UPPSALA
UNIVERSITET

IT 12 036
Examensarbete 15 hp
September 2012

Implementing verification of concurrent systems using Binary Decision Diagrams

Tomas Sävström

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Implementing verification of concurrent systems using Binary Decision Diagrams

Tomas Sävström

Verification of programs through the use of formal methods have become popular as it can guarantee the programs correct. Concurrent programs have always been hard to test because of the nature of the program, as these programs are used in many branches of the software industry formal methods to prove these programs correct have been developed.

In this report one of these methods are first described and then implemented using Binary Decision Diagrams to see if this would lead to an optimization of the current implementation of the method.

Handledare: Lukas Holik
Ämnesgranskare: Parosh Abdulla
Examinator: Olle Gällmo
IT 12 036
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

The ever-present problem when designing and implementing programs is that there normally is no way to be sure that the program is working correctly unless the input space for the program is small enough to test every input. Programs are extensively tested to find faults, the only problem with it is that testing can only find fault not show the absences of them, so regardless of the amount of testing there is no guarantee that the program works correctly for all inputs.

Because of this, ways to prove programs correct through the use of model-checking and state space exploration have been developed and used on a number of programs, see [12]. Among those hardest to test are concurrent programs, as faults can appear at very specific circumstances that depend on the operating system process scheduler. These faults might not appear at all during testing but appear on the customer machines after release of the software. Because of this nature, the faults are also very hard to reproduce which makes the debugging process more difficult. Especially hard is it when the number of processes is a parameter of the system.

The main problem with trying to prove such programs correct is that the amount of processes is undecided and can be any number. Normally, this means that the program is needed to be proven correct for all values of n, where n is the number of processes in the system, but because n can have any value, this is not possible. Methods to prove these kinds of programs correct can be found in [1,3,4]. In this work, we will concentrate on method [1], where the problem is solved by identifying a cutoff point which guarantees the program correct for any value of n without having to test for larger n. This cutoff point is a value of n for which 2 criteria holds. This will be explained in 3.6.

The method internally manipulates k-tuples of states of processes, so in the case where the programs contain many states that the processes can be in, this will lead to a large number of possible k-tuples. The value of k is the main problem, since one has at most S^k of k-tuples, where S is the number of states. This problem is called the "state explosion problem" and causes a bottleneck in the method because it has to handle so many possible states. This problem normally appears when dealing with reachable configurations of concurrent programs. To alleviate this problem Binary Decision Diagrams can be used to symbolically encode the state space so that not every individual k-tuple is enumerated. The BDD works like a member function, i.e. given a k-tuple the BDD will give true if the k-tuple is in the set and false otherwise.

2 Parametric system

The purpose of this algorithm is to verify that a given system, modelled as described below, will not end up in a state that is considered bad in that system. Below the definition of the system model will be given. All definitions below come from [1]

2.1 System model

A system is modeled as follows:

> A parameterized system is a pair $\mathcal{P} = (Q, \Delta)$ where Q is a finite set of *local states* of a process and Δ is a set of *transition rules* over Q. A transition rule is either *local* or *global*. A local rule is of the form $s \to s'$ where process changes its local state from s to s' independently of the local states of the other processes. A global rule is of the form **if** $\mathbb{Q}S$ **then** $s \to s'$, where $\mathbb{Q} \in \{\exists_L, \exists_R, \exists_{LR}, \forall_L, \forall_R, \forall_{LR}\}$ and $S \subseteq Q$.
>
> The parameterized system $\mathcal{P}$ above induces a *transition system* $\mathcal{T} = (C, \to)$ where $C = Q^\star$ is the set of its *configurations* and $\to \subseteq C \times C$ is the *transition relation*. The transtion relation $\to$ contains a transation $c \to c'$ iff $c = c_L\, s\, c_R, c' = c_L\, s'\, c_R$, and one of the two conditions holds: (i) Δ contains a local rule $s \to s'$, or (ii) Δ contains a global rule **if** $\mathbb{Q}S$ **then** $s \to s'$, and one of the following conditions is satisfied:
>
> - $\mathbb{Q} = \exists_L$ and $c_L \in Q^\star S Q^\star$ or $\mathbb{Q} = \exists_R$ and $c_R \in Q^\star S Q^\star$.
> - $\mathbb{Q} = \exists_{LR}$ and $c_L c_R \in Q^\star S Q^\star$.
> - $\mathbb{Q} = \forall_L$ and $c_L \in S^\star$ or $\mathbb{Q} = \forall_R$ and $c_R \in S^\star$.
> - $\mathbb{Q} = \forall_{LR}$ and $c_L c_R \in S^\star$.
>
> A configuration, c_n, is reachable from a given set, X, if there is a succession of rules, $c_0 \to c_1, c_1 \to c_2, \ldots, c_{n-1} \to c_n$ where $c_0 \in X$.

An example is a model of a very simple mutual exclusion algorithm. In this system only one process can be in the critical section at one time and the processes can be in one of the following states, waiting or critical. If one would describe a parameterized system like this using the model above it would be like follows, $\mathcal{P} = (Q, \Delta)$ where $Q = \{Wa, Cr\}$ and $\Delta = \{$ if $\forall_{LR} Wa$ then $Wa \to Cr, Cr \to Wa\}$. From this parameterized system the transition system, $\mathcal{T} = (C, \to)$, is induced where $C = \{Wa, Cr\}^+$ and $\to$ would contain the following transitions for different k, where k is the size of the configurations:

- $k = 1$:
  - $\rightarrow_1 = \{Wa \rightarrow Cr, Cr \rightarrow Wa\}$
- $k = 2$:
  - $\rightarrow_2 = \{WaWa \rightarrow CrWa, WaWa \rightarrow WaCr, CrWa \rightarrow WaWa, WaCr \rightarrow WaWa\}$
- ...

This way to model is then used in the verification problem that the algorithm in 3.6 will try to solve. This verification problem is: Given a set of initial configurations, a set of bad configurations, and a parameterized system, are the bad configurations reachable from the initial configurations in the induced system? In other words is it possible to get to a bad configuration starting from a configuration in the initial set? For the verification algorithm to work, 3 functions are needed. Theses are the subword function, the post function and the concretization. They are defined in the next section.

3 Verification Algorithm

For this section we fix a parameterized system $\mathcal{P} = (Q, \Delta)$ and a transition system $\mathcal{T} = (C, \rightarrow)$, both defined above, that are used in the definitions. We first give necessary definitions needed to present the algorithm and then define the algorithm.

3.1 Subword

Let u and S be words, where $S = s_1 \dots s_n$, u is the a subword to S iff $u = s_{i_1} \dots s_{i_k}, 1 \leq i_1, i_k \leq n$ and $i_j < i_{j+1}$ for all $1 \leq j \leq k$.

What this means is pretty straightforward, u is a subword to S if u can be created by picking letters from $s_1, \dots, s_n$ and then putting them in the order $s_1 < s_2 < \dots < s_n$.

3.2 Abstraction

Given a configuration, c, $\alpha_k(c)$ stands for the set containing all subwords of c of size k

3.3 Post-image

We define the *post-image* of a set $X \subseteq C$ to be the set $post(X) := \{c' | c \to c' \wedge c \in X\}$.

What this means is that $post(X)$ will contain all configurations that can be reached by applying one of the transitions in $\to$ to the configurations in X.

3.4 Concretization

For every $k \in \mathbb{N}$, let $\mathcal{V}_k$ be the set of words of length k over Q.The *concretization* function $\gamma : \cup_{0 \leq k} 2^{\mathcal{V}_k} \to 2^C$ inputs a set of views V where $V \subseteq \mathcal{V}_k$ for some k, and returns the set of configurations (of sizes larger than or equal to k) that can be reconstructed from the views in V. In other words, $c \in \gamma(V) \iff \alpha_k(c) \subseteq V \wedge |c| \geq k$.

What this means is that $\gamma(V)$ will contain all configurations which have all subwords of size k in V, where k is the size of configurations in V. $\gamma_k(V)$ then stands for all configurations in $\gamma(V)$ of size k.

3.5 Abstract Post-image

$Apost_k(V) = \alpha_k(post(\gamma(V)))$, where $V \subseteq \mathcal{V}_k$.

3.6 Verification Algorithm

This algorithm works with a set of initial configurations, I, a set of bad configurations, Bad, and a parameterized system $\mathcal{P} = (Q, \Delta)$ and the task is to verify that starting from the configurations in I any configuration in Bad can not be reached by using the rules in Δ. This algorithm is not guaranteed to terminate as the problem is undecidable. It has been shown in [1] that the algorithm returns safe if none of the bad configurations can be reached and return unsafe if they can be reached.

$\mathcal{R}_n$ is defined the following way:

For $n \in \mathbb{N}$ we use $\mathcal{R}_n$ to denote the set of reachable configurations of length n and $\mathcal{R}$ to denote the set of all reachable configurations.

There are three criteria that are needed to be fulfilled for the algorithm to be able to work. These are:

- (**Computing** $\alpha_k(I)$) That $\alpha_k(I)$ can be computed .

- (**Exsistens of saftey threshold**) That there is a known saftey threshold, θ, which means that any configuration of size k where $k \geq \theta$ can be decided if it is bad or not.
- (**Computing** $Apost_k(V)$) That $Apost_k(V)$ can be calculated for any $V \subseteq \mathcal{V}_k$. Computing $Apost_k(V)$ is a central component if the verification algorithm. This is generally a hard problem as it involves computing the effects of applying the successor function post to the set $\gamma(V)$, which is infinite as can be seen in the definition. As shown in [1], for the class of systems we work with, it is sufficient to consider only those configurations in $\gamma(V)$ whose sizes are up to $k + l$, where l is some small constant that depends on the precise form of the transition rules. This set is finite so the post image can be computed.

Where α_k and $Apost_k$ are defined above.

The algorithm works in the way that for the current value of n it calculates all reachable configurations from I, the initial configurations, and checks if any of these configurations are in Bad. If any one these are in Bad, Unsafe is returned as the system can reach a configuration considered bad. If this is not the case all subwords of $\mathcal{R}_n$ of size k are added to V_k and then it starts checking if has reached the cutoff point.

To be sure that the cutoff point is reached $\mathcal{R} \subseteq \gamma(V_k)$ must be fulfilled, here $\mathcal{R}$ stands for the set of all reachable configurations. To show this $\gamma(V_k)$ is needed to fulfil the conditions of an inductive invariant. The two conditions that are needed for a set, S, to be an inductive invariant are:

- (S covers I) 1: $I \subseteq S$
- (S is stable under Post) 2: $post(S) \cup S = S$

So to test that $\gamma(V_k)$ is an inductive invariant these conditions are tested. Condition 1 is tested by, $\alpha_k(I) \subseteq V_k$ on line 10 in the pseudocode above this works because $\alpha_k(I) \subseteq V_k$ iff $I \subseteq \gamma(V_k)$. Condition 2 is tested by $Apost_k(V_k) \subseteq V_k$ also on line 10. The reason for this test to satisfy condition 2 is because $post(\gamma(V_k)) \cup \gamma(V_k) \subseteq \gamma(Apost(V_k)) \cup \gamma(V_k)$ gives that if $Apost(V_k) \cup V_k = V_k$ then $post(\gamma(V_k)) \cup \gamma(V_k) = \gamma(V_k)$. In other words, because $Apost$ overapproximates $post$, this leads to that if V_k is stable under $Apost$, then $\gamma(V_k)$ is stable under $post$.

The algorithm is defined the following way:

```
1  Algorithm: Verifcation Procedure
2  for n = θ to ∞ do
3  |  compute R_n
4  |  if R_n ∩ Bad ≠ ∅ then
5  |  |  return Unsafe
6  |  end
7  |  V_n := ∅
8  |  for k = θ to n do
9  |  |  V_k := V_k ∪ α_k(R_n)
10 |  |  if α_k(I) ⊆ V_k ∧ Apost_k(V_k) ⊆ V_k then
11 |  |  |  return Safe
12 |  |  end
13 |  end
14 end
```

4 Binary Decision Diagrams

Binary Decision Diagrams are defined the following way in [2]:

> A *Binary Decision Diagram (BDD)* is a rooted, directed acyclic graph with
>
> - one or two terminal nodes of out-degree zero labelled 0 or 1, and
> - a set of variable nodes u of out-degree two. The two outgoing edges are given by two functions $low(u)$ and $high(u)$. A variable $var(u)$ is associated with each variable node.
>
> A BDD is *Ordered* (OBDD) if on all paths through the the graph the variables respect a given linear order $x_1 < x_2 < \ldots < x_n$. An (O)BDD is *Reduced* (R(O)BDD) if
>
> - (**uniqueness**) no two destinct nodes u and v have the same variable name and low- and high-successor, i.e.,
>
> $var(u) = var(v), low(u) = low(v), high(u) = high(v)$ implies $u = v$,
>
>   and
> - (**non-redundant tests**) no variable node u has identical low-high-successor, i.e.,
>
> $low(u) \neq high(u)$.

In other words this means that a ROBDD is a binary tree where each leaf is one of the terminals 0 or 1 and each other node is associated with a variable in the formula that the ROBDD represents and has two children. It also fulfils the ordering set, the uniqueness and non-redundant tests criteria. The two children are the ROBDDs representing the variable being set to true and to false. The child that represent the variable u being false is the $low(u)$ edge and representing u being true is $high(u)$ edge. For example take the formula $(x_0 \wedge x_1) \vee x_2$. If we have the ordering $x_0 < x_1 < x_2$ the root of the ROBDD would be the node representing x_0 so the $high(x_0)$ edge would lead to the subtree representing the formula $x_1 \vee x_2$, as $1 \wedge x_1 = x_1$, and the $low(x_0)$ edge would lead to the subtree representing the formula x_2, as $0 \wedge x_1$ is 0 regardless of x_1 and $0 \vee x_2 = x_2$.

These ROBDDs can be handled as if they were formulas and combined using any binary logical operator, the pseudocode for how this is done can be found in [2].

5 Problem description

The problem that is to be solved by this project is to implement the algorithm defined in 3.6 using ROBDD to symbolically encode sets. The reason for this project is to try to avoid the bottleneck described in the introduction by not having to enumerate every configuration. The first part that is needed is how to encode the sets using ROBDDs, as can be seen in 3.6 what is then needed to implement is functions to create ROBDDs that encode the configurations, of size k, in the sets I, Bad and Δ , described above. What is also needed to implement is α_k, γ_k and $post$ described above. As sets are needed to run the three functions the functions to create the set will be implemented first and tested, after that each function will be implemented and tested separately and then all these will be combined to implement the algorithm in 3.6. In the following section we first describe how each part is implemented using ROBDDs which are then combined to create the algorithm.

6 Solution

To begin with how to encode sets using ROBDDs is described then rest of the work is divided into 6 sub-problems. These were creating a ROBDD modelling the initial configurations, I, creating a ROBDD modelling the transition rules, Δ, creating a ROBDD modelling the bad configurations, Bad, creating the $post$ function, creating the α_k function, and creating the γ_k function. These six parts are then combined to implement the algorithm in 3.6. In the following sections how these are implemented will be described and pseudocode for them will be given. When implementing we used Cudd for handling the ROBDDs. All im-

plementations are using Cudd that can be found at [6].

6.1 Encoding using ROBDDs

ROBDDs are in the project used to symbolically encode configurations and rules, defined in 2.1. They are encoded the following way:

Process State: A processes state s is encoded using a set of ROBDD variables, $b_0, b_1, \ldots, b_n$ where n is depended on the size of Q, defined in 2.1, n is the smallest n so that $2^n \geq SizeOf(Q)$. The variables b_0 to b_n are set to binary encode the value of the state s is in, where 0 = false and 1 = true.

Configuration: A configuration $s_0 s_1 \ldots s_k$ is encoded by the formula $s_0 \wedge s_1 \wedge \ldots \wedge s_k$ where s_i is an encoded process state defined above with the variables index by i. This means that each s_i will have there variables renamed from $b_0, b_1, \ldots, b_n$ to $b_0^i, b_1^i, \ldots, b_n^i$. So the formula that X that encodes a set of configurations will look like $c_0 \vee c_1 \vee \ldots \vee c_i$, where c_0 to c_i are encoded configurations as defined above.

Rule: A rule $s_0 s_1 \ldots, s_k \rightarrow s'_0 s'_1, \ldots, s'_k$ is encoded by the formula $s_0 \wedge s_1 \wedge \ldots \wedge s_k \wedge s'_1 \wedge s'_2 \wedge \ldots \wedge s'_k$, where s_i is an encoding of a process state that have the variables, b_j, renamed to b_j^i and s'_i is an encoding of a process state that have the variables, b_j, renamed to $b_j^{i'}$. So the formula that T encodes that encodes a set of rules will look like $r_0 \vee r_1 \vee \ldots \vee r_n$, where r_0 to r_n are rules as defined above.

For example take $Q = \{0, 1, 2, 3\}$ of process states would be encoded as:

- 0: $\neg b_0 \wedge \neg b_1$
- 1: $\neg b_0 \wedge b_1$
- 2: $b_0 \wedge \neg b_1$
- 3: $b_0 \wedge b_1$
- $Q : (\neg b_0 \wedge \neg b_1) \vee (\neg b_0 \wedge b_1) \vee (b_0 \wedge \neg b_1) \vee (b_0 \wedge b_1)$

A set $X = \{01, 23\}$ of configurations would be encoded as:

- 01: $\neg b_0^0 \wedge \neg b_1^0 \wedge \neg b_0^1 \wedge b_1^1$
- 23: $b_0^0 \wedge \neg b_1^0 \wedge b_0^1 \wedge b_1^1$
- $X : (\neg b_0^0 \wedge \neg b_1^0 \wedge \neg b_0^1 \wedge b_1^1) \vee (b_0^0 \wedge \neg b_1^0 \wedge b_0^1 \wedge b_1^1)$

A set $T = \{01 \rightarrow 11, 00 \rightarrow 01\}$ of rules would be encoded as:

- $00 \rightarrow 01$: $\neg b_0^0 \wedge \neg b_1^0 \wedge \neg b_0^1 \wedge \neg b_1^1 \wedge \neg b_0^{0'} \wedge \neg b_1^{0'} \wedge \neg b_0^{1'} \wedge b_1^{1'}$, call it r_0
- $01 \rightarrow 11$: $\neg b_0^0 \wedge \neg b_1^0 \wedge \neg b_0^1 \wedge b_1^1 \wedge \neg b_0^{0'} \wedge b_1^{0'} \wedge \neg b_0^{1'} \wedge b_1^{1'}$, call it r_1
- $T : r_0 \vee r_1$

6.2 CreateI

The result of this function is the ROBDD that encodes the initial configurations of size k of the system. The input for this function is an integer array, $init$, that has the size of Q for the model in 2.1 and fulfils that the sum of the integers in the array do not exceed k. If $init[i] >= 0$ this means that there should be $init[i]$ processes in state i in the initial configurations, if $init[i] < 0$ this means that there should be $init[i]$ or more processes in state i in the initial configurations.

```
1  Algorithm: CreateI
   Data: init, k
   Result: The ROBDD encoding the initial configurations of size k
2  CanHaveMore ← 0
3  Reslut ← ROBDD(true)
4  for i ← 0 to init.Length − 1 do
5  |  if init[i] > 0 then
6  |  |  for j ← 1 to init[i] do
7  |  |  |  AddNode(Result, i)
8  |  |  |  k ← k − 1
9  |  |  end
10 |  end
11 |  if init[i] < 0 then
12 |  |  CanHaveMore ← CanHaveMore + 1
13 |  |  for j ← init[i] + 1 to 0 do
14 |  |  |  AddNode(Result, i)
15 |  |  |  k ← k − 1
16 |  |  end
17 |  end
18 end
19 if CanHaveMore > 0 then
20 |  AddNodes(Result, init, k)
21 end
22 return Result
```

Here AddNode adds a process in state i to the initial configurations and AddNodes add processes in $init$ that can have more in all possible combination of amount.

6.3 CreateT

The result of this function is the ROBDD that encodes all the rules of size k from the Δ set described above in 2.1, these are given as an array containing rules. The function works in such a way that it goes through all configurations in Q^k, described in 2.1. For each configuration it looks if a rules left side matches, if it does the right side of the rule is calculated, combined with the current configuration and added to the result.

For example if the configuration $x_0x_1x_2$ and the rule $x_0 \rightarrow x_0'$ is currently being processed this would match. This would lead to the construction of the right side which will be $x_0'x_1x_2$ which would be combined with the configuration to give $x_0x_1x_2x_0'x_1x_2$ that would then be added to the result.

```
1  Algorithm: CreateT
   Data: Rules, k
   Result: The ROBDD encoding all rules in → of size k
2  AllPossible ← Q^k
3  Reslut ← ROBDD(false)
4  for i ← 0 to AllPossible.Size − 1 do
5  |  Current ← Next(AllPossible)
6  |  for j ← 0 to Rules.length − 1 do
7  |  |  if Matches(Current, Rule[j]) then
8  |  |  |  AddRule(Result, Rule[j])
9  |  |  end
10 |  end
11 end
12 return Result
```

Here $Next$ gives the next configuration in the set and $Matches$ gives true if the rule can be applied to the configuration.

6.4 CreateBad

The result of this function is the ROBDD that encodes all configurations of size k that are considered bad. The input for this function is an integer array, *bad*, that has the size of Q for the model in 2.1 and have that $\forall i : bad[i] \geq 0$. Here $bad[i]$ stands for the least number of processes needed to be in state i for a configuration to be considered bad. The function works much like CreateT in that it goes through all configurations in Q^k and adds the configurations that matches the information from *bad* to the result.

For example let *bad* have $bad[0] = 1$, $bad[1] = 2$, $bad[2] = 1$ and $bad[3] = 0$ and the current configuration be x_0, x_1, x_2, x_3, x_1 this would match the information in *bad* and would be added to the result, but if the configuration would have been x_0, x_1, x_2, x_3, x_3 this would not have matched as there is only one x_1

in the configuration.

```
1  Algorithm: CreateBad
   Data: bad, k
   Result: The ROBDD encoding all bad configurations of size k
2  AllPossible ← Q^k
3  Reslut ← ROBDD(false)
4  for i ← 0 to AllPossible.Size − 1 do
5  |  Current ← Next(AllPossible)
6  |  if MatchesArray(Current, Bad) then
7  |  |  AddConfig(Result, Current)
8  |  end
9  end
10 return Result
```

Here $Next$ gives the next configuration in the set and $MatchesArray$ gives true if the configuration fulfils the conditions in the array.

6.5 Post

The result of this function is given a ROBDD, T, encoding the rules of size k from $\rightarrow$, defined in 2.1, and a ROBDD, X, containing configurations of size k, it will return $post(X)$, defined in 3.3. The way this is done is by first doing $T \wedge X$ and then remove the first $x_0, x_1, \ldots, x_k$ from the result and rename the remaining $x_{k+1}, x_{k+2}, \ldots, x_{2*k}$ to $x_0, x_1, \ldots, x_k$. The reason it works is because T and X are encoded in the way described in 4. Because of this $T \wedge X$ will contain all rules that can be applied to X this as only rules that have a left side in X will still be in $T \wedge X$.

```
1  Algorithm: Post
   Data: T, X, k
   Result: The ROBDD encoding post(X)
2  Result ← T ∧ X
3  Temp ← RemoveVariablesUpTo(Result, k)
4  Result ← RenameVariables(Temp, k)
5  return Result
```

6.5.1 RemoveVariablesUpTo (Used in post)

The result of this function is given a ROBDD, $root$, and an integer, n, what will be returned is a new ROBDD that is $root$ with all nodes x_i where $i < n$ removed. This is done by going through the tree and on each node that has $i < n$ the node, x_i, is removed by replacing it with $RemoveVariablesUpTo(x_i.true, n) \vee RemoveVariablesUpTo(x_i.false, n)$ and if the node has $i \geq n$ it is returned. The reason this works is because the ROBDD has the ordering $x_o < x_1 < \ldots <$

x_k making the nodes to be removed appear first.

Given the ROBDD *root* that encodes the formula $\varphi_{root}[x_0, x_1, \ldots, x_k]$ for a set of configurations of size k, as defined in 4, where $k \geq n$. What is returned is the ROBDD that encodes the formula $\varphi_R = \exists_{x_0} \exists_{x_1} \ldots \exists_{x_n} \varphi_{root}$ of configurations of size k.

```
1 Algorithm: RemoveVariablesUpTo
  Data: root, n
  Result: The ROBDD root with the variables, x_i, with index(x_i) < n
          removed
2 if Constant(root) then
3 |  return root
4 end
5 if GetIndex(root) < n then
6 |  return RemoveVariablesUpTo(root.then, n) ∨
  |  RemoveVariablesUpTo(root.else, n)
7 end
8 return root
```

6.6 Subword

The result of this function is to given a set of configurations of size n encoded in a ROBDD X and integer $k \leq n$, it will return the ROBDD encoding $\alpha_k(X)$ defined in 3.2. This is done by calculating all ways to pick k processes from the n given. For each of way to pick k from n all processes not picked are removed from X and the rest renamed to $x_0, x_1, \ldots, x_k$ creating a new ROBDD X_i, where i goes from 0 to $\binom{n}{k}$. The result returned is then $X_0 \vee X_1 \vee, \ldots, \vee X_{\binom{n}{k}}$.

For each new way to pick k processes from n and the ROBDD X encoding the formula $\varphi_X[s_0, s_1, \ldots, s_n]$ a ROBDD is created that encodes the formula $\varphi_{S_i}[s_{i_0}, s_{i_1}, \ldots, s_{i_k}]$ where $i_0, i_1, \ldots, i_k$ are in $\{0, 1, 2, \ldots, n-1, n\}$ and $i_0 < i_1 < \ldots < i_k$ what is returned is then $\varphi_{Result} = \varphi_{S_0} \vee \varphi_{S_1} \vee \ldots \vee \varphi_{S_{\binom{n}{k}}}$.

```
1  Algorithm: Subword
   Data: k, n, V
   Result: The ROBDD encoding α_k(V)
2  Result ← ROBDD(false)
3  Possible ← α_n({0,1}^⋆)
4  ToRemove ← new integer array of size n-k
5  for i ← 0 to size.Possible − 1 do
6  |  Current ← Next(Possible)
7  |  if Contains(k, Current, 1) then
8  |  |  ToRemove ← IndexOfZeros(Current, ToRemove)
9  |  |  Temp ← RemoveVariablesGiven(V, ToRemove)
10 |  |  Result ← Result ∨ Rename(Temp, k)
11 |  end
12 end
13 return Result
```

Here $Contains$ returns true if Current contains k number of 1:s and $IndexOfZeros$ put the indexes of the zeros in Current in ToRemove.

6.6.1 RemoveVariablesGiven (Used in subword)

The result of this function is given a ROBDD, $root$, with n variables and an integer array, $ToRemove$, of size $k \leq n$, a new ROBDD is created that is $root$ with all nodes, x_i, that have index in $ToRemove$ removed and the rest renamed to $x_0, x_1, \ldots, x_{n-k}$.

Given the ROBDD $root$ that encodes the formula $\varphi_{root}[x_0, x_1, \ldots, x_k]$ for a set of configurations of size k, as defined in 4, where $k \geq n$, where $n = size(ToRemove)$. What is returned is the ROBDD that encodes the formula $\varphi_R = \exists_{x_{ToRemove[0]}} \exists_{x_{ToRemove[1]}} \ldots \exists_{x_{ToRemove[n]}} \varphi_{root}[x'_0/x_0, x'_1/x_1, \ldots, x'_{k-n}/x_{k-n}]$ of configurations of size k, where $x'_0, x'_1, \ldots, x'_{k-n}$ are the x_i that fulfil $\forall j$: $ToRemove[j] \neq i$.

```
Algorithm: RemoveVariablesGiven
  Data: root, ToRemove
  Result: The ROBDD root with the variables, x_i, in ToRemove removed
if Constant(root) then
    return root
end
Index ← GetIndex(root);
if In(Index, ToRemove) then
    return RemoveVariablesGiven(root.then, ToRemove) ∨
    RemoveVariablesUpTo(root.else, ToRemove)
else
    return
    (Rename(Index) ∧ RemoveVariablesGiven(root.then, ToRemove)) ∨
    ((¬Rename(Index) ∧ RemoveVariablesGiven(root.else, ToRemove))
end
```

6.7 Concretization

The result of the function is given a set of configurations, encoded in a ROBDD X, of size k and an integer, n, it will return $\gamma_n(X)$ defined in 3.4. It does this by going through all ways to pick k processes from n. For each way to pick, $x'_0, x'_1, \ldots, x'_k$, a new ROBDD is created, C_i where i goes from 0 to $\binom{k}{n}$, with the variables $x_0, x_1, \ldots, x_k$ renamed to $x'_0, x'_1, \ldots, x'_k$. The result is then $C_0 \wedge C_1 \wedge \ldots \wedge C_{\binom{k}{n}}$.

For each new way to pick n processes from k a ROBDD is created that encodes the formula $\varphi_{C_i}[s_{i_0}, s_{i_1}, \ldots, s_{i_n}]$ where $i_0, i_1, \ldots, i_n$ are in $\{0, 1, 2, \ldots, k-1, k\}$ and $i_0 < i_1 < \ldots < i_n$ what is returned is then $\varphi_{Result} = \varphi_{C_0} \wedge \varphi_{C_1} \wedge \ldots \wedge \varphi_{C_{\binom{k}{n}}}$.

```
Algorithm: Concretization
  Data: k, n, V
  Result: The ROBDD encoding γ_n(V)
Result ← ROBDD(true)
Possible ← α_n({0,1}^⋆)
for i ← 0 to size.Possible − 1 do
    Current ← Next(Possible)
    if Contains(k, Current, 1) then
        Result ← Result ∧ RenameToIndexies(Current, V)
    end
end
return Result
```

Here $Next$ gives the next configuration in the set and $Contains$ returns true if Current contains k number of 1:s

7 Experiment results

In this section, the times measured will be given and then discussed in the next section. The algorithm tested is PNCSAcover and comes from [11]. The PNCSAcover algorithm was chosen because it contains fairly many states and rules and was something that the previous implementation did not mange to verify. We expect the process to finish for $n = 9$, with the result of the PNCSAcover algorithm being unsafe as was seen in [5]. All times below are measured when runing the algorithm in 3.6 for different values of n. As can be seen below, the algorithm runs out of memory when trying to run createT for $n = 5$ because of this the other functions times stop at $n = 4$. For concretization and subword, n is the size of the modelled system and k is the size of configurations that should be returned. The example we are running comes from example 4.7 of Alain Finkel, The minimal coverability graph for Petri nets, Proc. APN'93, LNCS 674 according to [11].

Function times in seconds					
n	CreateT	CreateI	CreateBad	Create$\mathcal{R}_n$	Post
2	< 1	< 1	< 1	< 1	< 1
3	2	< 1	< 1	< 1	< 1
4	220	< 1	1	2	< 1
5	unable to allocate memory	-	-	-	-

Table 1: Function times

Create$\mathcal{R}_n$	
n	NumberOfPosts
2	1
3	53
4	57
5	-

Table 2: Post Count

Subword		
n	k	Time(sec)
2	2	< 1
3	3	< 1
4	4	< 1

Table 3: Times for subword

Concretization		
n	k	Time(sec)
2	3	< 1
3	4	< 1
4	5	< 1

Table 4: Times for concretization

8 Result discussion

As can be seen in the results post, subword and concretization are fast which is positive as concretization and post where the main problem with time in the previous implementation. The reason for this is the use of ROBDDs which leads to the implementations in 6, which use ROBDD operations only and never deal with individual configurations at a low level. This shows that using ROBDDs have a good impact on the time spent on these functions. The current implementation of concretization will work regardless of how larger you want the new ROBDD. As the algorithm only uses this to create ROBDDs of one size bigger, this function could be optimized by implementing it only to work in this case.

As can also be clearly seen by the results is that createT is a bottleneck, this is because the way it is implemented goes through all possible k-tuples which was what we wanted to get away from. The reason for this implementation is that it was the most straightforward one and a better one has not been found so far. There should be a better way of doing this so that individual configurations are not needed to be taken in to account and this is what would be needed to be fixed in the future to make this implementation more viable. One idea of how to do this for local rules would be to start by creating a ROBDD encoding the formula $(s_0 \Leftrightarrow s_k) \wedge (s_1 \Leftrightarrow s_{k+1}) \wedge \ldots \wedge (s_{k-1} \Leftrightarrow s_{2k-1})$, and then change one place at a time so that the rule is created.

The current way to implement createBad will also go through all possible k-tuples which would lead to poor performance. A way to do this without going through all possible k-tuples should also exist and is something that would further increase performance. The current implementation is like this because again it was the most straightforward and a better way was not found.

The current way to implement createI will create all configurations that fulfils the criteria and it works quite fast, this part might be able to improve but if that is the case it would probably not lead to a big difference.

9 Conclusion

The conclusion that can be drawn from this project is that ROBDDs are very good to use when creating new sets that are combinations of already existing sets. The way that new sets are created from configurations and rules in this project are not different from what was previously implemented and still suffer the same problems. But as it is more efficient to combine the sets using the needed functions this way to implement the verification algorithm would be faster if a better way is found.

10 Refrences

1. P. A. Abdulla, F. Haziza and L. Holík : All for the Price of Few - (Parametrized Verication through View Abstraction), prepared for submition to VMCAI'12

2. H. R. Andersen. Binary Decision Diagrams. Department of Information Technology, Technical University of Denmark, Lyngby, Denmark, 1997. Lecture notes for 49285 Advanced Algorithms E97, Springer

3. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized verifcation with automatically computed inductive assertions. In: Proc. 13th Int. Conf. on Computer Aided Verifcation. Lecture Notes in Computer Science, vol. 2102, pp. 221-234 (2001), Springer

4. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized verifcation of a cache coherence protocol: Safety and liveness. In: Proc. VMCAI 2002. pp. 317-330 (2002), Springer

5. A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In CAV '10: Proc. 20th Int. Conf. on Computer Aided Verification, volume 6174 of LNCS. Springer, 2010.

6. Tomáš Vojnar, Lecture Notes, Formal Analysis and Verification http://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/fav-lecture-04.pdf

7. Fabio Somenzi, Manual, CUDD: CU Decision Diagram Package http://vlsi.colorado.edu/~fabio/CUDD/

8. Jackie Rice, Tutorial, CUDD http://www.cs.uleth.ca/~rice/cudd.html

9. Ethan L. Schreiber, Tutorial, CUDD http://www.cs.ucla.edu/~ethan/documents/schreiber_cudd_tutorial.pdf

10. University of California, Tutorial, BLIF http://www.cs.uic.edu/~jlillis/courses/cs594/spring05/blif.pdf

11. Verification Group – Université Libre de Bruxelles, Examples, PNCSAcover http://www.ulb.ac.be/di/ssd/ggeeraer/eec/

12. J. B. Almeida, M. J. Frade, J. S. Pinto, S. M. D. Sousa : Rigorous Software Development - An Introduction to Program Verification, Springer London Ltd, England, 2011.