant(tmp_next_states)) /*Is zero?*/
                {

                        //printf("Step number = %d $$$$$$$$$$", step++);
                        fflush(stdout);
                        curr_states = tmp_next_states;


                        /*Get the next states by adding termination constraints*/
                        tmp1 = Cudd_bddAnd (manager, curr_states, deltat);
                        Cudd_Ref(tmp1);
                        tmp2 = Cudd_Support(manager, domain_bdd);
                        Cudd_Ref(tmp2);
                        tmp_next_states = Cudd_bddExistAbstract (manager, tmp1, tmp2);
                               Cudd_Ref(tmp_next_states);
                        Cudd_RecursiveDeref (manager, tmp1);
                               Cudd_RecursiveDeref (manager, tmp2);

                        //printf("curr state \n");
                        //Cudd_PrintMinterm (manager, curr_states);
                        //printf("next state \n");
                        //Cudd_PrintMinterm (manager, tmp_next_states);

                        q = tmp_next_states;
                        tmp_next_states = Cudd_bddPermute (manager, q, p);
                        Cudd_Ref(tmp_next_states);
                               Cudd_RecursiveDeref (manager, q);

                }

        /*Now try to apply communication constraints on it*/
        //printf("curr state \n");
        //Cudd_PrintMinterm (manager, curr_states);
        tmp1 = Cudd_bddAnd (manager, curr_states, deltac);
        Cudd_Ref(tmp1);
        tmp2 = Cudd_Support(manager, domain_bdd);
        Cudd_Ref(tmp2);
        next_states = Cudd_bddExistAbstract (manager, tmp1, tmp2);
               Cudd_Ref(next_states);
        Cudd_RecursiveDeref (manager, tmp1);
               Cudd_RecursiveDeref (manager, tmp2);


        q = next_states;
        next_states = Cudd_bddPermute (manager, q, p);
        Cudd_Ref(next_states);
               Cudd_RecursiveDeref (manager, q);

        // This is wrong
        /*q = next_states;
        r = Cudd_Not(curr_states);
        Cudd_Ref(r);

        next_states = Cudd_bddAnd(manager, q, r);
```

```
Cudd_Ref(next_states);
       Cudd_RecursiveDeref (manager, q);
       Cudd_RecursiveDeref (manager, r);*/


//printf("next state $$$- \n");
//Cudd_PrintMinterm (manager, next_states);

/*If there is no communication transition*/
if(Cudd_IsConstant (next_states))
        {
                //printf("yes, it is a constant");

                /*Go back one step*/
                Cudd_RecursiveDeref(manager, next_states);
                tmp1 = Cudd_bddAnd (manager, curr_states, deltap);
                Cudd_Ref(tmp1);
                tmp2 = Cudd_Support(manager, domain_bdd);
                Cudd_Ref(tmp2);
                next_states = Cudd_bddExistAbstract (manager, tmp1, tmp2);
                       Cudd_Ref(next_states);
                Cudd_RecursiveDeref (manager, tmp1);
                       Cudd_RecursiveDeref (manager, tmp2);

                q = next_states;
                next_states = Cudd_bddPermute (manager, q, p);
                Cudd_Ref(next_states);
                       Cudd_RecursiveDeref (manager, q);
                //next_states = Cudd_bddAnd(manager, next_states, Cudd_Not(curr_states));
        }
else
        {



                tmp1 = Cudd_bddAnd (manager, next_states, deltap);
                Cudd_Ref(tmp1);
                tmp2 = Cudd_Support(manager, domain_bdd);
                Cudd_Ref(tmp2);
                tmp_next_states = Cudd_bddExistAbstract (manager, tmp1, tmp2);
                       Cudd_Ref(tmp_next_states);
                Cudd_RecursiveDeref (manager, tmp1);
                       Cudd_RecursiveDeref (manager, tmp2);

                q = tmp_next_states;
                tmp_next_states = Cudd_bddPermute (manager, q, p);
                Cudd_Ref(tmp_next_states);
                       Cudd_RecursiveDeref (manager, q);


                if(!Cudd_IsConstant (tmp_next_states)) // If we dint get false

                {
                        Cudd_RecursiveDeref(manager, next_states);
                        next_states = tmp_next_states;

                }
                else
                        Cudd_RecursiveDeref(manager, tmp_next_states);

        }
/*printf("curr state - \n");
Cudd_PrintMinterm (manager, curr_states);
printf("next state - \n");
Cudd_PrintMinterm (manager, next_states);*/

q = Cudd_bddAnd (manager, next_states, deadlock_states);
Cudd_Ref(q);
if (!Cudd_IsConstant(q)) /*Is zero? No state possible*/
        {
                printf("-----The program has a path that will deadlock-----\n");
                deadlock = 1;
                break;

        }
       Cudd_RecursiveDeref (manager, q);
```

```
                q = Cudd_Not (explored_states);
                Cudd_Ref(q);
                curr_states = Cudd_bddAnd (manager, next_states, q);
                Cudd_Ref(curr_states);
                       Cudd_RecursiveDeref (manager, next_states);
                       Cudd_RecursiveDeref (manager, q);
        }
        while (!Cudd_bddLeq(manager, curr_states, explored_states));

        if (deadlock == 0)
                                printf("-----The program has no path that will deadlock-----\n");
        free(p);


}

/* Individual transitions */
void  get_process_bdds (DdManager *manager)
{
        int i, j;
        DdNode * temp, *p;
        process_bdds = (DdNode**) malloc (num_processes*sizeof(DdNode*));

        for ( i = 0; i < num_processes; i++)
        {
                process_bdds[i] = Cudd_ReadLogicZero (manager);
                Cudd_Ref (process_bdds[i]);
                for (j = 0; j < individual_num_transitions[i]; j++)
                        {
                                temp = encode (manager, 2 * process_bits, to_number (individual_process_transitions[i][j]), 0);
                                Cudd_Ref (temp);
                                p = process_bdds[i];
                                process_bdds[i] = Cudd_bddOr (manager, temp, p);
                                Cudd_Ref (process_bdds[i]);
                                    Cudd_RecursiveDeref (manager, temp);
                                    Cudd_RecursiveDeref (manager, p);

                        }
        }


}

/* Get the initial state of the processes */
int get_init_states (DdManager *manager)
{
        int i;
        DdNode *temp, *p;
        char *tmp_str = (char*)  malloc (process_bits * sizeof(char));
        tmp_str[process_bits] = '\0';
        init_state = Cudd_ReadOne (manager);
        Cudd_Ref(init_state);
        for ( i = 0; i < num_processes; i++)
        {
                if(i == 0)
                strncpy(tmp_str, individual_process_transitions[i][1],process_bits);
                else
                strncpy(tmp_str, individual_process_transitions[i][0],process_bits);
                tmp_str[process_bits]= '\0';
                temp = encode (manager, process_bits, to_number (tmp_str), i * process_bits);
                p = init_state;
                init_state = Cudd_bddAnd (manager, temp, p);
                Cudd_Ref (init_state);
                    Cudd_RecursiveDeref (manager, temp);
                    Cudd_RecursiveDeref (manager, p);


        }


}

/* Detect deadlocks */
int detect_deadlocks (DdManager *manager)
{
        get_init_states (manager);
        get_process_bdds (manager);
```

```
        explore_states (manager);

}


/*Main program*/
int main (int argc, char**argv)
{
        if (argc <= 1)
        {
                printf("Usage: bdd-deadlock <filename.dat>\n");
                exit(1);
        }

        DdManager *manager;
        DdNode * temp;
        read_file (argv[1]);
        manager = Cudd_Init(total_bits,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);
        Cudd_AutodynEnable(manager, CUDD_REORDER_SIFT_CONVERGE);
        detect_deadlocks (manager);
        cuddGarbageCollect (manager, 1);
}
```