Efficient Symbolic Model Checking of Software Using Partial Disjunctive Partitioning

Sharon Barner and Ishai Rabinovitz

IBM Haifa Research Laboratory, Haifa, Israel

Abstract. This paper presents a method for taking advantage of the efficiency of symbolic model checking using disjunctive partitions, while keeping the number and the size of the partitions small. We define a restricted form of a Kripke structure, called an *or-structure*, for which it is possible to generate small disjunctive partitions. By changing the image and pre-image procedures, we keep even smaller *partial disjunctive partitions* in memory. In addition, we show how to translate a (software) program to an or-structure, in order to enable efficient symbolic model checking of the program using its disjunctive partitions. We build one disjunctive partition for each state variable in the model directly from the conjunctive partition of the same variable and independently of all other partitions. This method can be integrated easily into existing model checkers, without changing their input language, and while still taking advantage of reduction algorithms which prefer conjunctive partitions.

1 Introduction

Symbolic model checking suffers from the known problem of state explosion. This explosion usually happens while performing the image or pre-image computation. In order to cope with this problem, symbolic model checkers use partitioned transition relations [8]. Using ordered conjunctive partitioning [7] is quite simple and sometimes allows early quantification while computing the image or pre-image; this serves to decrease the needed memory.

The RuleBase model checker [1] uses ordered conjunctive partitioning, and previous work showed its application to general purpose software [5,6]. In this paper, we show how disjunctive partitioning can be used to increase the efficiency of symbolic model checking for software.

Disjunctive partitioning, first introduced in [8], has several advantages over conjunctive partitioning. First, both image and pre-image computations are more efficient using disjunctive partitions, since quantification distributes over disjunction but not over conjunction [9,8]. For the same reason, distributed model checking using disjunctive partitions is also more scalable than using conjunctive partitioning, since each process can do the quantification on its own. As a result, the "heavy" computation is divided by the number of processes.

Despite the advantages of disjunctive partitioning, use of the technique is generally hindered by the difficulty in building the partitions. The method presented in [8] is efficient only for asynchronous circuits. It builds the disjunctive partitions using an interleaving model, which allows only one wire to change its value at a time.

Both [2] and [4] suggested how to build disjunctive partitions for synchronous circuits. In [2], we see how to decompose an FSM into smaller FSMs, and then use this decomposition to split the conjunctive partitioned transition relation into a disjunction of conjunctive partitioned transition relations. In [4], a set of mutually exclusive events is used to decompose the behavior of the circuit to disjunctive partitions. Large disjunctive partitions are split into conjunctive partitions, which results in a DNF partitioning as in [2]. Both methods need additional information on the circuit in order to get a good decomposition.

Disjunctive partitioning is also used in [10], where each transition is a separate disjunctive partition. The contribution of [10] is in presenting the order in which the transitions should be executed in order to achieve improved performance.

While all the above works are applicable to models generated for software, applying them to software is problematic. The method of [8] is applicable to parallel software, but does not decompose each process to disjunctive partitions. On the other hand, [10] creates a large number of disjunctive partitions. The methods of [2] and [4] are not automated and require additional information from the user. We introduce a new method applicable to software models in which the decomposition is generated automatically, without additional information from the user. The number of disjunctive partitions created is similar to that of the conjunctive partitions for the same model, and the BDD size of the disjunctive partitions is comparable to that of the conjunctive partitions.

Software has the feature that in each step there is little change in the program variables. It is quite easy to build a model for software where each step changes only the pc (program counter), and at most, one additional state variable. We present a modeling language called ODL, which is natural for defining such models. We also present a method for translating from conjunctive partitions to disjunctive partitions and vice versa. These translations can be easily adapted by any symbolic model checker that uses conjunctive partitioning and by doing so, may benefit from the advantages of disjunctive partitioning.

In the traditional image computation algorithm, each disjunctive partition must represent the next value for all variables, so the disjunctive partition of state variable x should indicate the change of x and pc, and the fact that all other variables keep their value. The latter information might severely impact the BDD size of the partition. In this work, we change the image and pre-image computation in such a way that they can work on the *partial disjunctive partition* of x, which represents only the changes of x and pc, and not the fact that all other variables keep their value. Using this algorithm decreases the BDD size needed to represent the disjunctive partitions and improves the image computation. This method is applicable not only for software models, but also to some other methods ([8], [2] and [4]) based on the fact that only a subset of the variables in the model can change their value in each disjunctive partition.

Finally we suggest two schemes for distributed model checking that use the disjunctive partitioning.

In our work we implemented the translation from conjunctive partitioned transition relation to disjunctive partitioned transition relation. We show that the size of the partial disjunctive partitions is equal to, or even smaller than, the size of the conjunctive partitions. In addition, we show that calculating reachability analysis using disjunctive partitions significantly outperforms calculation using conjunctive partitions.

The remainder of this paper is structured as follows: Section 2 states the preliminaries. Section 3 presents the generation of the model from the software and the ODL modeling language. Section 4 presents the translation between conjunctive and disjunctive partitions, and vice versa. Section 5 introduces partial disjunctive partitions and their advantages, and Section 6 presents the distributed version. In Section 7 we present some experimental results. We conclude and suggest some directions for future work in Section 8.

2 Preliminaries

A finite program can be modeled by a Kripke structure M over a set of atomic propositions AP. $M = (S, S_0, R, L)$, where S is a finite set of states, S_0 is a set of initial states, $R \subseteq S \times S$ is a total transition relation, and $L : S \to 2^{AP}$ is a labeling function that labels each state with the set of atomic propositions that are true in that state. The states of the Kripke structure are coded by a set of state variables $\bar{v}$. Each valuation to $\bar{v}$ is a state in the structure. Model checking is a technique for verifying finite state systems represented as Kripke structures. The basic operations in model checking are the *image computation* and the *pre-image computation*. Given a set of states S and a transition relation R, represented in symbolic model checking by the BDDs $S(\bar{v})$ and $R(\bar{v}, \bar{v}')$ respectively, the image computation finds the set of all states related by R to some state in S and the pre-image computation finds the set of all states such that some state in S is related to them by R. More precisely, $image(S(\bar{v}), R(\bar{v}, \bar{v}')) = \exists \bar{v}(S(\bar{v}) \wedge R(\bar{v}, \bar{v}'))$ and $pre\_image(S(\bar{v}'), R(\bar{v}, \bar{v}')) = \exists \bar{v}'(S(\bar{v}') \wedge R(\bar{v}, \bar{v}'))$. The result of $image(S(\bar{v}), R(\bar{v}, \bar{v}'))$ is over $\bar{v}'$. In order to get the result over $\bar{v}$, all BDD variables are “unprimed”.

A conjunctive partitioned transition relation is composed of a set of partitions and_R_i such that $R(\bar{v}, \bar{v}') = \bigwedge_i and\_R_i(\bar{v}, \bar{v}')$. In case each state variable can be described by a single conjunctive partition (as in this work), we have that $and\_R_{v_i} = (v_i' = f_{v_i}(\bar{v}))$ and thus each partition is a function of $\bar{v}$ and v_i' rather than $\bar{v}$ and $\bar{v}'$. The image computation in this case is $image(S(\bar{v})) = \exists \bar{v}(S(\bar{v}) \wedge (\bigwedge_{v_i} and\_R_{v_i}(\bar{v}, v_i')))$.
Computing $\exists x A(\bar{v})$ is referred to as quantifying x out of A. Early quantification [8] can make image and pre_image computations even more efficient. Early quantification is done by quantifying a variable x out of the intermediate BDD result, after conjuncting the last conjunctive partition that is dependent on x. Quantifying a variable out of the intermediate BDD may reduce the size of the BDD and as a result make the image computation easier.

A disjunctive partitioned transition relation is composed of a set of disjunctive partitions or_R_i such that $R(\bar{v},\bar{v}') = \bigvee_i or\_R_i(\bar{v},\bar{v}')$. In the case where each state variable can be changed only in a single disjunctive partition, we have that $or\_R_{v_i} = (v_i' = f_{v_i}(\bar{v})) \wedge (\forall y \neq v_i : \ \ y = y')$. The image computation when using disjunctive partitions is done by calculating $image(S(\bar{v})) = \exists \bar{v}(S(\bar{v}) \wedge (\bigvee_{v_i} or\_R_{v_i}(\bar{v},\bar{v}')))$. Because existential quantification distributes over disjunction, we have that every quantification is "early", and thus $image(S(\bar{v})) = \bigvee_{v_i} \exists \bar{v}(S(\bar{v}) \wedge or\_R_{v_i}(\bar{v},\bar{v}'))$. Because the quantification is done "early" for every v in the disjunctive partitioning, all intermediate BDD results depend only on $\bar{v}'$, while when using conjunctive partitions the intermediate BDD results may depend both on $\bar{v}$ and $\bar{v}'$. Thus, using disjunctive partitions usually results in smaller intermediate BDDs than when using conjuncting partitions.

Note that as opposed to a conjunctive partition, the naive disjunctive partition is dependent on the entire vector $\bar{v}'$, rather than just a single v_i'. We return to this point later and show how to avoid it by modifying the image computation.

Let $A \subseteq S$ be a set of states and let $\bar{x}$ be a set of variables. We use the notation $A|_{\bar{x}}$ to indicate the projection of the set A onto $\bar{x}$. That is:
$A|_{\bar{x}} = \{s \in S |\ \exists a \in A$ such that s and a agree on all values of the variables in $\bar{x}\}$.

3 Generating a Model from Software

Previous work showed the application of symbolic model checking to general purpose software [5,6] by translating C source code to EDL (Environment Description Language), a dialect of SMV [9], which is the input language to the RuleBase model checker. EDL, like SMV, is naturally suited for building of conjunctive partitions. That previous work was based on a specially-built parser and was limited to a small subset of C. In this work, we build a similar model using a full-blown compiler front-end. The most important thing about this model is that it has the following structure.

Definition 1. *An* or-structure *is a Kripke structure in which for every two states s, s': if $R(s,s')$ then s and s' are different from each other only in the values of the pc and no more than a single additional state variable x.*

The model we build has a state variable for each global variable in the C code and a state variable named pc (program counter) that holds the value of the next statement to be performed. The model also has stacks to support local variables, functions and recursion, and some special variables to support arrays and pointers (without pointer arithmetic). The basics of the generation process are explained here using a simple example. Afterward, we will discuss the special treatment for pointers and arrays.

The translation process first translates the C code to intermediate code. There are two reasons for using intermediate code: 1. It will ease the support of other input languages in the future. 2. It generates the *pc* in a way such that for each value of *pc*, a maximum of one memory location changes its value. One

```
1:   z = 0;
2:   while(x > 0) {
3:        z+ = 5;
4:        x − −;
5:   }
```

(a) C code of div.c

```
18 : r1 ← 0
19 : z ← r1
22 :
24 : r2 ← x
26 : pc ← (r2 > 0)?29 : 27
27 : pc ← 53
29 :
35 : r3 ← z
37 : r4 ← 5
39 : r5 ← r3 + r4
41 : z ← r5
44 : r6 ← x
46 : r7 ← r6 − 1
48 : x ← r7
50 : pc ← 22
53 :
```

(b) The intermediate code of div.c

Fig. 1. Example of translation from C to intermediate code.

may object to using intermediate code because it increases the number of values pc can get, and therefore increases the number of states in the model. While this is true, the number of pc values is only multiplied by a small factor and herefore adds to the state variables only 2 or 3 bits, which are negligible.

In Figure 1.a we can see a fragment of a C program. The code has two global variables called x and z. This code is translated to an assembly-like intermediate code shown in Figure 1.b. In the intermediate code, there is a list of instructions, each with a unique pc (program counter), listed at the beginning of each line. The pc is updated to the pc of the next line if not specified otherwise. The first two lines indicate the behavior for $pc = 18$ and $pc = 19$. This is the intermediate code generated for line 1 in the C code ($z = 0$). At $pc = 18$ the value 0 is inserted to $r1$, and $r1$ is inserted to z in $pc = 19$. Lines like the one for $pc = 22$, which don't have any code, are used as jump targets and only update the pc to the pc of the next line. Lines for $pc = 24$ through 27 perform the while condition: first in $pc = 24$ x is inserted into $r2$ and then in $pc = 26$ it is checked if it is bigger than 0. A true answer sets the pc to 29 (enter the loop), while a false answer sets it to the pc of the next line, which in turn sets the pc to 53 - after the loop.

Next we translate the intermediate code into a model. There are two possible translations: The first one is to translate the intermediate code to a language that has the style of a guarded transition system. Each transition is of the form: $pc = PC_1 \Longrightarrow (X \leftarrow f(X, Y, Z) \wedge pc \leftarrow PC_2)$. The guard is always a condition about the value of the pc (each value of the pc has exactly one transition) and the transition changes the value of the pc and perhaps the value of one additional

```
♯define main_r1 0
♯define main_r2 x
♯define main_r3 z
♯define main_r4 5
♯define main_r5 main_r3 + main_r4
♯define main_r6 x
♯define main_r7 main_r6 − 1

pc = 19 ⟹ (z ← main_r1 ∧ pc ← 22)
pc = 22 ⟹ (pc ← 26)
pc = 26 ⟹ (pc ← if (main_r2 > 0)
                        then 29 else 27)
pc = 27 ⟹ (pc ← 53)
pc = 29 ⟹ (pc ← 41)
pc = 41 ⟹ (z ← main_r5 ∧ pc ← 48)
pc = 48 ⟹ (x ← main_r7 ∧ pc ← 50)
pc = 50 ⟹ (pc ← 22)
```

(a) Model in ODL representation

```
♯define main_r1 0
♯define main_r2 x
♯define main_r3 z
♯define main_r4 5
♯define main_r5 main_r3 + main_r4
♯define main_r6 x
♯define main_r7 main_r6 − 1

next(pc) ← case
  pc = 19 : 22
  pc = 22 : 26
  pc = 26 : if (main_r2 > 0)
               then 29 else 27
  pc = 27 : 53
  pc = 29 : 41
  pc = 41 : 48
  pc = 48 : 50
  pc = 50 : 22
  else : pc
esac;
next(x) ← case
  pc = 48 : main_r7
  else : x
esac;
next(z) ← case
  pc = 19 : main_r1
  pc = 41 : main_r5
  else : z
esac;
```

(b) Model in EDL representation

Fig. 2. Example of div.c translation to EDL and ODL.

state variable [1]. We refer to this language as ODL. The translation to ODL is presented in Figure 2.a. The other possibility is to translate the intermediate code to EDL (Figure 2.b). For both possibilites we model the registers using a $\sharp define$. In this way, the registers won't use any bits in the model. This is possible because the intermediate code defines and uses each register only once.

The translation to ODL is very simple. Each line in the intermediate code is translated to a guarded expression representing the changes for this value of the pc. For example in $pc = 19$, z gets $main\_r1$ (the $\sharp define$ that represents register $r1$), and pc is set to 22. In the EDL code, we need to gather all the assigns of a state variable to the same place. For instance, the code for next(z) includes assignments for the lines for pcs 19 and 41 of Figure 1.b. Another difference is that in ODL it is implicit that every state variable that is not mentioned, keeps its value, while the EDL explicitly codes it.

At first glance, it seems preferable to translate to ODL because it's simpler to translate C code to ODL, and it is simpler to translate ODL to disjunctive

[1] Note that this transition may change a different variable depending on the value of other state variables. However, only one state variable will change its value at any one time. For instance, an assignment of the form $a[i] = 5$ will change a[0] or a[1], etc., depending on the value of i. But only one array location will change at any one time.

partitions. But translating the C code to EDL allows us to use RuleBase to read EDL, build the conjunctive partitions, and perform pre-model-checking *reductions*. A reduction is simply a conservative abstraction, that is, one that preserves both positive and negative truth values. Conjunctive partitions are more natural for performing simple reductions such as constant propagation as well as other more sophisticated reductions performed by RuleBase. Thus, even if we did not have conjunctive partitions, we would want to build them and translate the result of the reduction back to disjunctive partitions. Thus, we present methods for translating from conjunctive to disjunctive partitioning and vice versa in order to enable flexibility in our tool. In practice, using the reductions and translating the reduced conjunctive partitioning to disjunctive partitioning indeed proved to be useful. In addition, analyzing the translations enables us to bound the size of the disjunctive partitions, with respect to the conjunctive partitions.

3.1 Dealing with Pointers and Arrays

Modeling pointers and arrays creates a problem, because in general an assignment to a variable X from an array or a pointer causes the variable X to be dependent on more memory locations than an assignment from a scalar. In a naive approach, the BDD size of the partition for X will be quite large, because of the dependence on multiple variables. Furthermore, the large number of variables in a single partition results in many constraints on the BDD order for the entire model, which might result in a larger BDD size not just for the partition in question, but for the entire design.

We solve this problem by using cut-points [3]. Our translation adds four variables for each array . For array ar we add: l_index_{ar}, l_array_{ar}, r_index_{ar} and r_array_{ar} (the prefix l/r means that the array is in the left/right side of the assignment). We translate an assignment $x = ar[i]$ to the three assignments described in Figure 3(a), and an assignment $ar[i] = x$ to the three assignments described in Figure 3(b).

When using this translation on code containing assignments $x = ar[i]$; $x = ar[j]$; $y = ar[i]$; $y = ar[j]$;, we get that r_index_{ar} is dependent on i and j, r_array_{ar} is dependent on r_index_{ar} and all ar cells, and x and y are dependent only on r_array_{ar}. Without cut-points, we would have had that both x and y are dependent on i, j and all cells of array ar.

In pointers, the problem is even more severe because there are generally more memory locations that can be affected by a pointer dereference than cells in an array. Still, the same idea is useful for pointers.

Note that using cut-points and $\sharp defines$ for modeling registers causes a problem when translating statements like $x = a[i] + a[j]$. We avoid this problem by splitting such statements into two: $temp = a[i];\ x = temp + a[j]$.

Our translation has another attribute. An assignment such as $a[a[i]] = 5$ is translated in the intermediate code into two different accesses to the array, one to get $a[i]$ and the second to assign to $a[a[i]]$, so that our translation creates the code in Figure 3(c).

$$\begin{aligned} &r\_index_{ar} = i \\ &r\_array_{ar} = ar[r\_index_{ar}] \\ &x = r\_array_{ar} \end{aligned}$$

(a) Translating $x = ar[i]$

$$\begin{aligned} &l\_index_{ar} = i \\ &l\_array_{ar} = x \\ &ar[l\_index_{ar}] = l\_array_{ar} \end{aligned}$$

(b) Translating $ar[i] = x$

$$\begin{aligned} &r\_index_{a} = i \\ &r\_array_{a} = a[r\_index_{a}] \\ &l\_index_{a} = r\_array_{a} \\ &l\_array_{a} = 5 \\ &a[l\_index_{a}] = l\_array_{a} \end{aligned}$$

(c) Translating $a[a[i]] = 5$

Fig. 3. Translation of array expressions

3.2 Splitting of Self-Assignment Statements

Assignments statements in the code can be of two kinds:

1. *Self-assignment statement* - Assignment to a variable x in which the assigned value is a function of x (e.g., $x+ = y$ or $x = x + w + z$). Such an assignment can be further divided into two kinds: *constant self-assignment statement* where we update the variable with a constant (e.g., $x* = 4$, $x + +$), and *variable self-assignment statement* (e.g. $x+ = y$, $x = x * b + c$).
2. *Foreign-assignment statement* - Assignment to a variable x in which the assigned value is not dependent on the value of x. (e.g. $x = y$ or $x = w + z$).

In order to reduce BDDs size and achaive better performance we split variable self-assignment statements like $x+ = y$ into two: $temp = x$, $x = temp + y$. This split increases the number of *pc* values and adds one variable (for all splits) but improves the overall performance. The reason will be explaind in section 4.1. Constant self-assignment statements can remain as is.

4 Translating between Disjunctive and Conjunctive Partitions

In this section, we show how to build the disjunctive partition of a state variable x, $or\_R_x(\bar{v}, \bar{v}')$, from its conjunctive partition $and\_R_x(\bar{v}, x')$ and vice versa. Our construction is applicable only to or-structures where each dereference, such as arrays and pointers, is broken by a cut-point. Let *pc* be the state variable that codes the program counter of the program and $\bar{y}$ be the state variables which are different from *pc* and x.

Definition 2. *$dep\_states_x(\bar{v})$ is a set of states such that for every $s \in dep\_states_x(\bar{v})$ there exists s' such that $R(s, s')$ and x has different values in s and s'.*

Intuitively, $dep\_states_x(\bar{v})$ are all the states related to lines in the C program where x is assigned a value, except for the case where x is assigned the same value it had before the assignment.

Definition 3. *$dep\_pcs_x(pc)$ is the set of pc values which are related to statements in which x may change* [2].

Definition 4. *The* partial disjunctive partition *of a state variable x, denoted by $por\_R_x(pc, x, \bar{y}, x', pc')$, is the disjunctive partition $or\_R_x(\bar{v}, \bar{v}')$ without the requirement that the variables in $\bar{y}$ are left unchanged.*
$(or\_R_x(\bar{v}, \bar{v}') = por\_R_x(pc, x, \bar{y}, x', pc') \wedge (\bar{y} = \bar{y}'))$

4.1 Building Disjunctive Partitions from Conjunctive Partitions

We now show how to build each disjunctive partition from the conjuctive partition of the same state variable and the conjuctive partition of pc.

Translation for $x \neq pc$: First we show how to build $or\_R_x(\bar{v}, \bar{v}')$ for $x \neq pc$.

1. Calculate $dep\_states_x(\bar{v})$:
$$dep\_states_x(\bar{v}) = \exists x'(and\_R_x(\bar{v}, x') \wedge (x \neq x')).$$
2. Intersect the quantification of x from $dep\_states_x(\bar{v})$ with the conjunctive partitions of x and pc:
$$por\_R_x(pc, x, \bar{y}, x', pc') =$$
$$= (\exists x(dep\_states_x(\bar{v}))) \wedge and\_R_x(\bar{v}, x') \wedge and\_R_{pc}(\bar{v}, pc')$$
3. Intersect $por\_R_x(\bar{v}, x', pc')$ with $\bar{y} = \bar{y}'$ to indicate that the other variables do not change:
$$or\_R_x(\bar{v}, \bar{v}') = por\_R_x(pc, x, \bar{y}, x', pc') \wedge (\bar{y} = \bar{y}')$$

We use $dep\_states_x(\bar{v})$ in our construction and not $dep\_pcs_x(pc)$ because two states in which the pc value is identical do not necessarily change the same state variable. For example, consider the C statement $a[i] = 5$ and assume that it is related to $pc = 7$. For each value of i this statement changes a different state variable. Thus, the value $pc = 7$, which is related to this statement, will be in more than one disjunctive partition. If we had used $dep\_pcs_x(pc)$ the state $\{pc = 7; i = 2\}$ would have been both in the partition of $a[2]$ and $a[1]$. As a result, after conjuncting the disjunctive partition of $a[1]$ with $\bar{y} = \bar{y}'$ it would have contained another transition, that does not exist in the original model and changes only pc and not $a[1]$ or $a[2]$. This transition would have been entered to the disjunctive partition of $a[1]$ because $a[2]$ is in $\bar{y}$. The quantification that appears in $por\_R_x(pc, x, \bar{y}, x', pc')$ is discussed in detail later.

[2] x may not always change its value in a certain pc. For example, when x is a cell in an array, $a[0]$, and the assignment is $a[i] = 5$, a[0] is assigned a value only if $i = 0$ and stays unchanged otherwise.

Translation for pc: Calculating $por\_R_{pc}(pc, x, \bar{y}, pc')$ is a bit different.

1. Calculate $dep\_pcs_x(pc)$ for each $x \neq pc$:
$$dep\_pcs_x(pc) = dep\_states_x(\bar{v})|_{pc}$$
2. Calculate the set of pc values $jump\_pcs(pc)$ that are related to statements in which pc is the only state variable that is changed. These pc values are related to statements in which there is a control branch like an if statement.
$$jump\_pcs(pc) = \bigwedge_{x \neq pc} (\overline{dep\_pcs_x(pc)})$$
3. Intersect $and\_R_{pc}(\bar{v}, pc')$ with $jump\_pcs(pc)$ to get the value of pc' for this pc value.
$$por\_R_{pc}(pc, x, \bar{y}, pc') = jump\_pcs(pc) \wedge and\_R_{pc}(\bar{v}, pc')$$
4. Intersect $por\_R_{pc}(pc, x, \bar{y}, pc')$ with $\bar{y} = \bar{y}'$, where $\bar{y}$ is all variables that are different from pc.
$$or\_R_{pc}(\bar{v}, \bar{v}') = por\_R_{pc}(pc, x, \bar{y}, pc') \wedge (\bar{y} = \bar{y}')$$

Discussion: The general idea is that transitions in which only the pc changes should be in the partition of the pc, and transitions in which both the pc and some variable x change should be in the partition of x. Naively, this means that a line with some assignment would appear in the partition of the variable being assigned, while a line without an assignment would appear in the partition of the pc. However, things are not so simple. Consider the assignment $x = 5$. If x has the value 5 before the assignment, then a transition from this line changes only the pc. If x has another value before the assignment, then this line changes both x and the pc. A naive construction of the or-partitions from the and-partitions would put the transition from a state where x has the value 5 into the partition of the pc, rather than into the partition of x. We would like to put this transition into the partition of x, because in this way the BDDs will be in some sense "cleaner" - that is, we hope that the BDD size will be smaller. Two other related problems are the case of assignments of the form $x+ = y$, where y has the value 0, and the case of assignments to $a[i]$ for some array a, where i is out of the array bounds. Our method deals with such cases as explained below.

In order to deal with assignments of the form $x = 5$ we quantify x out of $dep\_states_x(\bar{v})$ before conjuncting the result of the quantification with $and\_R_x(\bar{v}, x')$ and $and\_R_{pc}(\bar{v}, pc')$. By doing so we ignore the value of x before the assignment.

The need to deal with assignments of the form $x+ = y$ (for which $y = 0$ may cause a problem in a naive construction) is the source of the splitting of variable self-assignment statements into two, as described in 3.2 above. This way, we avoid dealing with such assignments in the construction itself.

The problem with assignments such as $a[i] = 5$ needs some explanation. Consider an array $a[0..2]$ of size three and a statement $a[i] = 5$, where i equals 7. Because $a[7]$ is not a real variable in the program, there is no corresponding state variable in our model (otherwise the model would have been unbounded). Thus, in such a case, in our model only the *pc* is changed, and the conjunctive partitioned transition relation contains a transition which changes only the *pc*. But this statement is related to transitions that do change variable values (for $i < 3$), and thus does not "belong" in the partition for the *pc* (according to our notion of "cleanness"). It is possible to overcome this problem by adding a new $overflow$ variable to the model, the disjunctive partition of which will capture this behavior.

Finally, we note that in the general case, our translation does not work for statements such as $a[i] = a[5]$ or $a[a[i]] = a[i] + 1$. However, when the model is generated, as we suggested in section 3.1, such statements are always split up into several statements and therefore the problem is avoided.

4.2 Building Conjunctive Partitions from Partial Disjunctive Partitions

We previously discussed how to build a disjunctive partition from a conjunctive partition. In this subsection, we present the translation in the opposite direction.

1. We first calculate $dep\_pcs_x(pc)$ simply by looking at the *pc*s that appear in $por\_R_x(\bar{v}, x', pc')$

$$dep\_pcs_x(pc) = por\_R_x(\bar{v}, x', pc')|_{pc}$$

2. Now we can calculate $and\_R_x(\bar{v}, x')$. It is formed from a union of two sets: the states in which x changes its value and the states in which x saves its value.

$$and\_R_x(\bar{v}, x') = (\exists pc'(por\_R_x(\bar{v}, x', pc')) \vee (\overline{dep\_pcs_x(pc)} \wedge x = x')$$

3. Now we can calculate $and\_R_{pc}(\bar{v}, pc')$. It is calculated by gathering the transition pc to pc' in all the partial disjunctive partitions of the variables and conjuncting it with $por\_R_{pc}(\bar{v}, pc')$.

$$and\_R_{pc}(\bar{v}, pc') = por\_R_{pc}(\bar{v}, pc') \vee (\bigvee_{x \neq pc} por\_R_x(\bar{v}, pc', x')|_{(pc,pc')})$$

5 Using Partial Disjunctive Partitions

In the previous section, we showed how to calculate disjunctive partitions. Using this, we can take advantage of the superior efficiency of disjunctive partitioning. However, if the sizes of the disjunctive partitions are larger than the corresponding conjunctive partitions it is not certain that we have gained anything. In this section we examine the answer to this question. First let's look at $or\_R_x(\bar{v}, \bar{v}')$. By definition, $or\_R_x(\bar{v}, \bar{v}') = por\_R_x(pc, x, \bar{y}, pc', x') \wedge (\bar{y} = \bar{y}')$. It is possible to build an example in which $|or\_R_x(\bar{v}, \bar{v}')| = O(n \cdot |por\_R_x(pc, x, \bar{y}, pc', x')|)$, where n is the number of state variables. An example is the assignment $x \leftarrow y$, where x is the first variable in the BDD order after pc and y is the last state variable in the BDD order.

In order to avoid this factor, we do not calculate $or\_R_x(\bar{v}, \bar{v}')$. We calculate only $por\_R_x(pc, x, \bar{y}, x', pc')$ and rewrite the procedures that calculate image and pre_image operations in such a way as to use $por\_R_x(pc, x, \bar{y}, x', pc')$ instead of $or\_R_x(\bar{v}, \bar{v}')$. In the next subsection, we present the new algorithm for image and pre_image computation and prove its correctness. After, that we will bound the size of $por\_R_x(pc, x, \bar{y}, x', pc')$.

5.1 Image and Pre_Image Computations Using Partial Disjunctive Partitions

When computing image(pre_image) using disjunctive partitions, it is possible to calculate the image(pre_image) on each disjunctive partition independently and then union the results. In this subsection, we introduce how to compute image or pre_image when only $por\_R_x(pc, x, \bar{y}, pc', x')$ is given for each variable x.

Lemma 1. $pre\_image(S(pc', x', \bar{y}'), or\_R_x(pc, x, \bar{y}, pc', x', \bar{y}')) =$

$$pre\_image(S(pc', x', \bar{y}), por\_R_x(pc, x, \bar{y}, pc', x'))$$

From this lemma, we get a simple algorithm that in the first step unprimes $\bar{y}'$ in $S(pc', x', \bar{y}')$ (linear in the size of the BDD), and then performs the ordinary pre_image algorithm on the result. The proof of this lemma is given in the full version of this paper.

Lemma 2. $image(S(pc, x, \bar{y}), or\_R_x(pc, x, \bar{y}, pc', x', \bar{y}')) =$

$$image(S(pc, x, \bar{y}'), por\_R_x(pc, x, \bar{y}', pc', x'))$$

Here again, we have a simple algorithm. First prime $\bar{y}$ in $S(pc, x, \bar{y})$ and in $por\_R_x(pc, x, \bar{y}, pc', x')$ and then calculate the image using the results. The proof is almost the same as of the previous lemma.

5.2 Bounding the Size of the Partial Disjunctive Partitions

In this subsection, we bound the size of partial disjunctive partitions. The proofs of these claims are long, technical, and tedious. Proof sketches are given in the full version of this paper. Despite the relatively large upper bound, in practice, these extreme examples are rare. See Section 7 for experimental results.

Since every variable is dependent on pc, it seems wise to place pc as the first state variable in the BDD ordering. All of the following lemmas assume that the BDD ordering follows this idea.

We define $\widehat{por\_R}_x(\bar{v}, x')$ to be $por\_R_x(\bar{v}, x', pc')$ without the condition on the value of pc':

$$\widehat{por\_R}_x(\bar{v}, x') = (\exists x(dep\_states(x, \bar{v}))) \wedge and\_R_x(\bar{v}, x').$$

We can now rewrite the definition of $por\_R_x(\bar{v}, x', pc')$ using $\widehat{por\_R}_x(\bar{v}, x')$:

$$por\_R_x(\bar{v}, x', pc') = \widehat{por\_R}_x(\bar{v}, x') \wedge and\_R_{pc}(\bar{v}, pc').$$

The following lemmas will first bound the size of $\widehat{por\_R}_x(\bar{v}, x')$ and only then the size of $por\_R_x(\bar{v}, x', pc')$.

6 Scalability for Distributed Model Checking

We now turn to the scalability of disjunctive partitioning. We claim that symbolic model checking with disjunctive partitioning is not only more efficient than with conjunctive partitioning, it also scales better. This is a direct result of the fact that quantification distributes over disjunctive partitioning, but not over conjunctive partitioning. Since $image(S(\bar{v})) = \bigvee_x \exists \bar{v}(S(\bar{v}) \wedge or\_R_x(\bar{v}, \bar{v}'))$, when using disjunctive partitions or partial disjunctive partitions we can calculate the image using one partition on each processor including quantification and then union the results of all processors. Because the image computation may be exponential in the number of BDD nodes and the union operation is linear in the number of BDD nodes, distributing the partitions between n processors divides the "heavy" work by n. Note that when image computation is done distributively using conjunctive partitions it requires another step in which the partial results are "anded" together before quantification. Thus, the work done after all the processors have calculated their results may still be exponential in the number of BDD nodes. We now suggest two distributed algorithms for disjunctive partitions. The first algorithm is simple and uses a master and several slaves. The master will send $S(\bar{v})$ to all the slaves and start sending each idle slave a disjunctive partition. Each slave that gets a disjunctive partition will perform the image computation with this partition and union it with previous computations it made. When there are no more partitions and all slaves are idle, the master will gather all the slaves' results and union them. Reachability computation is then performed by repeated image computations of the former algorithm. One drawback with this scheme is that while the server computes the union of all the slaves' results, the slaves are idle.

The second algorithm avoids this problem. In this algorithm, each process P_i is responsible for several partitions TR_i, and has its own reachability set RS_i. There is also a (shared) queue of sets of states and each process has two pointers to this queue: a shared pointer for entering sets to the queue and a private one for reading from the queue. As a result, all processors read all the sets that enter the queue. At the beginning the queue has the initial set of states. Each process P_i, at each iteration takes the next set S from the queue (according to its pointer), removes from it the parts it already handled $S = S \setminus RS_i$ and adds the result to

Example	Num of vars	Conjunctive partitions		Disjunctive partitions	
		Reachability time	Maximal step time	Reachability time	Maximal step time
simple	505	11024 s	95.7 s	23.5 s	0.54 s
factorial	159	31.8 s	0.9 s	0.11 s	0.01 s
insert sort	197	264.6	3.9 s	15.23 s	0.08 s
quick sort	282	10197 s	10 s	172 s	0.8 s
merge sort	654	952 s	7.77 s	0.62 s	0.04 s
pointer quick sort	693	1546 s	5.8 s	57 s	1.8 s
pointer merge sort	716	> 8 h	> 99 s	78 s	0.25 s

Fig. 4. Comparison of reachability computation using conjunctive partitions against using partial disjunctive partitions.

RS_i, then calculates the image of S using TR_i getting $image_i = image(S, TR_i)$. In order to continue only with the new states, the reachable states are removed from $image_i$ getting $new_i = image_i \setminus RS_i$. In the case where $new_i \neq \emptyset$, it is put in the next entry of the queue. When all processors are trying to read from the queue and they are all pointing to an empty slot in the queue, the algorithm has ended. At the end, each process has the whole reachability set because it saw all the image computation results of all processes in the queue and no new set of states is entered to the queue.

7 Experimental Results

We implemented the translation from conjunctive partitioned transition relation to partial disjunctive partitioned transition relation in the IBM model checker RuleBase [1]. We compared reachability analysis using conjunctive partitions with reachability analysis using partial disjunctive partitions on models that were translated from software programs. These software programs were written in C and contain pointers and arrays. In both cases, we applied dynamic BDD reordering. In order to obtain a fair comparison between these algorithms, we ran each one twice. In the first run, the algorithm reordered the BDD with no time limit in order to find a good BDD order. The initial order of the second run was the BDD order found by the first run. The partial disjunctive partitioning outperforms the conjunctive partitioning with respect to execution time, as shown in Figure 4. We compared the sizes of partial disjunctive partitions with those of conjunctive partitions under the same BDD order. The table in Figure 5 shows the maximal and minimal ratios between a specific variable partial disjunctive partition size and its conjunctive partition size. We specifically note the ratio of the *pc* variable and the size of its partial disjunctive partition. In addition, we show the maximal conjunctive partition and maximal partial disjunctive partition not including *pc*. We observed that the partial disjunctive partitions were in the same order of magnitude or even smaller than the conjunctive partitions. This was achieved by the use of partial disjunctive partitions instead of ordinary disjunctive partitions. In our experiments we found that the

Examples	# Vars	Relations between partitions size			Partitions size		
		Min disj/conj	Max disj/conj	pc disj/conj	Disj pc	Max conj	Max disj
simple	505	0.65	1.54	1.00	8101	10788	10777
factorial	159	0.53	1.27	1.00	3562	1447	1433
insert sort	197	0.58	1.38	0.98	3201	360	390
quick sort	282	0.53	1.34	1.00	12595	1422	1124
merge sort	654	0.47	1.49	1.00	7925	8346	8341
pointer quick sort	693	0.46	1.71	1.00	17650	62225	52155
pointer merge sort	716	0.35	1.30	0.99	6861	66987	32145

Fig. 5. Comparison between size of conjunctive partitions and partial disjunctive partitions.

size of each ordinary disjunctive partition $(or\_R_x(\bar{v}, \bar{v}'))$ was up to 84 times the size of it corresponding partial disjunctive partition.

8 Conclusions and Future Work

Using partial disjunctive partitions seems to be a successful and natural scheme for software models. In this work, we show how to apply disjunctive partitioning to software models while keeping the partitions small. We also show how to enhance the image and pre-image computation to support our partial disjunctive partitions and make model checking algorithms more efficient. However, this is only the beginning and there are a number of directions for future work. As we note above, we handle variables with a large number of bits by creating a single partition for each variable containing the behaviors of all its bits. Future work will explore the possibility of implementing the DNF partitioned transition relation [4], where the disjunctive partition of a state variable is composed of conjunctive partitions of its bits.

As we claimed in Section 6, disjunctive partitioned transition relation is natural for distributed algorithms. It seems wise to implement and explore both algorithms presented in that section. Special attention should be given to finding a good distribution of the disjunctive partitions over the processes in order to achieve good load balancing.

Acknowledgments. We thank Cindy Eisner, Yoad Lustig and Ziv Nevo for many helpful discussions.

References

1. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an industry-oriented formal verification tool. In *Proc. DAC96*, pp. 655–660, 1996.
2. G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *Proc. DAC97*, pp. 728–733, 1997.

3. G. Cabodi, P. Camurati, and S. Quer. Auxiliary variables for extending symbolic traversal techniques to data paths. In *Proc. DAC94*, pp. 289–293, 1994.
4. W. Chan, R. Anderson, P. Beame, and D. Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In *Proc. ISSTA98*, 1998.
5. C. Eisner. Model checking the garbage collection mechanism of SMV. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
6. C. Eisner and D. Peled. Comparing symbolic and explicit model checking of a software system. In *Proc. SPIN2002*, LNCS 2318, pp. 230–239, 2002.
7. D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Proc. CAV94*, LNCS 818, pp. 299–310, 1994.
8. J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pp. 49–58, 1991.
9. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
10. M. Solé and E. Pastor. Traversal techniques for concurrent systems. In *Proc. FMCAD 2002*, LNCS 2517, pp. 220–237, 2002.