Saturation Heuristic for Faster Bisimulation with Petri Nets

Malcolm Mumme

June 2, 2009

1 Abstract

Bisimilarity is an extensional notion of equivalence of states of an automata. A *bisimulation* is a set of such equivalences among the states of automata. The bisimulation problem is the task of deciding bisimilarity for all pairs of states of an automata. The present work applies the *Saturation* [15] heuristic and interleaved MDD partition representation to the bisimulation problem. Previous work [15] comparing the use of various partition representations in a classical fixed-point algorithm for the bisimulation problem, motivated the present work. By focusing on bisimulation for systems with deterministic transition relations (Petri Nets) we show that bisimulation can be expressed as a state-space exploration problem. This suggests the use of the saturation[3] heuristic, which has been found to be relatively efficient for state-space generation. The present work explores the use of our novel saturation-based bisimulation algorithm in the context of the SMART verification tool. We compare the execution time and memory consumption of our saturation-based fully-implicit methods (using interleaved MDDs) with the execution time and memory consumption of fully-implicit and partially-implicit methods (using non-interleaved MDDs) considered in the previous work, as applied to the same bisimulation problems. We found that with some models having very many equivalence classes in their bisimulation partitions, our novel algorithm gave better speed performance than any of the other algorithms tested. The memory utilization of the classic algorithm was surprisingly better in those cases where it could be applied. With models having very small numbers of equivalence classes in their bisimulation partitions, our novel algorithm performed only slightly less well than the fastest algorithm, while the classic algorithm continues to show better memory utilization.

2 Introduction

The formal verification of properties of moderately complex systems frequently involves operations on finite-state automata (FSA) that are very large (containing many states). Algorithms for verification of system properties can require time that grows much worse than linearly in the size of the relevant FSA. The *bisimulation* operation involves (hopefully implicit) comparison of all FSA states to determine if they are (extensionally) distinguishable. The FSA are frequently so large, that they must be stored using a *symbolic* representation, which itself may be quite large. As such large FSA have prohibitive resource requirements, additional improvements are desired. Fortunately, properties of FSA that need to be verified are usually defined extensionally. The results of verification depend only on the behavior of the input FSA, and not intensional characteristics, such as the details of their representation. Conveniently, the input FSA are often not the smallest FSA having the same behavior. Consequently, one may often replace the input FSA with equivalent smaller FSA having the same behavior, using the largest bisimulation to find the smallest equivalent FSA. The search for the largest bisimulation of a given FSA is called the *relational*

coarsest partitioning problem. The relational coarsest partitioning problem was solved optimally (for explicit representations) by Paige and Tarjan [16] in 1987. Other solutions ([1] [9] [21] discussed below) to this problem have been found for FSA in symbolic representations. We expected the current work to represent an improvement on these methods.

2.1 SMART

SMART (Stochastic Model checking Analyzer for Reliability and Timing) (see `http://www.cs.ucr.edu/~ciardo/SMART`) is a modeling and verification tool that implements various forms of model checking and Markov system analysis, among other things. When so directed, SMART applies symbolic methods to the solution of model checking problems and the solution of Markov systems. As research in these areas progresses, improved methods of solution are integrated with the SMART tool. The present work implements one of these steps in the evolution of SMART.

2.2 *Symbolic* Methods

Symbolic methods in model checking are distinct from the more commonly known *mathematical* symbolic methods. Mathematical symbolic methods employ finite symbolic formulas to represent a much larger (often infinite) collection of facts. For example, the finite formula: $\forall x, y \in \mathbb{N} : x + y = y + x$ represents an infinitely large number of simple facts: $0 + 0 = 0 + 0$, $0 + 1 = 1 + 0$, $1 + 0 = 0 + 1$, $1 + 1 = 1 + 1$, $0 + 2 = 2 + 0$, $1 + 2 = 2 + 1$, $2 + 0 = 0 + 2$, $2 + 1 = 1 + 2$, $2 + 2 = 2 + 2$, etc.

By contrast, symbolic methods in model checking employ finite data structures to represent other (usually much larger) finite collections of data. Figure 1, for example, contains 2 different pictures of a symbolic representation for the bit string "1000011101111100".

Typically, *Decision Diagrams (DDs)* are the finite data structures employed, and they represent finite sets, relations, and functions. The following types of decision diagrams are explained individually in the following subsubsections. *(Ordered) Binary Decision Diagrams ((O)BDDs)* typically are used to represent sets of states of systems where each component of a state is binary, such as the states of bits in a register. BDDs can also represent relations between sets of such states. *Multi-way Decision Diagrams (MDDs)* typically are similarly used on such systems except where each component of a state has some finite number of values (possibly greater than 2), such as certain Petri nets. *Multi-Terminal Decision Diagrams (MTDDs)* can represent mappings from states (of various kinds) onto any explicitly represented finite set. *Edge-Valued Decision Diagrams (EVDDs)* can compactly represent functions from states onto reals (floats, actually).

The various kinds of decision diagrams support efficient algorithms for a variety of operations. OBDDs and MDDs provide support for efficient set operations, such as union, intersection, and difference. When used to represent relations, they also support composition, application, and transitive closure (using the saturation heuristic), with somewhat more effort. MTDDs support efficient arbitrary pairwise operations on the represented functions (that is, computing $H \leftarrow F \odot G$, so that $H(x) = F(x) \odot G(x)$, for arbitrary operations $\odot$), and efficient reduction operations, provided that the ranges of all functions remain of reasonable (small) size. Some EVDDs efficiently support various pairwise real operations.

2.2.1 (Ordered) Binary Decision Diagrams (BDDs) [2]

BDDs use a binary tree-like structure to represent a (hopefully larger) table of booleans, or anything representable as a table of booleans. A set of binary register states can be mapped to a table of booleans, by constructing the table index from the bits of the state. Consider the domain $[0, 15]$, which could represent the 16 states { "0000", ... "1111" } of a 4-bit register. We show how to represent a subset $\{0, 5, 6, 7, 9, 10, 11, 12, 13\}$ = { "0000" , "0101" , "0110" , "0111" , "1001" , "1010" , "1011" , "1100" , "1101" } of this domain as a table of booleans. In one column, list the domain elements in order. In the second column, enter "0" in those rows ($\{1, 2, 3, 4, 8, 14, 15\}$) where the domain element is not in the set, and enter "1" in those rows ($\{0, 5, 6, 7, 9, 10, 11, 12, 13\}$) where the domain element is in the set.

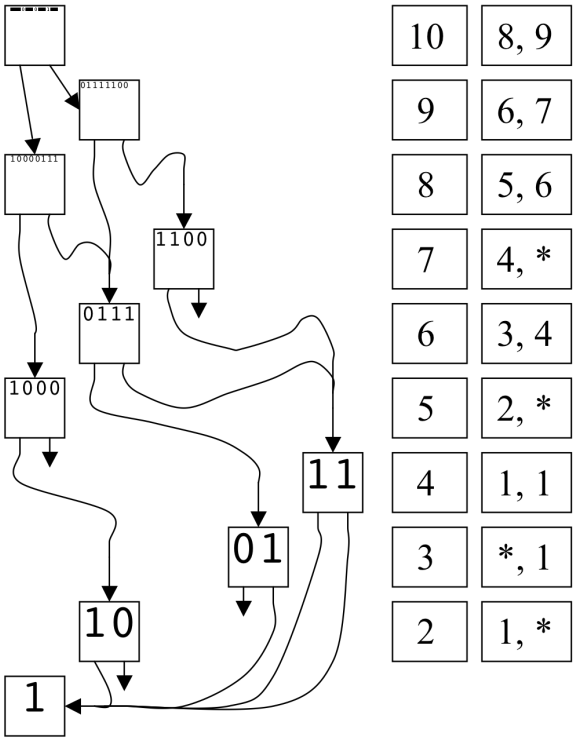


Figure 1: A symbolic representation for a sequence of bits (1000011101111100). Here, each node is labeled with a sequence of bits, which is represented by the subtree starting at that node. Each subtree, starting at some node, represents the sequence of bits constructed by concatenating the sequences of bits represented by the subtrees starting at the children of that node. In this binary diagram, notice that each node has a bit string twice as long as those of its children. Arrows going nowhere point to imaginary nodes containing an appropriate number of zeros. The table on the left shows how this tree might be represented in the memory of a computer. Each row corresponds to a node in the diagram, and the two numbers in the rightmost box of each row indicate destinations of arrows from the corresponding node. "*"s represent arrows going nowhere.

The contents of the second column comprise a bit string "1000011101111100" (or table of booleans) that represents the subset $\{0, 5, 6, 7, 9, 10, 11, 12, 13\}$. Next, we will illustrate the transformation of the table of booleans into a corresponding BDD. Figure 2 (left) shows an *unreduced* binary tree representing the same bit string. In the bottom row are leaves of the tree, that are considered to represent individual 1's and 0's. In the other rows are nodes, each with two ordered outgoing edges. So that the correctness of this representation will be conspicuous, each of these nodes is labeled with a bit string it is considered to represent, and the nodes are drawn with sizes proportional to the sizes of the bit strings they represent. Note that at each level of the tree, all the bit string sizes are the same, being half the size of the bit strings represented by nodes of the preceding level. Also note that edges from a level only target nodes at the very next level.

Except for the leaves, each level of the tree is associated with a single one of the four bits in the state of the register being modeled. The top, or root, node of the tree is associated with the leftmost register bit, while the lowest (other than the leaves) is associated with the rightmost register bit. Each outgoing edge from a node is associated with a specific value of the bit associated with the node's level. The leftmost edge is associated with the value 0, while the rightmost edge is associated with the value 1. These associations provide the following simple method, using the binary tree, for testing if a specific register state is a member of the represented set. Starting at the root of the tree, consider consecutive bits of the register state. For a bit whose value is 0, move down the left edge from the node, and consider the next bit of the state. Likewise, if a bit has the value 1, move down the right edge from the node, and consider the next bit of the state. Proceeding thusly, until the bits of the state are exhausted, a leaf node is found. The state is a member of the represented set iff the leaf has the label "1". This procedure, together with the BDD contents, can be thought of as an implementation of the characteristic function of the set, and, at the same time, as an implementation of the constant table of booleans from which it was derived.

0 “0000”	1
1 “0001”	0
2 “0010”	0
3 “0011”	0
4 “0100”	0
5 “0101”	1
6 “0110”	1
7 “0111”	1
8 “1000”	0
9 “1001”	1
10 “1010”	1
11 “1011”	1
12 “1100”	1
13 “1101”	1
14 “1110”	0
15 “1111”	0

As examples, consider the states 6 (“0110”), and 14 (“1110”). To test state 6 for membership, start at the root, and consider the first bit (“0”) of state 6. Selecting the left edge leads to the node labeled “10000111”, then consider the second bit (“1”) of state 6. Select the right edge to find the node labeled “0111”, and proceed to the third bit, also “1”. Follow the right edge to the node labeled “11”, and consider, finally the last bit of state 6, a “0”. The left edge targets a leaf node labeled “1”; consequently we know that state 6 is a member of the set. With state 14, we start again at the root, and according to the values of the consecutive bits (“1”, “1”, “1”, and “0”), we select the right edge, proceeding to the “01111100” node, the right edge of that, going to the node marked “1100”, the right edge of that node, targeting a “00” node, then the left edge of that, ending in a leaf node marked “0”, showing that state 14 is not a member of the represented set.

Figure 2 (middle) shows a related graph, after partial *reduction*. The difference between this graph and the graph in Figure 2 (left) is the removal of redundant nodes. Each node representing the same value, as some node to its left, has been removed. Edges targeting removed nodes have been adjusted to target remaining nodes representing the same value. Although this example shows a very minor decrease in size due to reduction, BDDs representing FSMs in model checking applications frequently exhibit a size improvement by many orders of magnitude, between the *quasi-reduced* BDD and the corresponding unreduced BDD.

Figure 2 (right) simply shows the same graph, where the nodes are drawn with constant size. An additional small improvement can be obtained by removing nodes which represent bit strings containing only “0”s. Edges targeting such nodes are replaced by special *NULL* edges, here (Figure 1 and elsewhere) drawn as arrows leading nowhere. The result is the BDD previously shown as Figure 1 (left). As long we know that the size of the string represented at each level, we can easily determine the number of “0”s represented by a given NULL edge, since it previously targeted a node representing a bit string that was half the size of the bit strings represented at the previous level. The above method for membership testing still applies to partially reduced graphs, except that upon encountering a NULL edge, one may immediately conclude that the element is not a member of the set. Figure 1 (right) illustrates a possible tabular computer representation of the graph in Figure 1 (left).

It is still possible to decrease the size of the graph by removing nodes where all outgoing edges target the same node. Any edges targeting such a node are retargeted to the targets of edges of the removed node. In such a *fully reduced* representation, it is necessary to mark each node with an indication of its level, since edges are no longer limited to proceed from one level

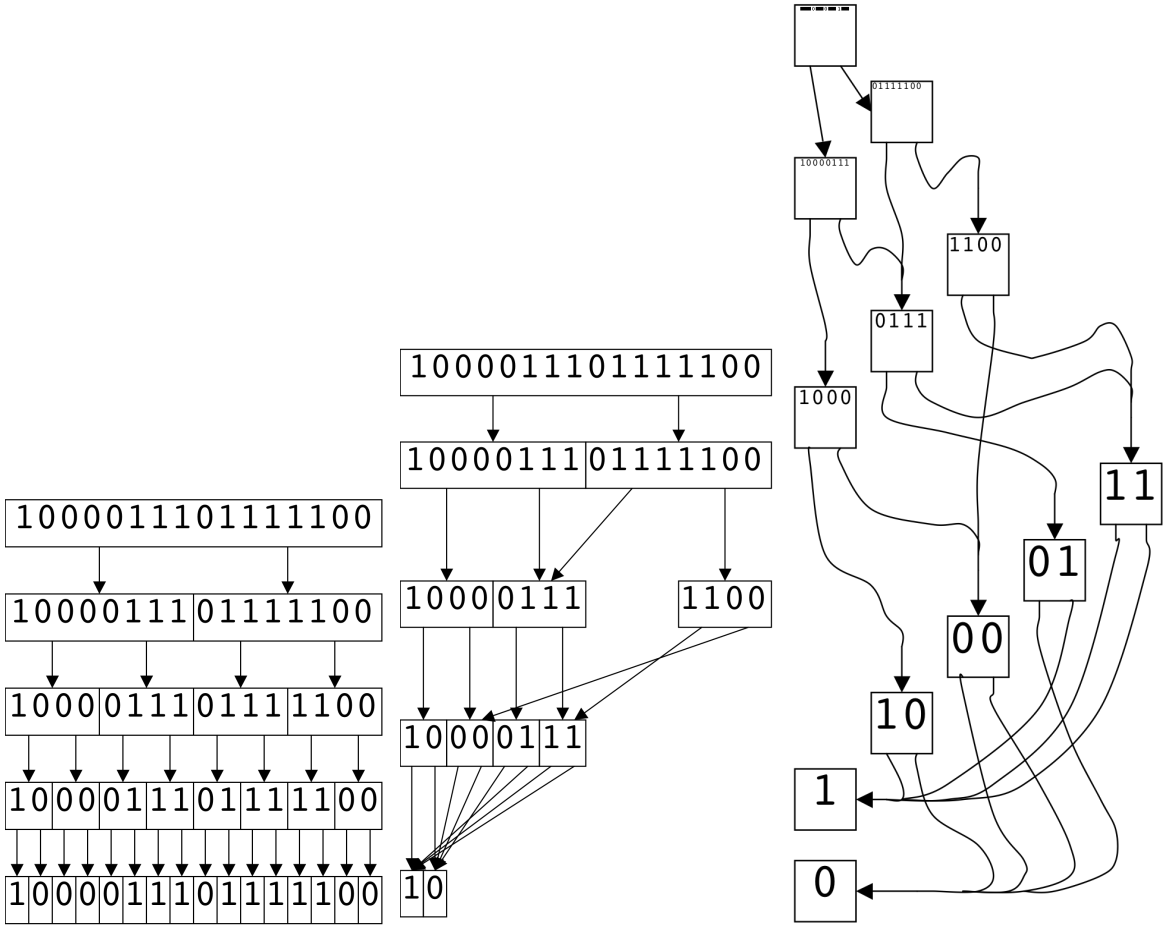


Figure 2: Binary trees for "1000011101111100".

to the very next level.

Fully reduced diagrams are sometimes preferred for reasons of storage efficiency; however, they will not be discussed further here, as they would complicate the explanation of relevant algorithms. The above method for membership testing, for example, must be modified to use the level markings to determine which state bit to consider. Additional variations and extensions of BDDs are used, some of which are described later.

2.2.2 (Ordered) Multi-way Decision Diagrams (MDDs) [12]

Whereas nodes of quasi-reduced BDDs always have two outgoing edges, MDDs have no such restriction, but are otherwise similar. They occur in unreduced, quasi-reduced and fully reduced varieties, and with additional extensions yet to be described. MDDs are ideal for storing a set of states of a machine where the individual state variables that compose the state have more that two different values. As with BDDs, reduced and quasi-reduced varieties of MDDs provide multiple orders of magnitude improvement in the size of representations used in many model checking problems.

Consider a state machine with four individual variables, whose values are in the domain $\{0, 1, 2\}$. We could use a notation such as "0221" to denote the state where the first variable has the value "0", the second and third variables have the value "2", and the fourth and last variable has the value "1". Figure 3 shows an MDD representation of a set of states { "0001", "0010", "0100", "0111", "1011", "1020", "1110", "1121", "2000", "2021", "2101", "2120" } of this machine.

In this diagram, the edges are labeled with associated values, while the nodes have not been labeled. This graph is quasi-reduced. The full tree, even if NULL edges were used, would require ten additional non-leaf nodes, and eleven redundant leaf

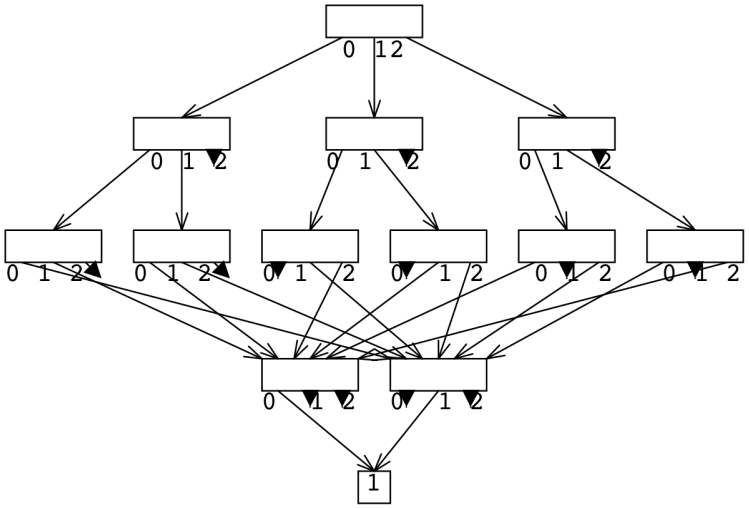


Figure 3: MDD{"0001", "0010", "0100", "0111", "1011", "1020", "1110", "1121", "2000", "2021", "2101", "2120"}.

nodes labeled "1". A slightly modified method for testing membership may be used with this form of decision diagram. As with BDDs, start at the root node, and consider consecutive variables. Follow the edge from the current node whose label matches the value of the variable, then consider the next variable. If a NULL edge is followed, or the leaf node "0" is found, the element is not in the represented set, while arrival at the leaf node "1" indicates membership in the set.

This diagram, derived from a simple model, can also illustrate the importance of *variable ordering*. It has been found that careful choice, of the order in which to consider variables, can be used to significantly decrease the size of decision diagrams. We illustrate this by exchanging positions of the second and third variable, and constructing a new quasi-reduced diagram from the resulting table.

We start with a table of states, in lexicographic order:

The ordered state table:

"0001"
"0010"
"0100"
"0111"
"1011"
"1020"
"1110"
"1121"
"2000"
"2021"
"2101"
"2120"

, then exchange positions:

"0001"
"0100"
"0010"
"0111"
"1101"
"1200"
"1110"
"1211"
"2000"
"2201"
"2011"
"2210"

, then reorder the table:

"0001"
"0010"
"0100"
"0111"
"1101"
"1110"
"1200"
"1211"
"2000"
"2011"
"2201"
"2210"

.

Then produce the new diagram from the (lexicographically) reordered table. The resulting diagram, pictured in Figure 4, has one third fewer non-leaf nodes than the previous diagram. As with many other optimizations, careful variable ordering can produce orders of magnitude reduction in the size of many large diagrams. In the general case, determination of the optimal variable ordering, to reduce MDD size, is a difficult problem. Fortunately, effective heuristics have been found for variable order choice, that produce very good reductions in MDD size. These heuristics are vaguely based on the notion that related variables should be placed near each other in the ordering. In the above example, in Figure 3, the first and third

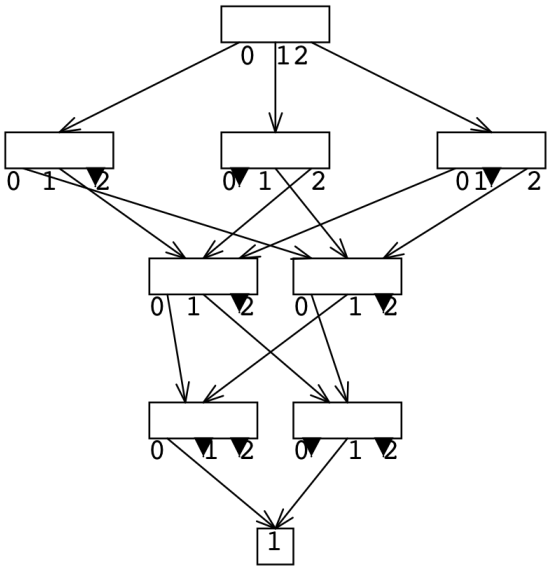


Figure 4: MDD for {"0001", "0010", "0100", "0111", "1011", "1020", "1110", "1121", "2000", "2021", "2101", "2120"}, after reordering (switching second and third variables) to: {"0001", "0100", "0010", "0111", "1101", "1200", "1110", "1211", "2000", "2201", "2011", "2210"}.

variables were related to each other in the original problem, as were also the second and fourth variables. In the reordered MDD, in Figure 4, where the second and third variables were switched, the related variables became adjacent, resulting in the improvement in BDD size. We shall see later exactly how those variables were related.

Although the MDDs shown so far have each had a fixed number of edges associated with the non-leaf nodes, there is no requirement for an MDD to do so. In fact, there are important algorithms which depend on the use of MDDs where the number of edges is not fixed, and cannot even be determined a priori.

2.2.3 Multi-Terminal Decision Diagrams (MTDDs) [10]

In the same way that BDDs and MDDs are equivalent to tables of booleans, MTDDs are equivalent to tables of values from any finite type. That is, they implement functions with arbitrary finite ranges, as opposed to characteristic functions (with boolean range). MTDDs can also be a BDD (MTBDD) or an MDD (MTMDD), the main difference being that plain BDDs and MDDs have leaf nodes with values "0" and "1", while MTDDs can have values from the associated finite type. As an illustration of MTDDs, an MTMDD for the "Rock-Paper-Scissors" game is shown in Figure 5. The range of the function is { win, loose, draw }. The diagram shows the outcome of the game, for player 1, given that the first variable represents the choice made by player 1, and the second variable represents the choice made by player 2. The procedure for evaluating a function, represented this way, is similar to the membership testing procedure for MDDs. The leaf node one arrives at, at the end of the procedure, determines the value of the function.

It should be noted that, although the range of such functions would usually be drawn from simple enumerable types, the range could also be a finite subset of some continuous set, such as the reals. If a bet was riding on the outcome of the game, for example, then the range of the function might be { 0.0, -5.0¢, +5.0¢}.

Some algorithms [6] for *Markovian Lumping* (a problem analogous to bisimulation, but pertaining to the solution of Markov systems) use this type of MTDD to represent functions onto $\mathbb{R}$, by adjusting the range to match the finite set of values of interest at a particular time. Note that this use of MTDDs is *explicit* in the number of different values in the range of the function. That is, a separate component of the data structure must exist for each distinct value in the range. Consequently, the size of an MTDD representing some function f, is at least O($|range(\ f\ )|$).

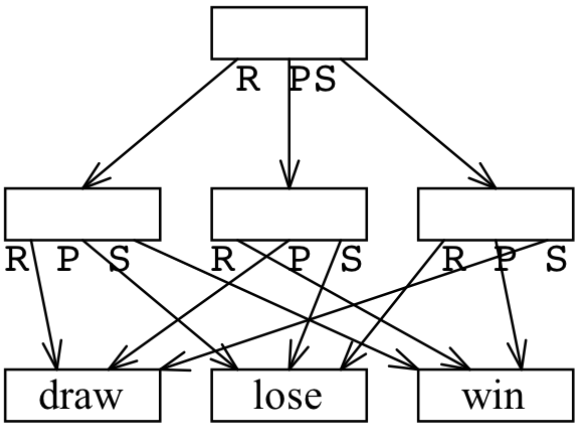


Figure 5: MTMDD for "Rock-Paper-Scissors" game.

2.3 Representation of sets, relations and functions using Decision Diagrams

Use of the appropriate form of decision diagram for the representation of sets, relations, and functions can provide multiple orders of magnitude improvement in the size of the representations, as well as in the computation time required for operations on those objects, as compared with the use of explicit representations.

2.3.1 Representation of sets using Decision Diagrams

In decision diagrams, sets are represented implicitly. That is, sets are represented as a graph, which was reduced from a tree, which was derived from a boolean table of all values of the characteristic function of the represented set. Thus sets are represented as boolean functions, converted to tables, then trees, then reduced to decision diagrams. Hence, if one wished to represent a multi-set, one might start by converting its characteristic function to a table of integers, then to a tree, then reducing it to a MTDD or to an EVDD.

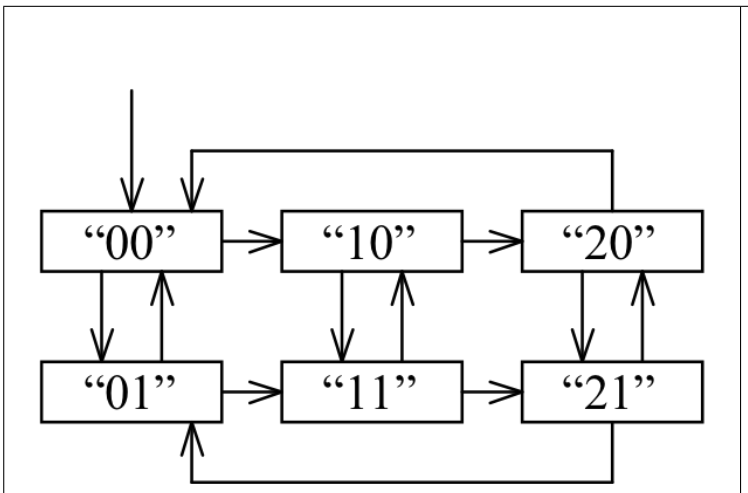
2.3.2 Representation of relations using Decision Diagrams

Relations may be represented as sets, by considering them to be sets of pairs, or other tuples, as one is likely to do in a study of relational algebra. One way to obtain a representation of the value of a tuple is to concatenate the values of the variables belonging to all the elements of the tuple. Another way to represent the value of a tuple will be discussed in section 2.3.3. Converting all such representations, that apply to a particular relation, to a table of booleans provides the necessary characteristic function for building a decision diagram.

Consider the state transition relation for the FSA shown in Figure 6 (left). The FSA shown in Figure 9 (left) describes the reachable behavior of the pair of FSA shown in Figure 7. Here, we see that the first variable in the state of the FSA in Figure 6 (left) is the entire state for the first (left) FSA shown in Figure 7. Of course, the second variable in the state of the FSA in Figure 6 (left) is the state of the second (right) FSA from Figure 7. These two sub-states have already been concatenated to make the state of the FSA in Figure 6.

The FSA in Figure 6 (left) may be encoded as the relation shown in Figure 6 (right). We may then, for each tuple, concatenate the variable values from the corresponding elements, to produce the following set: { "0001", "0010", "0100", "0111", "1011", "1020", "1110", "1121", "2000", "2021", "2101", "2120" }. We can then build a table of booleans to represent this set, after listing the domain in the first column:

The MDD that represents this set has already been shown in Figure 3.



⟨ “00” → “01” ⟩
⟨ “00” → “10” ⟩
⟨ “01” → “00” ⟩
⟨ “01” → “11” ⟩
⟨ “10” → “11” ⟩
⟨ “10” → “20” ⟩
⟨ “11” → “10” ⟩
⟨ “11” → “21” ⟩
⟨ “20” → “00” ⟩
⟨ “20” → “21” ⟩
⟨ “21” → “01” ⟩
⟨ “21” → “20” ⟩

Figure 6: A FSA and its transition relation (without edge labels).

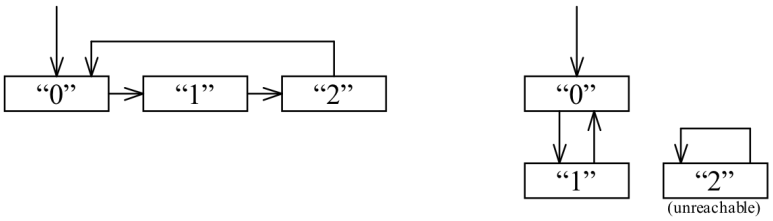


Figure 7: FSAs whose product is the FSA of Figure 6.

“0000”	0
“0001”	1
“0002”	0
“0010”	1
“0011”	0
“0012”	0
“0020”	0
“0021”	0
“0022”	0
“0100”	1
“0101”	0
“0102”	0
“0110”	0
“0111”	1
“0112”	0

(cont.)	
“0120”	0
“0121”	0
“0122”	0
“0200”	0
“0201”	0
“0202”	0
“0210”	0
“0211”	0
“0212”	0
“0220”	0
“0221”	0
“0222”	0
“1000”	0
“1001”	0

(cont.)	
“1002”	0
“1010”	0
“1011”	1
“1012”	0
“1020”	1
“1021”	0
“1022”	0
“1100”	0
“1101”	0
“1102”	0
“1110”	1
“1111”	0
“1112”	0
“1120”	0

(cont.)	
“1121”	1
“1122”	0
“1200”	0
“1201”	0
“1202”	0
“1210”	0
“1211”	0
“1212”	0
“1220”	0
“1221”	0
“1222”	0
“2000”	1
“2001”	0
“2002”	0

(cont.)	
“2010”	0
“2011”	0
“2012”	0
“2020”	0
“2021”	1
“2022”	0
“2100”	0
“2101”	1
“2102”	0
“2110”	0
“2111”	0
“2112”	0
“2120”	1
“2121”	0

(cont.)	
“2122”	0
“2200”	0
“2201”	0
“2202”	0
“2210”	0
“2211”	0
“2212”	0
“2220”	0
“2221”	0
“2222”	0

2.3.3 Interleaved Decision Diagram representation of relations

The use of sets of pairs, to represent relations, similarly to the above description, may be implemented in other ways. One other way to obtain a representation of the value of a tuple is to shuffle, or *interleave*, the values of the variables belonging to all the elements of the tuple.

Here we take the set of tuples representing the previously described transition relation:

{ ⟨ "00" → "01" ⟩, ⟨ "00" → "10" ⟩, ⟨ "01" → "00" ⟩, ⟨ "01" → "11" ⟩, ⟨ "10" → "11" ⟩, ⟨ "10" → "20" ⟩, ⟨ "11" → "10" ⟩, ⟨ "11" → "21" ⟩, ⟨ "20" → "00" ⟩, ⟨ "20" → "21" ⟩, ⟨ "21" → "01" ⟩, ⟨ "21" → "20" ⟩ },

and in each case perform a perfect shuffle on the two variable values from the domain and the two variable values from the range, producing a single parameter with four variables, yielding the following set:

{"0001", "0100", "0010", "0111", "1101", "1200", "1110", "1211", "2000", "2201", "2011", "2210"}.

This can easily be recognized as the same set in the caption of Figure 4, and results in the same small MDD representation. Inspection of the related tables and diagrams allows one to see how the first variable and the third variables in the MDD from section 2.2.2 are related to each other. They originate as the state variable in the FSA in Figure 7 (left), as incorporated as the first variable in the state of the FSA of Figure 6. The state transition relation, of that FSA, then incorporates the state same variable in both its domain and its range, producing related variables (the first and third variables) in the set-of-tuples encoding of that relation. Similarly, the second and fourth variables in the MDD from section 2.2.2 originate as the state variable in the FSA in Figure 7 (right), and are similarly incorporated into the domain and range, of the state transition relation of the FSA of Figure 6.

Thus, we can see from this example that the use, of interleaved representation of relations brings certain related variables together, in the MDD representation. This provides some of the benefits of good variable ordering without a (potentially costly) variable re-ordering step, in the production of the final MDD.

2.3.4 Representation of functions of states using Decision Diagrams

Although we have used boolean functions to represent relations, it should be noted for the sake of completeness, that in set theory, functions are a kind of relation where each element of the domain is the first member of exactly one tuple. Consequently, functions of states, whose ranges are representable using decision diagrams, can be represented using decision diagrams in the same manner as the corresponding relation. Although it frequently happens (especially in the present work) that a state transition relation happens to be a function, it is typical for these functions to be represented in the same manner as are general relations.

2.4 Simple Saturation Heuristic

The saturation heuristic has been effectively used to provide rapid state-space generation in the SMART tool. The output of this algorithm is the transitive closure of a number of relations applied to a set of initial states. The sophisticated state-space generation algorithm in the SMART tool is able to generate finite state spaces without being provided with a (finite) potential state space in which to search. We here describe a saturation algorithm which, for the sake of brevity, requires a given finite potential state space $\hat{S}$.

2.4.1 Encoding of state-sets and transition relations for the saturation heuristic

Sets of states are encoded as MDDs similarly to the above descriptions. Each of N state variables corresponds to a unique level of the MDD, which has N levels. This correspondence is called the variable ordering. The leaves of the tree are considered to be at level 0, the nodes of level $k+1$ have edges leading to nodes of level k, so that the root is at level N. A state variable is said to be at level k if it corresponds to level k of the MDD in the variable ordering. Values of individual variables are uniquely mapped to a small range of integers, which are then used in the encoding of the corresponding level of the MDD. The ability to produce such a mapping depends on knowing the finite potential state space.

Given this encoding of state-sets, each transition relation $\mathcal{T}$ is encoded as an interleaved MDD t with $2N$ levels, similarly to the examples given previously. Variable k from the domain is corresponds to level $2k$ in t, while variable k from the range corresponds to level $2k-1$ in t. This interleaved representation generally results in a relatively compact representation, as mentioned previously, and also allows a fairly efficient algorithm for application of a relation/function to sets of state values.

2.4.2 Spans of transition relations

In typical problems of interest, the state transitions each involve relatively few state variables. In particular, for a given transition $\mathcal{T}$, most state variables are unchanged by the state transition, and do not influence the value of the result(s). Given a particular variable ordering, the *Top* of a given transition is the number of the highest numbered variable involved with the transition, while the *Bot* of a transition is the number of the lowest numbered variable involved with the transition. The span of the transition MDD t is then $1+Top(t)-Bot(t)$. The saturation heuristic takes advantage of the fact that, for many problems, careful variable ordering can cause encodings of transition relations to have a *span* that is much shorter than $2N$. This efficiency partly arises because the operation of applying a transition MDD t to a state-set MDD involves manipulating state-set MDDs only at levels in $[Bot(t), N]$. Many other symbolic algorithms also benefit from this *locality* property.

Moreover, the saturation heuristic only applies a transition t to sets where variables above level $Top(t)$ are fixed in value. consequently, transition t is applied in a way that only manipulates state-set levels in $[Bot(t), Top(t)]$. Consequently, it is not surprising that variable orderings that produce shorter transition spans often result in relatively efficient saturation processing. It should be noted here that in the $2N$ level encoding of t, typically, level $2k$ corresponding to domain variable k will only have one node, if k is not in $[Bot(t), Top(t)]$. In this case, level $2k-1$ will have $|domain(k)|$ nodes, each corresponding to one of the possible values of variable k. As they are effectively unused, it is not necessary to store levels k of a transition t where $k > Top(t)$, so that the MDD for t actually has $Top(t)$ levels.

2.4.3 Simplified Algorithm

We apply the transitive closure of the indexed collection of transitions $t_{[\mathcal{E}]}$ to an initial space S_{in} by calling:

$S_{closed} \leftarrow$ *SaturationClosure*($t_{[\mathcal{E}]}$, S_{in})

where *SaturationClosure* and its helper function *SaturateChildren* are defined as follows:

2.4.4 Discussion

Line 8 (of *SaturationClosure*) contains the basic algorithmic step for transitive closure. It extends a set S by applying the function t_α. If all transition relations $t_{[\mathcal{E}]}$ have $Top(t_\alpha) = N$, for $\alpha \in \mathcal{E}$, we can ignore the helper function and lines 1,4, and 9 of *SaturationClosure*. The remaining code clearly iterates the functions $t_{[\mathcal{E}]}$ to a fixed point starting with S_{in}. It is the classical fixed-point algorithm.

In cases of interest, we expect some transition relations have $Top(t_\alpha) < N$, for some $\alpha \in \mathcal{E}$. In these cases, we must understand

```
MDD SaturationClosure( MDD t_[E], MDD S_in ) is
  local MDD S, MDD S_old
   1 if S is leaves, then return S
   2 if SaturationClosure( t_[E], S_in ) is found in memo cache, then return cached result
   3 S ← S_in
   4 S ← SaturateChildren( t_[E], S )                    • first saturate children in input
   5 repeat:
   6   S_old ← S
   7   for each α ∈ E where Top( t_α ) is this level (of S), loop:
   8     S ← S ∪ t_α(S)                                  • apply transition at this level
   9     S ← SaturateChildren( t_[E], S )                • re-saturate in result
  10   end loop
  11 until S_old = S
  12 place S = SaturationClosure( t_[E], S_in ) into memo cache
  13 return S
```

Simplified Saturation Algorithm

```
MDD SaturateChildren( MDD t_[E], MDD S_in ) is
  local mutable MDD node S
   1 S ← a mutable copy of root node of S_in
   2 for each child s of S, loop:
   3   replace edge to s in S with: edge to SaturationClosure( T_[E], s )   • saturate child
   4 end loop
   5 return unique(S)                                  • canonical (immutable interned) version of S
```

Helper Function

the operation of lines 1,4,9, and the helper function *SaturateChildren*. The helper function merely applies the saturation heuristic *SaturationClosure* recursively to children of a partial result S. During calls to *SaturateChildren*, we never apply functions t_α where $Top(t_\alpha) = N$, but only those t_α where $Top(t_\alpha) < N$.

Consider the first call to *SaturationClosure*. Before any transition function at level N is ever applied, *SaturateChildren* is called first, so that all children of S will be closed under all transitions t_α where $Top(t_\alpha) < N$. Consider what this means. When *SaturateChildren* operates on child s of S, the value of variable N is known, since it has traversed the edge of S leading to s. Since all transitions t_α that it applies to s will have $Top(t_\alpha) < N$, we know that variable N is not involved with t_α and will remain unchanged by t_α. Consequently, the operation that *SaturateChildren* performs by applying various $\{t_\alpha | Top(t_\alpha) < N\}$ to each child of S individually, produces the same result as if the various $\{t_\alpha | Top(t_\alpha) < N\}$ were applied to applied to S. Consequently, S produced by *SaturateChildren* is also closed under all transitions t_α where $Top(t_\alpha) < N$. For brevity, we say S is saturated below level N. We may then conclude that the use of *SaturateChildren* in lines 4 and 9 maintains the invariant that S is saturated below level N throughout the loop on lines 5-11, with one exception. At the end of line 8, of course, where some t_α where $Top(t_\alpha) = N$ is applied, S might not be saturated below level N. Consequently, the returned value of S will be saturated below level N.

It is also easy to see that S is closed under $\{t_\alpha | Top(t_\alpha) = N\}$, since the final iteration of the inner loop together with the exit condition of the outer loop requires: $S_{old} = S = S \cup t_\alpha(S) =$ *SaturateChildren*$(\ t_{[\mathcal{E}]},\ S\ )$ for all $\alpha \in \mathcal{E}$, and we can easily show that $S \subseteq$ *SaturateChildren*$(\ t_{[\mathcal{E}]},\ S\ )$. We can be certain of termination, since $|S|$ will increase on each non-final iteration of the outer loop, but cannot increase beyond $|\hat{S}|$.

With correctness having been shown, we may now consider the question of why the saturation heuristic usually performs well compared to the classical algorithm. The following speculative observations have been made:

1. The saturation of children of a set in this discrete transitive closure problem is reminiscent of the local iteration of a block decomposition of a sparse matrix inversion problem where large values are mostly near the diagonal. If one were to consider combining all the transition relations $t_{[\mathcal{E}]}$ into a single transition relation t_*, one would find that if the

spans of the individual transition relations were small, the transition matrix representation of t_* would tend to have its non-zero elements near the diagonal, or near diagonals of sub-matrices.

2. It has been noted that the memory size of the MDD representing the final closed set S is usually less than the MDD memory size of many intermediate (non-closed) values of S. By saturating the children of S before applying transitions at level N, thus ensuring that the children are closed under some transitions, we keep their memory sizes low compared to what they could be in some other algorithm. By operating on smaller MDDs, we may improve the run-time of our operations

3. Saturation preserves the benefits of sharing among canonical MDDs by using a function cache. Consider a space s at level k, and two edges e_1, and e_2 from s. Assume these edges correspond to values v_1 and v_2, respectively, of variable k. If operations at level k or above do not treat the values v_1 and v_2 of variable k differently, and e_1, and e_2 point to the same child of s in the initial space, then, in all stages of the computation, in various updated versions s' of s, edges of s' correspond to values v_1 and v_2 will point to the same child of s'. Consequently, all updates of the child are shared by both edges.

2.5 Bisimulation

There are a variety of applications for bisimulation (see [14] for an overview of work in this area). Bisimulation[14] is used in definitions of equivalent computations in various models of concurrent computation, and also has applications such as FSA minimization and logic synthesis. In this work, we are primarily concerned with algorithmic aspects.

A bisimulation of a FSA is a binary equivalence relation among its states. Any two states in this relation are observationally equivalent, in the sense that an FSA in one state cannot be distinguished from the FSA in the other state. Both states have the same sets of transitions enabled, and upon execution of the same transition, both FSA transition to new states that are also in the bisimulation. Thus, if any device, or simulation, was constructed to operate according to the rules of the FSA, one could not distinguish between two states in a bisimulation, by external observations or communication with the device. Devices whose FSA models have large bisimulations are therefore likely to have more states than necessary to implement their behavior. Fortunately, the size of the largest bisimulation indicates the size of the smallest equivalent FSA model having the same bahavior.

Formally, the largest bisimulation $\sim$ of an FSA $\langle S, \mathcal{T}_{[\mathcal{E}]}, c\rangle$ (states (S), labeled transition relations ($\mathcal{T} \subseteq S \times \mathcal{E} \times S$), State coloring ($c : S \rightarrow color$)), is the largest binary equivalence relation $\mathcal{B} \subseteq S \times S$, among states S where:

1. Colors of bisimilar states (a pair of states in $\mathcal{B}$) are equal
   $\forall \langle s_1, s_2\rangle \in \mathcal{B} : c(s_1) = c(s_2)$
2. A pair of bisimilar states has only matching transitions, to bisimilar states.
   $\forall \langle P, Q\rangle \in \mathcal{B}, \alpha \in \mathcal{E}, P' \in S : (\ \langle P, \alpha, P'\rangle \in \mathcal{T}_{[\mathcal{E}]} \Rightarrow \exists Q' \in S : \langle Q, \alpha, Q'\rangle \in \mathcal{T}_{[\mathcal{E}]} \wedge \langle P', Q'\rangle \in \mathcal{B}\ )$
   Abbreviating $\langle X, \alpha, Y\rangle \in \mathcal{T}_{[\mathcal{E}]}$ as $X \xrightarrow{\alpha} Y$ clarifies the formula:
   $\forall \langle P, Q\rangle \in \mathcal{B}, \alpha \in \mathcal{E}, P' \in S : (\ P \xrightarrow{\alpha} P' \Rightarrow \exists Q' \in S : Q \xrightarrow{\alpha} Q' \wedge \langle P', Q'\rangle \in \mathcal{B}\ )$

2.6 Relational coarsest partition

The relational coarsest partition of the state space of an FSA corresponds to the largest bisimulation of that FSA. It is a partition of the state space of the FSA, where the members of any partition element are all extensionally indistinguishable from each other.

The relational coarsest partition problem can be stated as follows:

The relational coarsest partition of an FSA $\langle S, \mathcal{T}_{[\mathcal{E}]}, c\rangle$ is the smallest partition P of states S, where:

1. All states in the same partition block have a common color.
   $\forall B \in P : \exists C \in color : \forall s \in B : c(s) = C$
2. For each transition label, all states in a block have transitions (with that label) to state(s) in a second block, or none do.
   $\forall \alpha \in \mathcal{E}, B_1 \in P, B_2 \in P : \exists q \in \{true, false\} : \forall s_1 \in B_1 : q \Leftrightarrow \exists s_2 \in B_2 : s_1 \stackrel{\alpha}{\rightarrow} s_2$

Algorithms for solving this problem are discussed in the section on related work, and in the section describing our work.

3 Main Idea

The main idea, mentioned as a possible extension in our previous work [15], is the recasting of bisimulation as a transitive closure problem, allowing use of the highly successful saturation heuristic to accelerate convergence to the fixed point. Our continued use of interleaved diagrams to represent the partition of the state space into equivalence classes, as recommend by Ciardo, allows the data structure to remain compact during the entire process, even when there are very many equivalence classes.

The idea is to compute the largest bisimulation $\sim$ by first computing the *anti-bisimulation*: $\approx = \{\langle s_1, s_2 \rangle \in S^2 | \overline{(s_1 \sim s_2)}\}$, then computing $\sim$ as $S^2 \setminus \approx$. The advantage to this method is that, when the transition relations are deterministic, computing the anti-bisimulation takes the same form as a simplified reachable-state space generation problem, for which the saturation heuristic often provides a relatively efficient solution.

Given a particular state space S and a deterministic transition relation $\mathcal{T} : S \rightarrow S$, ($S$ is closed under $\mathcal{T}$), we produce the relation:

$T \subseteq S^4 = \{\langle \langle s_1, s_2 \rangle, \langle s_3, s_4 \rangle \rangle | s_1 = \mathcal{T}(s_3) \wedge s_2 = \mathcal{T}(s_4)\}$,

under which $\approx$ must be closed. In an actual bisimulation problem, there may be many deterministic transition relations $\mathcal{T}_\alpha : S \rightarrow S$ (where $\alpha \in \mathcal{E}$ an index set), each having a corresponding transition relation $T_\alpha \subseteq S^4$, under which $\approx$ must be closed. To compute $\approx$, we start with a certain subset of S^2 that is known to be a subset of $\approx$, and construct $\approx$ as the transitive closure of all $\{T_\alpha : \alpha \in \mathcal{E}\}$, using the saturation heuristic.

We expect this to yield space savings compared to non-interleaved representations, since existing non-interleaved representations are explicit in the number of equivalence classes found. We expect additional space savings compared to our previous interleaved but non-saturating algorithms, in keeping with the observation that saturated (closed under some operations) MDDs are often smaller than alternatives. We expect improvement in execution time for the saturation-based algorithm, consistently with the known efficiency of saturation-based transitive closure algorithms. We also expect execution time improvement for our algorithms, due to their manipulation of smaller structures, especially in cases involving many classes, as compared with algorithms [16] [1] [21] that do not use interleaved representation. By reduction of memory and processing requirements, we hope to enable the processing of FSAs that could not be processed by previous methods. The primary motivation behind our previous work and the present work is to provide the benefits of interleaved representation of partitions to the lumping problem. It appears [11] [7] [8] [5] [6] that these ideas have not yet been applied to the lumping problem.

4 Related Work

The following papers are representative of the previous fastest known explicit and symbolic algorithms for bisimulation.

4.1 *Three partition refinement algorithms* [16]

This Article by Paige and Tarjan [16], describes their $O(m(log(n)))$ time algorithm for Relational coarsest partition. (where m = number of edges, n= number of vertices). This is an 'explicit' algorithm that operates on data structures directly representing the nodes and edges of the problem.

The main idea is to do iterative "splitting" on a sequence of partitions of the graph of the state space U. Each iteration splits partition members, each called B, using another partition member S, called the splitter. In the new algorithm, S is either the entire set U, or S is chosen so that it is a subset of a previous splitter S_0 and at most half the size of S_0. Splitting B using S yields up to 3 subsets, B_0, and B_1, and B'. B_0 contains members of B with edges to S and no edges to $S_0 - S$. B_1 contains members of B with edges to both S and $S_0 - S$. B' contains members of B with only edges to $S_0 - S$. Using a fairly simple backward adjacency list representation, members of $B_0 \cup B_1$ can easily be identified in time linear with the number of edges from $B_0 \cup B_1$ to S.

A naive algorithm would also traverse the representation of $S_0 - S$, to identify the members of B', and to distinguish the members of B_0 from those of B_1. A major contribution of this paper is a method of associating edge counts with the vertices of B, of the number of edges to vertices of S, and maintaining them. The use of these edge counts allows for the identification of members of B_0 and of B_1 in time linear with the number of edges from $B_0 \cup B_1$ to S. After moving the vertices of B_0 and B_1 out of the representation of B, the part that remains in B now becomes the representation for B', so there is no need to explicitly identify the individual members of B' as part of the splitting process. These simplifications allow the overall iterative splitting algorithm to achieve the $O(m(log(n)))$ bound.

The edge counts are maintained and used as follows. An edge count data structure exists for each pair of a member of U, and potential splitter S. Each data structure representing an edge from a member of U to a member of S also contains a pointer to the corresponding edge count data structure, which represents the number of such edges. Whenever S is itself split, usually the data structures representing S will be re-used to represent one of the largest products of such a split, while new data structures will be used to represent the other products. The edge count data structures will be updated while updating edges pointing to members of S that are moved to other products.

When splitting B on splitter S, we traverse reverse adjacency lists of members of S to determine members of $B_0 \cup B_1$. We will re-use data structures representing S_0, to represent $S_0 - S$, and reuse structures representing B, to represent B'. Consequently, it will be unnecessary to explicitly identify members of B'. When traversing reverse adjacency lists of members of S to determine members of $B_0 \cup B_1$, we also count how many edges occur, from each member of $B_0 \cup B_1$, to members of S. After this traversal, the members of B_0 and members of $B_0 \cup B_1$ can easily be distinguished, since the counts for members of B_0 (but not of B_1) will be identical to the edge counts for the member of edges pointing to S_0.

For analysis purposes, observe that any given element of U can only occur as a member of a splitter, up to $O(log(n))$ times, since a splitter S is always chosen to be, at most, half the size of S_0, a previous splitter containing S. Observing that each splitting operation is performed by traversing reverse adjacency lists of elements of a splitter, so that each element contributes cost linear in the size of its adjacency list each time it occurs in a splitter, we derive the overall $O(m(log(n)))$ bound.

4.2 *Symbolic Bisimulation Minimisation*[1]

This (1992) article by Amar Bouali and Robert De Simone describes an algorithm that is nearly the same as the algorithm presented in our previous work [15]. It is among the earliest work on using symbolic methods for bisimulation. The main purpose of their algorithm is to calculate *weak bisimulation*, a form of bisimulation where some transitions are considered to be unobservable. They transform a weak bisimulation problem in to the standard bisimulation problem by pre-processing the collection of transition relations. They perform a comparison of their algorithm using only interleaved decision diagrams, with their algorithm using only non-interleaved diagrams. They found that the relative performance using these structures depended on the relational coarsest partition. Interleaved decision diagrams performed better when the partition had very many equivalence classes, while non-intereaved diagrams performed better when it had very few equivalence classes.

4.3 *An efficient algorithm for computing bisimulation equivalence*[9]

This (2004) algorithm by Agostino Dovier, Carla Piazza, and Alberto Policriti operates by first producing a (probably) good initial partition of the set of states, based on the calculated *rank* of states. They can do this because states with different rank cannot be equivalent. For their non-symbolic implementation, they use the set-theoretic notion of rank. The possible presence of cycles in a state transition graph forces them to use non well-founded set theory. In order to compute rank symbolically, they adjust the definition of rank, so that, for many states, rank is essentially the longest distance from the state to a final state through the transition graph after all cycles have been condensed to a single state. Fortunately, they are able to show that their algorithm always terminates in a linear number ($O(n)$, where n is the number of states) of symbolic steps. After their initial partitioning based on rank, other algorithms must be used to finish the partitioning. The rank sequence of the initial partition can be used to efficiently direct the ordering of splitting operations.

4.4 *Forwarding, Splitting, and Block Ordering to Optimize BDD-based Bisimulation Computation*[21]

This (2007) paper by Ralf Wimmer, Marc Herbstritt, and Bernd Becker describes several methods to accelerate symbolic bisimulation computation. The main algorithm is similar to the one used in our previous work [15], as well as in Bouali and De Simone [1], although partition elements are represented explicitly, and assigned unique serial numbers. Aside from the complication of handling invisible transitions, the main optimizations they employ are as follows:

1. They use *signatures* of states to compute the refinements of partition elements (or blocks), as does our previous work [15]. States with different signatures will belong to different partition blocks. Their DD representation for signatures places the state variables toward the root, and the signature variables toward the leaves. This allows them to simply substitute block serial numbers into the DD at the level of the signature, to produce a refined partition efficiently, as the canonicity of the DD representation guarantees that each node at that level corresponds to a distinct block. This technique is also employed in [6].
2. *Block Forwarding* refers to the update of the current partition immediately after blocks are split. They split only one block at a time, so they need only calculate signatures one block at a time. Therefore, they update the partition after each block splitting, allowing subsequent splitting operations to benefit from the more refined partition immediately. This technique is also employed in [5].
3. *Split-Driven Refinement* refers to the use of a *backward signature* (similar to the inverse of the state transition relation) to determine which blocks may be made splittable by the splitting of a given block. This allows the algorithm to skip splitting certain blocks whose elements signatures do not include any blocks that have been split since the certain block was created.
4. *Block ordering* is the deliberate choice of which splitter to use at any given time that such a choice is available. They mention two heuristic orderings, both of which produce improved run times. The choice of the block with a larger backward signature, as well as the choice of the larger block, both produced improvements.

They cannot use interleaved variable ordering for their representation of partitions, since their block forwarding, splitting and ordering optimizations depend on explicit treatment of individual partition blocks.

5 Algorithms

In this section, the algorithms used in our current and previous work are given, starting with the main saturation-based algorithm (A), preceded by definitions used by various algorithms.

The algorithms below (except algorithm A) are described in a manner independent of variable ordering. Consequently, the algorithms are not bifurcated in that way. In some cases, however, more than one version of an algorithm is given, either to illustrate a choice in optimization, or to show some structural choice.

5.1 Definitions

The following operator definitions will be used in various algorithms. It should be noted that we consider predicates (relations) to be sets of tuples, so that we use predicate notation "$F(x, y)$" interchangeably with set notation "$\langle x, y\rangle \in F$" in much of the following:

(parameters a, b, c, x, y are states, while F, G are predicates over, or sets of, states):

$DC_1(F, G) \triangleq F'$, where $F'(a, x, y) = F(x, y) \wedge G(a)$ inserts don't-care parameter (from G) into position 1

$DC_1(F, G) \triangleq F'$, where $F'(a, x, y, z) = F(x, y, z) \wedge G(a)$ inserts don't-care parameter (from G) into position 1

$DC_2(F, G) \triangleq F'$, where $F'(x, a, y, z) = F(x, y, z) \wedge G(a)$ inserts don't-care parameter (from G) into position 2

$DC_2(F, G) \triangleq F'$, where $F'(x, a, y) = F(x, y) \wedge G(a)$ inserts don't-care parameter (from G) into position 2

$DC_2(F, G) \triangleq F'$, where $F'(x, a) = F(x) \wedge G(a)$ inserts don't-care parameter (from G) into position 2

$DC_3(F, G) \triangleq F'$, where $F'(x, y, a) = F(x, y) \wedge G(a)$ inserts don't-care parameter (from G) into position 3

$DC_3(F, G) \triangleq F'$, where $F'(x, y, a, z) = F(x, y, z) \wedge G(a)$ inserts don't-care parameter (from G) into position 3

$DC_4(F, G) \triangleq F'$, where $F'(x, y, z, a) = F(x, y, z) \wedge G(a)$ inserts don't-care parameter (from G) into position 4

$F^{-1}(y, x) \triangleq F(x, y)$ switch variables

$proj_{\vee 2}(F) \triangleq F'$, where $F'(x, y, z) = \bigwedge a : F(x, a, y, z)$ project second parameter by disjunction

$proj_{\vee 3}(F) \triangleq F'$, where $F'(x, y) = \bigvee a : F(x, y, a)$ project third parameter by disjunction

$(F \cap G)(\ldots args \ldots) \triangleq (F(\ldots args \ldots) \wedge G(\ldots args \ldots))$ intersection

$(F \cup G)(\ldots args \ldots) \triangleq (F(\ldots args \ldots) \vee G(\ldots args \ldots))$ union

$(\overline{F})(\ldots args \ldots) \triangleq \neg(F(\ldots args \ldots))$ complement

$(F \cup\!\!\cdot\, G)(\ldots args \ldots) \triangleq (F(\ldots args \ldots) \neq G(\ldots args \ldots))$ states in F or G but not both ($= (F \setminus G) \cup (G \setminus F)$)

Efficient recursive algorithms for computing $(F \cap G)$, $(F \cup G)$, $(\overline{F})$, $(F \setminus G)$, and $|F|$ using caches, are well known, and are provided by the SMART MDD library.

Algorithms for computing $proj_{\vee 2}$, $proj_{\vee 3}$ (existential quantification), $\bigcup_i F_i$ (unions of many sets), and $DC_n(\ldots)$ (insertion of don't-care parameters), are described below, after algorithm A and algorithm 1.

5.2 Saturation based anti-bisimulation algorithm

This is our novel algorithm which operates by reducing a bisimulation problem to a transitive closure problem. It is a customization of the generic *SaturationClosure* algorithm, for the case of bisimulation.

Given decision diagrams representing a state space S, a set of deterministic transition relations $\{\mathcal{T}_\alpha \in S \times S : \alpha \in \mathcal{E}\}$, it produces a decision diagram representing the maximum bisimulation $\sim \subseteq S \times S$ where $\langle s_1, s_2 \rangle \in \sim$ iff s_1 and s_2 are bisimilar.

Given that a diagram representing S has L levels, a diagram representing any $\mathcal{T}_\alpha \subseteq S \times S$ has $2L$ levels and a diagram representing $\sim$ has $2L$ levels.

We produce $\sim$ as follows: Let $\sim \leftarrow (S \times S) \setminus \mathit{BiSat3}(\ S \times S,\ T_{[\mathcal{E}]},\ \overline{\mathcal{B}_{init}}\ )$,

where $\overline{\mathcal{B}_{init}} = \bigcup_{\alpha \in \mathcal{E}} (\mathcal{S}_\alpha \times (S \setminus \mathcal{S}_\alpha)) \cup ((S \setminus \mathcal{S}_\alpha) \times \mathcal{S}_\alpha)$ and where $\mathcal{S}_\alpha = \{s \in S | \exists s' : s \xrightarrow{\alpha} s'\}$ note: $\mathcal{S}_\alpha = \{s : \alpha \text{ is enabled in } s\}$

and where $\forall \alpha \in \mathcal{E} : T_{[\alpha]}(i, j, i', j') = \mathcal{T}_{[\alpha]}(i, j) \wedge \mathcal{T}_{[\alpha]}(i', j')$ note: (The $T_{[\alpha]}$ are pair transition relations $T_{[\alpha]} : (S \times S) \longrightarrow (S \times S)$) note also:(Levels of $T_{[\alpha]}$ should be interleaved in order: i_L, j_L, i'_L, j'_L, ..., i_1, j_1, i'_1, j'_1)

where *BiSat3* is computed as in Algorithm A.

```
MDD BiSat3( MDD S×S, MDD T_[E], MDD B_in ) is
  local MDD B, MDD B_old
   1 if B_in is leaves, return B_in
   2 if BiSat3(S×S, T_[E], B_in ) is found in cache, return cached value
   3 B ← B_in
   4 repeat:
   5   B_old ← B
   6   B ← BiSat3Saturate(S×S, T_[E], B )          • first saturate grandchildren
   7   B_old ← B
   8   for each α ∈ E where Top( T_α ) is this level, loop:
   9     B ← B ∪ (T_α(B) ∩ S×S)
  10     B ← BiSat3Saturate(S×S, T_[E], B )        • saturate result grandchildren
  11   end loop
  12 until B_old = B
  13 place B = BiSat3(S×S, T_[E], B_in ) into cache
  14 return B
```

Algorithm A, Bisimulation using Saturation (deterministic transitions)

```
MDD BiSat3Saturate( MDD S×S, MDD T_[E], MDD B_in ) is
  local mutable MDD node B
   1 B ← DFS copy of top 2 levels of B_in
   2 for each edge e to grandchild b of B, let:
   3   e ← edge to BiSat3( s×s, T_[E], b )
   4     where s×s = grandchild of S×S corresponding to b
   5 traverse B replacing each child B of B with unique(B), to re-canonicalize children
   6 return unique(B), canonical version of B
```

Helper function for Algorithm A

This algorithm has the same general structure as the generic *SaturationClosure* algorithm, and it operates in generally the same manner. Some important differences must be explained.

1. The recursive part of this algorithm calculates the anti-bisumulation, $\overline{\mathcal{B}} = \overline{\sim}$, the set of distinguishable pairs in $S \times S$. This is done by taking the closure of the relations $T_\mathcal{E}$ instead of $\mathcal{T}_\mathcal{E}$. The functions $T_\mathcal{E}$ produce distinguishable pairs in their range whenever their input is distinguishable pairs because of the following.

   Remembering that $T_{[alpha}(i, j, i', j') = \mathcal{T}_\alpha(i, j) \wedge \mathcal{T}_\alpha(i', j')$, we then assume that states j and j', (the input of T_α) are distinguishable (not bisimilar) to prove that the states in the range pair (i and i') are distinguishable. Since the $\mathcal{T}_{[\mathcal{E}]}$ are functions, j is the only state for which $\mathcal{T}_\alpha(i, j)$ holds, likewise, j' is the only state for which $\mathcal{T}_\alpha(i', j')$ holds. Consequently, i has no α-labeled transition to any state bisimilar to j', since it's only transition leads to j, which we

assumed is distinguishable from j'. But i' does have such a transition to j'. Likewise, i' has no α-labeled transition to any state bisimilar to j, yet i does. Thus, i and i' (the states in the range pair) are not bisimilar.

2. We start with an initial set $\overline{\mathcal{B}_{init}}$ of distinguishable pairs, while the classic algorithm starts with no pairs being assumed distinguishable. This is because the above technique of applying the $T_{[\mathcal{E}]}$ is incomplete for discovering all distinguishable pairs. In fact, it will not discover any distinguishable pairs without initial pairs. By inspecting the definition, one observes that $\overline{\mathcal{B}_{init}}$ produces all pairs where there is some transition enabled for one member of the pair, but not the other member. The pairs in $\overline{\mathcal{B}_{init}}$ are those pairs that would be found by the classic algorithm after the first complete round of splitting, without forwarding. Starting with $\overline{\mathcal{B}_{init}}$, the above procedure will produce all pairs of distinguishable states.

3. In this algorithm, the helper function descends 2 levels. This is because we know that $Top(T_\alpha)$ is never an odd number (for $\alpha \in \mathcal{E}$), due to the way the $T_{[\mathcal{E}]}$ are constructed. Consequently, these functions are never applied at odd numbered levels of $\overline{\mathcal{B}}$, so it is safe for the helper function to descend past those levels.

4. An additional parameter ($\underline{S \times S}$) is present, indicating the domain of which $\overline{\mathcal{B}}$ is a subset, and indicating the domain and range of the $T_{[\mathcal{E}]}$. Because of the way the $\mathcal{T}_{[\mathcal{E}]}$ are constructed, they may have a domain larger than S. Consequently, some of the $T_{[\mathcal{E}]}$ may have a range beyond $S \times S$. This being the case, line 9 uses an intersection with $\underline{S \times S}$ to restrict the result of the application to the desired range. This is what necessitates the presence of the additional parameter, and the way the helper function descends into it concurrently with descending into $\overline{\mathcal{B}}$.

5.3 Fixed-point minimization algorithm

This is a classic symbolic algorithm for relational coarsest partition. It is essentially the same algorithm as that used by Bouali and De Simone [1].

Given representations of the following components of a labeled transition system:

$S \triangleq$ The state space of the system.

$Q_t(a,b) \triangleq$ (a transition from a to b exists, with label t) The transition matrix of a graph, and:

$E(a,b) \triangleq$ (a and b are in the same class) An existing partition of the same graph,

Define a function $E' = Refine(E)$ as follows:

$E'(a,b) = E(a,b) \wedge \Delta E(a,b)$, where:

$\forall t : T_t(a,b) = \bigvee c : (Q_t(a,c) \wedge E(b,c))$ a transition (labeled t) exists from a to b's equivalence class

$\Delta E(a,b) = \bigwedge t : \bigwedge c : (T_t(a,c) = T_t(b,c))$ a and b have the same transitions to the same equivalence classes.

Finally, starting with an initial partition based on state labels, $E_0(s_1, s_2) = (s_1$ and s_2 have the same state label), Calculate the closure of $Refine$ on E_0, yielding $E_{n_{max}} = Refine(E_{n_{max}})$, where $E_{n+1} = Refine(E_n)$.

E_0 is the equivalence relation for the relational coarsest partition of the given labeled transition system.

In practice, both algorithms are integrated, and we don't use state labels. Consequently, as implemented, our algorithm is more similar to the following:

With the integrated implementation, it becomes a simple matter to describe Forwarding:

In the *Refine* algorithm, the objective is to construct a refinement of the existing partition. We represent the partition as an binary equivalence relation $\mathcal{E}$, among members of $\mathcal{S}$, the state space, so that any two members of the same partition block, will occur in a pair of the relation $\mathcal{E}$. Since unlabeled states can only be distinguished by which states are the targets of

```
MDD Refine( ℕ 𝒩_transition_labels, MDD 𝒬_[], MDD ℰ, MDD 𝒮 ) is
  local MDD 𝒯_[], MDD Δℰ‾, MDD ℰ',
  1 Δℰ‾ ← ∅
  2 for t ∈ [0,𝒩_transition_labels) loop
  3   𝒯_t ← proj_∨3((DC_2(𝒬_t,𝒮)) ∩ (DC_1(ℰ,𝒮)))        • 𝒯_t(a,b) = ⋁ c : (𝒬(a,c) ∧ ℰ(b,c))
  4   Δℰ‾ ← Δℰ‾ ∪ proj_∨3(DC_2(𝒯_t,𝒮)⊎DC_1(𝒯_t,𝒮))   • Δℰ(a,b) = ⋀ t ∈ [0,t] : ⋀ c : (𝒯_t(a,c) = 𝒯_t(b,c))
  5 end loop
  6 ℰ' = ℰ \ Δℰ‾                                      • ℰ' = ℰ ∩ Δℰ
  7 return ℰ'
```

```
MDD RefineClosure( ℕ 𝒩_transition_labels, MDD 𝒬_[], MDD 𝒮, ℕ 𝒩_state_labels, MDD 𝒮ℒ_[] ) is
  local MDD ℰ_0, MDD ℰ, MDD ℰ_old,
  1 ℰ_0 ← ∅                                            • Construct initial partition ℰ_0
  2 for l ∈ [0,𝒩_state_labels) loop
  3   ℰ_0 ← ℰ_0 ∪ DC_2(𝒮ℒ_l, 𝒮ℒ_l)
  4 end loop
  5 ℰ ← ℰ_0                                            • ℰ(a,b) = ∃l ∈ [0,𝒩_state_labels) : (a ∈ 𝒮ℒ_l ∧ b ∈ 𝒮ℒ_l)
  6 repeat                                             • ℰ ← Refine( ℰ ) until ℰ = Refine( ℰ )
  7   ℰ_old ← ℰ
  8   ℰ ← Refine( 𝒩_transition_labels, 𝒬_[], ℰ, 𝒮 )
  9 until ℰ = ℰ_old
  10 return ℰ                                          • E_{n_max}
```

Algorithm 1, Closure of partition refinement.

```
MDD RefineClosure( ℕ 𝒩_transition_labels, MDD 𝒬_[], MDD 𝒮 ) is
  local MDD ℰ, MDD ℰ_old, MDD 𝒯_[], MDD Δℰ‾
  1 ℰ ← DC_2(𝒮,𝒮)                                     • ℰ(a,b) = a ∈ 𝒮 ∧ b ∈ 𝒮
  2 repeat                                             • ℰ ← Refine( ℰ ) until ℰ = Refine( ℰ )
  3   ℰ_old ← ℰ
  4   Δℰ‾ ← ∅
  5   for t ∈ [0,𝒩_transition_labels) loop
  6     𝒯_t ← proj_∨3((DC_2(𝒬_t,𝒮)) ∩ (DC_1(ℰ,𝒮)))      • 𝒯_t(a,b) = ⋁ c : (𝒬(a,c) ∧ ℰ(b,c))
  7     Δℰ‾ ← Δℰ‾ ∪ proj_∨3(DC_2(𝒯_t,𝒮)⊎DC_1(𝒯_t,𝒮))  • Δℰ(a,b) = ⋀ t ∈ [0,t] : ⋀ c : (𝒯_t(a,c) = 𝒯_t(b,c))
  8   end loop
  9   ℰ = ℰ \ Δℰ‾                                      • ℰ = ℰ ∩ Δℰ
  10 until ℰ = ℰ_old
  11 return ℰ                                          • E_{n_max}
```

Algorithm 1.1, Closure of partition refinement.

their transitions, we compose a state transition relation $\mathcal{Q}_t$ and the partition equivalence relation $\mathcal{E}$. This composition $\mathcal{T}_t$ is described on line 3. The relation $\mathcal{T}_t = \{\langle a,b\rangle | \bigvee c : (\mathcal{Q}_t(a,c) \wedge \mathcal{E}(b,c))\}$, is clearly a composition of $\mathcal{Q}_t$ and $\mathcal{E}^{-1}$, since whenever $\mathcal{T}_t(a,b)$, $\mathcal{Q}_t(a,c) \wedge \mathcal{E}^{-1}(c,b)$ for some c. Since $\mathcal{E}$ is an equivalence relation, $\mathcal{E} = \mathcal{E}^{-1}$, therefore we have $\mathcal{T}_t = \mathcal{Q}_t \circ \mathcal{E}$. The meaning of $\mathcal{T}_t(a,b)$ is that state a has a transition to the equivalence class containing b.

In the final partition $E_{n_{max}}$, two states a and b will be distinguishable (be in separate partition classes) if they do not have transitions to the same set of partition classes. We would therefore like to know when two states have transitions to the same partition classes, according to the current partition $\mathcal{E}$. So, in line 4, we construct $\overline{\Delta\mathcal{E}}$ for this purpose. $\overline{\Delta\mathcal{E}}(a,b)$ holds only if a and b have some difference c in the classes to which they transition. This is clearly the case, by definition of $\uplus$, $\overline{\Delta\mathcal{E}}(a,b)$ implies $\mathcal{T}_t(a,c) \neq \mathcal{T}_t(b,c)$, for some c. This means that either a has a transition to cs equivalence class and b does not, or that b has a transition to cs class and a does not. $\overline{\Delta\mathcal{E}}$ therefore accumulates pairs of states we are determining do not belong in the same class. We then refine $\mathcal{E}$ to be more like $E_{n_{max}}$ in line 6 of *Refine*, by removing from it those pairs of states accumulated in $\overline{\Delta\mathcal{E}}$.

```
MDD RefineClosure( ℕ 𝒩_transition_labels, MDD 𝒬_[], MDD 𝒮 ) is
  local MDD ℰ, MDD ℰ_old, MDD 𝒯_[], MDD Δℰ
 1 ℰ ← DC_2(𝒮,𝒮)                                          • ℰ(a,b) = a ∈ 𝒮 ∧ b ∈ 𝒮
 2 repeat                                                  • ℰ ← Refine*( ℰ ) until ℰ = Refine( ℰ )
 3   ℰ_old ← ℰ
 4   for t ∈ [0,𝒩_transition_labels) loop
 5     𝒯_t ← proj_∨3((DC_2(𝒬_t,𝒮)) ∩ (DC_1(ℰ,𝒮)))          • 𝒯_t(a,b) = ⋁ c : (𝒬(a,c) ∧ ℰ(b,c))
 6     Δℰ ← proj_∨3(DC_2(𝒯_t,𝒮)⊎DC_1(𝒯_t,𝒮))               • Δℰ(a,b) = ⋀ c : (𝒯_t(a,c) = 𝒯_t(b,c))
 7     ℰ = ℰ \ Δℰ                                          • ℰ = ℰ ∩ Δℰ (update ℰ for use by next iteration of inner loop)
 8   end loop
 9 until ℰ = ℰ_old
10 return ℰ                                                • E_{n_max}
```

Algorithm 1.2, Closure of partition refinement, with forwarding.

In the *RefineClosure* algorithm, lines 1-4 construct the initial partition $\mathcal{E}_0$ based on the state labels. Lines 6-9 iteratively refine the current partition, until the termination condition, $\mathcal{E} = \mathcal{E}_{old} = Refine(\ \mathcal{E}_{old}\ )$, is met. Since the initial partition is finite, and the *Refine* algorithm operates by removing members of $\mathcal{E}$, *RefineClosure* is guaranteed to terminate. The termination condition $\mathcal{E} = Refine(\ \mathcal{E}\ )$ guarantees that upon termination, $\mathcal{E}$ meets requirements 1, 2, and 3 listed above, for relational coarsest partition. Showing that requirement 4 is met is beyond the scope of this current writing.

Algorithm 1.1 changes *RefineClosure* to incorporate the content of *Refine*, while algorithm 1.2 applies forwarding to the closure loop. Here, $\mathcal{E}$ is updated on line 7 in the inner loop. Consequently, the calculations on lines 5 and 6 in the next iteration can benefit from using the recently updated value of $\mathcal{E}$.

5.4 Algorithm for $proj_{\vee 3}$ (existential quantification)

Here is our algorithm to calculate a binary relation $F' = proj_{\vee 3}(F)$, by projecting a parameter from a 3-ary relation F, such that $F'(x,y) = \bigvee a : F(x,y,a)$.

Given a (possibly) interleaved DD representing a set F, produce an DD representing the projection/quantification $F' = \vee F$ of F, satisfying the following:

$(\vee F)(x,y) \triangleq \bigvee a : F(x,y,a)$ project third variable by disjunction)

We will divide the nodes of the graph representing F into three classes:

1. ordinary nodes, at the level of variables of parameter x, not quantified over
2. ordinary nodes, at the level of variables of parameter y, not quantified over
3. reducible nodes, at the level of variables of parameter a, being existentially quantified

We will assume that (the data structure for) each node is identifiable in constant time as either ordinary or reducible.

Additionally, we will assume that there are no nodes which represent empty sets; Only NULL edges will be used to represent empty sets, which can be identified in constant time.

The following recursive algorithm is functionally equivalent to a non-recursive algorithm that starts at the leaf level, and while proceeding toward the root, performs (recursive) union projections on reducible nodes. The recursive implementation is unavoidable, as our BDD package only supports the functional style, that is, it does not support mutation of existing structures. Its helper function *operateUnion*, is described in the next subsection.

Lines 4-6 of this algorithm directly access the children of the input tree. Due to the cache memoization, this algorithm

```
MDD projectUnion( MDD F ) is
  local  MDD  G
  local  MDD  H
  local  MDD  R
 1 if F = ∅ then return ∅;                                          • handle empty set
 2 if F is a leaf then return F;                                    • handle trivial (leaf) case (true or false)
 3 if projectUnion( F ) is found in the cache, return projectUnion( F )
 4 construct a new node G as follows:                               • G ← node where child i = projectUnion( child i of F )
 5   for each edge slot H in F with index i
 6     place projectUnion( H ) into G with index i
 7   G ← unique( G )                                                • use unique table
 8 if F references an ordinary node then
 9   R ←G
10 else F is not ordinary ... F is reducible
11   R ← operateUnion( G )                                          • takes the union of all Gs children
12 endif
13 place R = projectUnion( F ) into cache
14 return R
```

Algorithm 2, existential quantification to project out all reducible nodes

efficiently does a depth first traversal of the input tree, copying the subtree as it proceeds. Whenever the depth first traversal is leaving a "reducible" node, the reduction is processed by calling the *operateUnion* function on line 11.

5.5 Algorithm for calculating $\bigcup_i F_i$ (unions of many sets)

This algorithm to union many sets, represented as MDDs, attempts to make good use of the union cache. The general problem of how to optimally use the union cache is a hard one. Here we use a simple greedy algorithm, that depends on an ordering of MDDs. We attempt to take unions of "near" MDDs before other unions, and hopefully populate the union cache with unions we are likely to re-use.

```
MDD operateUnion( MDD S ) is                                        • members of S only have ordinary nodes
  local  sorted MDD list  H[]                                       • any consistent total ordering will do, we use node address
  local  ℕ  n
 1 if S = ∅ then return ∅
 2 forall s ∈ children of S, insert s into H                        • put Ss children into a sorted list
 3 main loop:
 4 while length(H) > 1 loop
 5   n ← length(H)
 6   remove duplicates from H
 7   if duplicates were found, then continue with next iteration of main loop
 8   if ∃l ∈ [1,n) : ∃i ∈ [1,n−l] : H[i] ∪ H[i+l] is in cache, then  • try to use union cache
 9     let l be smallest such l, and then let i be smallest such i  • prefer unions of closer elements
10     remove H[i] and H[i+l] from H, but insert H[i] ∪ H[i+l] from cache
11     continue with next iteration of main loop
12   end if
13   remove H[1] and H[2] from H, but insert H[1] ∪ H[2]
14 end loop
15 return H[1]                                                      • the one remaining element
```

Algorithm 3, union of many MDDs.

We attempt to take the union of all children of the input MDD in a manner that takes maximum advantage of the union cache, which stores the results of various past union operations. In line 8 we attempt to find union cache entries that will

provide partial results for this calculation. By using a sorted list and by the conditions on line 9 we attempt to evaluate unions, that may be relatively easy to re-use in some later invocation. It is believed that duplicate entries occur frequently. Using a sorted list allows the removal of duplicates on line 6 to occur efficiently, and thereby avoid unnecessary work.

5.6 Algorithm for calculating $DC_n(\ldots)$ (insertion of don't-care parameters)

This algorithm is used to introduce don't-care variables into the decision diagram representation of a relation.

There is a parameter F giving the relation. Even don't-care variables must belong to some specific set, so there is also a parameter G, giving the set from which the don't-care variables values are drawn. Our algorithm description gives no explicit parameter to designate the levels into which the don't care variables are inserted (although our actual code does). As described here, our algorithm solves the following problem:

Given decision diagrams representing relations F and G, construct a (possibly interleaved) diagrams representing relation F', where

$F'(\ldots$ f-variables and g-variables $\ldots) = F(\ldots$ f-variables $\ldots) \wedge G(\ldots$ g-variables $\ldots)$

This is a generalization that covers all the cases:

$F'(a, x, y) = F(x, y) \wedge G(a)$ inserts don't-care parameter (from G) into position 1 ($DC_1(F, G)$)

$F'(a, x, y, z) = F(x, y, z) \wedge G(a)$ inserts don't-care parameter (from G) into position 1 ($DC_1(F, G)$)

$F'(x, a, y, z) = F(x, y, z) \wedge G(a)$ inserts don't-care parameter (from G) into position 2 ($DC_2(F, G)$)

$F'(x, a, y) = F(x, y) \wedge G(a)$ inserts don't-care parameter (from G) into position 2 ($DC_2(F, G)$)

$F'(x, a) = F(x) \wedge G(a)$ inserts don't-care parameter (from G) into position 2 ($DC_2(F, G)$)

$F'(x, y, a) = F(x, y) \wedge G(a)$ inserts don't-care parameter (from G) into position 3 ($DC_3(F, G)$)

$F'(x, y, a, z) = F(x, y, z) \wedge G(a)$ inserts don't-care parameter (from G) into position 3 ($DC_3(F, G)$)

$F'(x, y, z, a) = F(x, y, z) \wedge G(a)$ inserts don't-care parameter (from G) into position 4 ($DC_4(F, G)$)

I assume that a diagram representing F' has L levels, and that L is also the sum of the total number of variables in Fs and Gs parameters.

We produce F' as follows: Let $F' \leftarrow DC(\ L,\ F,\ G\ )$,

where DC is computed as in:

This recursive layer-by-layer construction of the output MDD constructs each layer by considering which variable belongs in the current layer using the test on line 5. The current layer of F is copied to the output MDD by lines 6-10 for a layer whose variable belongs to input F. Lines 12-15 deal with the case where the variable of the current layer belongs to G. When we use a node from F or G, we descend to its child for the recursive call. Lines 3-4 describe the relatively trivial case for leaves. Given that F' has L levels, L being the total number of variables in Fs and G, we will recursively reach the leaves of Fs and G when they are needed for use on lines 3-4.

```
MDD DC( ℕ L, MDD F, MDD G ) is
  local  MDD  R
   1 if F = ∅ ∨ G = ∅, return ∅
   2 if DC( L, F, G ) is found in cache then return DC_n( L, F, G ) from cache
   3 if level L of the output is leaves, then                    • F and G will also be leaves
   4   R ← 1
   5 else if level L of the output corresponds to input parameter F then
   6   create a new MDD node R
   7   for each child f of F with index i, loop
   8     place a corresponding child DC( L − 1, f, G ) into R, with index i
   9   end loop
  10   end if
  11 else (level L of the output corresponds to parameter G)
  12   create a new MDD node R
  13   for each child g of G with index i, loop
  14     place a corresponding child DC( L − 1, E, g ) into R, with index i
  15   end loop
  16 end if
  17 R ← unique( R )
  18 place R = DC( L, F, G ) into cache
  19 return R
```

Algorithm 4, insertion of don't care variables.

5.7 Number of equivalence classes in an interleaved equivalence relation

It is a relatively simple matter to calculate the number of equivalence classes in an equivalence relation represented in a non-interleaved manner. Here we give the algorithm we use to count equivalence classes in an interleaved equivalence relation.

Given an interleaved decision diagram representing an equivalence relation $E \subseteq S \times S$, calculate the number $|\{\{x|E(x,y)\}|y \in S\}|$ of equivalence classes of E. Any total ordering $<$ can be used to associate to each block B, in E, some unique minimum element $b \in B$, $\forall c \in B : b \leq c$, so the problem can be reduced to counting the counting the minimum elements b, such that $\forall c \in B : b \leq c$, where $B = \{c|E(b,c)\}$. Hence, we actually calculate:

$N_E = |\{\ b \in S \mid \forall c \in \{c|E(b,c)\} : b \leq c\ \}|$

As our ordering relation $<$, we will use the lexicographic ordering of states. Now we define:

$(proj_{\vee 2}(F))(x) \triangleq \bigvee a : F(x,a)$ project second variable (of 2) by disjunction

$(selectNonMin(F))(b,c) \triangleq b \not\leq c \wedge F(b,c)$ select pairs where the first element is not minimal

$mag(F) \triangleq \Sigma_{x \in S|F(x)} 1$ magnitude of set

Using these definitions, the calculation can be performed as: $N_E \leftarrow mag(\ S \setminus proj_{\vee 2}(\ selectNonMin(E)\ )\ )$

We now give the algorithm for computing $selectNonMin(E)$. I assume that the variable levels in E occur in pairs, with the same ordering for both parameters of E, the variable from the domain being first (closer to the root), followed by the corresponding variable from the range, the most significant variables being toward the root.

The function selects the subset of the tree corresponding to the case where the variable on the even level has a value b_0, greater than the value c_0 of the child on the odd level, taking two levels at a time as controlled by the loop statements on lines 6 and 9. If the inequality holds, as tested on line 11, the entire subtree is selected without further testing. If the opposite inequality holds, the entire subtree is rejected without further testing. Only in the case where the two values are

```
MDD selectNonMin( MDD E_b ) is
  local  MDD  R_b
  local  MDD  R_c
   1 if selectNonMin( E_b ) is in cache, return selectNonMin( E_b ) from cache
   2 if E_b is a leaf, then
   3   R_b ← ∅                                  • if we got here, the values are equal, so eliminate
   4 else                                       • not leaf
   5   R_b ← a new empty MDD node
   6   for each child E_c of E_b loop
   7     let b_0 be the value at this level for E_c within E_b in:
   8       R_c ← a new empty MDD node
   9       for each child E'_b of E_c loop
  10         let c_0 be the value at this level for E'_b within E_c in:
  11           if b_0 > c_0 then
  12             into R_c insert child E'_b, with value c_0
  13           else if b_0 = c_0 then
  14             into R_c insert child selectNonMin( E'_b ), with value c_0
  15           end if                           • insert nothing when b_0 < c_0
  16       end loop
  17       into R_b insert child unique( R_c ), with value b_0
  18   end loop
  19 end if
  20 R_b ← unique( R_b )
  21 place R_b = selectNonMin( E_b ) into cache
  22 return R_b
```

Algorithm 5, selection of non-minimum elements

equal as tested on line 13, the function is called recursively at line 14. There is therefore no way to arrive at a subtree, unless the values of variables considered at higher levels have occurred as equal up to this point. Therefore, when line 2 finds that the input is a leaf, line 3 can safely reject, since we are choosing to select for an inequality, while arriving at a leaf indicates equality of variable values.

5.8 Unified calculation of $(DC_2(Q,S)) \cap (DC_1(E,S))$, given $E \subseteq S \times S$

This algorithm produces the desired result $(DC_2(Q,S)) \cap (DC_1(E,S))$, without constructing any decision diagrams, such as $DC_2(Q,S)$ or $DC_1(E,S)$. This algorithm operates by syncronized recursive traversal of the structures representing the three inputs S, Q, and E as the output is constructed recursively.

Given decision diagrams representing a transition relation $Q \subseteq \hat{S} \times \hat{S}$, and an equivalence relation $E \subseteq S \times S$, this algorithm produces a decision diagram representing the unquantified composition $t = (DC_2(Q,S)) \cap (DC_1(E,S))$ of Q and E, limited to S. Expanding the definitions of DC_2 and DC_1, we see that:

$\forall a,b,c \in \hat{S} : t(a,b,c)$ iff $(Q(a,c) \wedge S(b)) \wedge (E(b,c) \wedge S(a))$

Since $E \subseteq S \times S$, $E(b,c)$ guarantees $S(b)$ (and $S(c)$), we may eliminate $S(b)$, deriving:

$\forall a,b,c \in \hat{S} : t(a,b,c)$ iff $Q(a,c) \wedge E(b,c) \wedge S(a)$

I assume that the variable levels in t and Q corresponding to parameters a and c occur in the same ordering, and also that the variable levels in t and E corresponding to parameters b and c occur in the same ordering.

Assume that a diagram representing S has L levels, a diagram representing Q or E has $2L$ levels and a diagram representing

t has $3L$ levels.

We produce t as follows: Let $t \leftarrow UcompL(\ 3L,\ Q,\ E,\ S\ )$,

where $UcompL$ is computed as in Algorithm 6:

```
MDD UcompL( ℕ N, MDD Q, MDD E, MDD S ) (memoized function) is
  local  MDD  R
   1 if UcompL( N, Q, E, S ) is found in cache, return UcompL( N, Q, E, S ) from cache
   2 if level N of the output is leaves, then                          • Q and E will also be leaves
   3   R ← Q ∩ E, using leaf rules
   4 else if level N of the output corresponds to parameter a then     • DC1(E) has a don't care at this level
   5   if Q = ∅ ∨ S = ∅ ∨ (Q and S have no children with same index) then R ←∅ , else:
   6     create a new MDD node R
   7     for each child q of Q and child s of S, having the same index, loop   • use only members of Q where a ∈ S
   8       place a corresponding child UcompL( N − 1, q, E, s ) into R
   9     end loop
  10   end if
  11 else if level N of the output corresponds to parameter b then     • DC2(Q) has a don't care at this level
  12   if E = ∅ then R ←∅ , else:
  13     create a new MDD node R
  14     for each child e of E loop
  15       place a corresponding child UcompL( N − 1, Q, e, S ) into R
  16     end loop
  17   end if
  18 else (level N of the output corresponds to parameter c)
  19   if Q = ∅ ∨ E = ∅ ∨ (Q and E have no children with same index) then R ←∅ , else:
  20     create a new MDD node R
  21     for each child q of Q and child e of E, having the same index, loop
  22       place a corresponding child UcompL( N − 1, q, e, S ) into R
  23     end loop
  24   end if
  25 end if
  26 R ← unique( R )
  27 place R = UcompL( N, Q, E, S ) into cache
  28 return R
```

Algorithm 6, unquantified composition, limited to reachable states S.

As in algorithm 4, we recursively descend the input MDDs, choosing how to construct each level of the output, based on whose variables occur at that level. At each level, we clone the structure of each input MDD involved with the variable at that level, always using the intersection, in the case where more than one input MDD is involved. When ever we clone a level of an input, we descend to its children recursively when we descend to construct the next (lower) level of the output. Uninvolved input MDDs are passed unchanged to the recursive call that constructs a child at the next level.

Lines 2,4, and 11 determine the inputs of the variable at the current level of the output being constructed. Lines 5-10 clone the output layer from input MDDs Q and S, since the layer corresponds to a variable in parameter a. Note the recursive call on line 8 passes the children of the cloned inputs, but passes the other inputs directly. Similarly, lines 12-17 clone the output layer from E, since a variable from parameter b is represented at this level. And lines 19-24 clone from Q and E since a variable from parameter c is found. If all operates correctly, the leaf level of the output is found when the leaf levels of the inputs are present, So that this case can be handled by lines 2 and 3.

5.9 Hybrid fixed-point minimization algorithm

This algorithm for relational coarsest partition uses interleaved MDDs to represent the transition relations Q_t, while using non-interleaved MDDs to represent the partition E of the state space.

It is similar to algorithm 1.2, except for the part of the inner loop dealing with partition calculation.

Given representations of the following components of a labeled transition system:

$S \triangleq$ The state space of the system.

$Q_t(a,b) \triangleq$ (a transition from a to b exists, with label t) The transition matrix of a graph, and:

$E(a,k) \triangleq$ (a is in class numbered k) An existing partition of the same graph,

Define functions $E' = RefineH_t(E)$ as follows:

let $T_{partial(t)}(a,c,d) \triangleq \bigvee b : (Q_t(a,b) \wedge E(a,c) \wedge E(b,d))$ a transition (labeled t) exists from a (in class c), to some b, in class d

let $W(a,c,d) \triangleq E(a,c) \wedge d = 0$ a relation just like E, except with a third member $d = 0$, to match the form of $T_{partial(t)}$

let $T_t \triangleq T_{partial(t)} \cup W$ associate the signature $\{\langle c,d\rangle | T_t(a,c,d)\} \cup \{\langle c,0\rangle | E(a,c)\}$ with each state a, where c is the class number in E of a, and the d's are the class numbers of states reachable from a by transition t.

Let $Sig(a)$ be the signature $\{\langle c,d\rangle | T_t(a,c,d)\}$ associated with state a

$newmap(s) = j \in \mathbb{N}^+$, a function that associates a unique positive integer (a new class number) with each different signature s, so that $\forall s_1, s_2 \in \{Sig(a) | a \in domain(E)\} : newmap(s_1) = newmap(s_2) \iff s_1 = s_2$

$E'(a,k) \triangleq k = newmap(Sig(a))$ a new partition that associates each state a with its new class number

Now define the function $RefineH$ as: $RefineH_1 \circ RefineH_2 \circ \ldots$, composing $RefineH_t$ for all transition labels t

Finally, starting with an initial partition based on state labels, $E_0(s,k) =$ (s has the kth state label), Calculate the closure of $RefineH$ on E_0, yielding $E_{n_{max}} = RefineH(E_{n_{max}})$, where $E_{n+1} = RefineH(E_n)$.

$E_{n_{max}}$ is the equivalence relation for the relational coarsest partition of the given labeled transition system. This is calculated using the following variation on algorithm 1.2. It uses a helper function, *SigRenum*, to renumber the partition classes.

Although this algorithm appears very similar to algorithm 1.2, it is optimized to support non-interleaved ordering for representation of the equivalence relation, while the transition relation Q_t is represented as an interleaved MDD. The interleaving intrinsic to Q_t is eliminated in line 6, where the variables and levels corresponding to parameter b of Q_t are eliminated by projection. As only the parameters a and b of Q_t were interleaved, no interleaving remains. The explanation of the signature relation of Algorithm 1 also applies here to some extent. For this $\mathcal{T}_{partial[t]}$ constructed in line 6, the meaning is similar to the meaning of $\mathcal{T}_t$ from Algorithm 1, except numbers are used to represent partitions classes in $\mathcal{E}$. $\mathcal{T}_{partial[t]}$ (a, b) means that a has a transition to the class containing b. There is one complication. The construction of $\mathcal{T}_{partial[t]}$ gives it the same domain as Q_t, which may not be the entire state space S. Also, this algorithm does not operate by removing pairs from E as does Algorithm 1. This algorithm operates by replacement of E. As it would be unacceptable to lose members of the state space from the domain of E, we use an extra MDD $\mathcal{W}$ to ensure that members are not lost. Line 5 constructs $\mathcal{W}$ with the same domain as $\mathcal{E}$, using a distinct unused partition class number as the only element of the range. This enters into the total signature relation $\mathcal{T}_t$ in line 7. The call to *SigRenum* on line 8 finally assigns a new class number to each equivalence class, as refined using the total signature relation $\mathcal{T}_t$

```
MDD HybridClosure( ℕ 𝒩_transition_labels, MDD 𝒬_[], MDD 𝒮 ) is                    • (Calculate E_{n_max})
  local MDD ℰ, MDD ℰ_old, MDD 𝒲, MDD 𝒯_partial[], MDD 𝒯_[]
 1 ℰ ← DC_2(𝒮, {1})                          • ℰ(a,1) for all a, (put all states it the only class (1) of the partion)
 2 repeat                                    • ℰ ← RefineH*( ℰ ) until ℰ = RefineH( ℰ )
 3   ℰ_old ← ℰ
 4   for t ∈ [0, 𝒩_transition_labels) loop
 5     𝒲 ← DC_3(ℰ, {0})                      • has the same range as ℰ
 6     𝒯_partial[t] ← proj_∨2(DC_4(DC_3(𝒬_t, [0,|𝒮|]), [0,|𝒮|]) ∩ DC_4(DC_2(ℰ, 𝒮), [0,|𝒮|]) ∩ DC_1(DC_2(ℰ, [0,|𝒮|]), 𝒮))
       • 𝒯_partial[t](a,c,d) = ⋁ b : (𝒬_t(a,b) ∧ ℰ(a,c) ∧ ℰ(b,d))
 7     𝒯_[t] ← 𝒯_partial[t] ∪ 𝒲              • make the signature relation total by including range of ℰ
 8     ℰ ← SigRenum( 𝒯_[t] )                 • renumber partition classes
 9   end loop
10 until ℰ = ℰ_old
11 return ℰ                                  • E_{n_max}
```

Algorithm 7, Closure of hybrid partition refinement, with forwarding.

5.10 Unified calculation of: $Q \circ E$

(where $Q \circ E \triangleq DC_4(DC_3(Q,[0,|S|]),[0,|S|]) \cap DC_4(DC_2(E,S),[0,|S|]) \cap DC_1(DC_2(E,[0,|S|]),S))$

Given decision diagrams representing a transition relation $Q \subseteq \hat{S} \times \hat{S}$, and an equivalence relation $E \subseteq S \times [1,N]$, where $N = |S|$, produce a decision diagram representing the unquantified composition $t \subseteq S \times S \times [1,N] \times [1,N] = DC_4(DC_3(Q,[0,|S|]),[0,|S|]) \cap DC_4(DC_2(E,S),[0,|S|]) \cap DC_1(DC_2(E,[0,|S|]),S))$ of Q and E, so that:

$\forall a,b \in \hat{S}, c,d \in [1,N] : t(a,b,c,d)$ iff $Q(a,b) \wedge E(a,c) \wedge E(b,d)$

I assume that the variable levels in t and Q corresponding to parameters a and b occur in the same ordering, and also that the variable levels in t and E (the first occurrence) corresponding to parameters a and c occur in the same ordering. Also, of course, the variable levels in t and E (the second occurrence) corresponding to parameters b and d occur in the same ordering.

Assume that a diagram representing $\hat{S}$ or S has K levels, a diagram representing $[1,N]$ has $M = \lceil \log_2(N+1) \rceil$ levels, a diagram representing E has $K+M$ levels, a diagram representing Q has $2K$ levels, and a diagram representing t has $2K+2M$ levels.

We produce t as follows: Let $t \leftarrow$ *UcompH*($2K+2M$, Q, E, E),

where *UcompH* is computed as in Algorithm 8:

As with Algorithm 6 and Algorithm 4, *UcompH* recursively constructs its output, constructing each level by cloning corresponding parts of input MDDs. In this case, there are four parameters involved, necessitating the four cases (lines 5-10 for parameter a, lines 12-17 for b, lines 19-24 for c, and lines 26-31for d). Otherwise, the same principles generally apply.

5.11 Numbering of equivalence classes of a signature relation

Given a decision diagram representing an signature relation $E \subseteq S \times K$, assign a unique partition number n_k to each signature k of K in the range of E, and produce a partition relation E' such that $E'(s,n_k) = E(s,k)$. Conveniently, we can use the canonicity of decision diagrams to our advantage. Each unique signature value k in K will be represented by a unique DD. Consequently one need only traverse the diagram, representing E, and replace each new member of K encountered, with a new integer, obtained by incrementing a global counter. We use the built-in caching functions of the SMART MDD library, to memoize the function implementing our algorithm. This ensures that each occurrence of the same signature is replaced by

```
MDD UcompH( ℕ L, MDD Q, MDD E1, MDD E2 ) (memoized function) is
  local  MDD  R
 1 if UcompH( L, Q, E1, E2 ) is found in cache, return UcompH( L, Q, E1, E2 ) from cache
 2 if level L of the output is leaves, then                    • Q, E1 and E2 will also be leaves
 3   R ← Q ∩ E1 ∩ E2, using leaf rules
 4 else if level L of the output corresponds to parameter a then    • DC1(DC2(E)) has a don't care at this level
 5   if Q = ∅ ∨ E1 = ∅ ∨ (Q and E1 have no children with same index) then R ←∅ , else:
 6     create a new MDD node R
 7     for each child q of Q and child e of E1, having the same index, loop    • use only a where Q(a,*) and E(a,*)
 8       place a corresponding child UcompH( L − 1, q, e, E2 ) into R
 9     end loop
10   end if
11 else if level L of the output corresponds to parameter b then    • DC4(DC2(E)) has a don't care at this level
12   if Q = ∅ ∨ E2 = ∅ ∨ (Q and E2 have no children with same index) then R ←∅ , else:
13     create a new MDD node R
14     for each child q of Q and child e of E2, having the same index, loop    • use only b where Q(*,b) and E(b,*)
15       place a corresponding child UcompH( L − 1, q, E1, e ) into R
16     end loop
17   end if
18 else if level L of the output corresponds to parameter c then    • DC4(DC3(Q)), DC1(DC2(E)) have don't cares
19   if E1 = ∅ then R ←∅ , else:
20     create a new MDD node R
21     for each child e of E1 loop
22       place a corresponding child UcompH( L − 1, Q, e, E2 ) into R
23     end loop
24   end if
25 else (level L of the output corresponds to parameter d)    • DC4(DC3(Q)), DC4(DC2(E)) have don't cares
26   if E2 = ∅ then R ←∅ , else:
27     create a new MDD node R
28     for each child e of E2 loop
29       place a corresponding child UcompH( L − 1, Q, E1, e ) into R
30     end loop
31   end if
32 end if
33 R ← unique( R )
34 enter R = UcompH( L, Q, E1, E2 ) into cache
35 return R
```

Algorithm 8, an unquantified composition, using E two ways

the same unique partition number,

The function will be performed as follows: *init_SigRenum*(); $E' \leftarrow$ *SigRenum*(E), where *init_SigRenum*() and *SigRenum* are defined as in Algorithm 9:

As with many other algorithms, the algorithm *SigRenum* recursively copies its input MDD to its output while doing depth-first-traversal. This can easily be seen in lines 7-10. This algorithm expects signatures to occur at some known level of the MDD. In the output, this algorithm replaces each different signature with a unique partition class number. Each new class number is produced by incrementing a counter on line 4. The number is represented by a narrow (where each node has exactly one child, except the leaf) BDD. The BDD is constructed by using the bits from the binary representation of the number as the values of variables at the levels of the BDD.

```
init_SigRenum() is
  global  ℕ  counter
   1 clear cache for SigRenum( MDD ℰ )
   2 counter ← 0
```

```
MDD SigRenum( MDD ℰ ) (memoized function) is
  global  ℕ  counter
  local  MDD  ℛ
   1 if SigRenum( ℰ ) is found in cache, return SigRenum( ℰ ) from cache
   2 ℛ ← a new empty MDD node
   3 if ℰ is at the level of the signature in this DD, then
   4   counter ← counter + 1;                  • if we got here, the signature is new, assign unique number
   5     into ℛ insert child BDD representing value counter
   6 else                                      • not at signature level, just duplicate node, with function applied recursively
   7   for each child e of ℰ loop              • Use bits of counter to construct a narrow BDD
   8    let v be the value at this level for e within ℰ in:
   9     into ℛ insert child SigRenum( e ), with value v
  10   end loop
  11 end if
  12 ℛ ← unique( ℛ )
  13 enter ℛ = SigRenum( ℰ ) into cache
  14 return ℛ
```

Algorithm 9, renumbering of signatures

6 Design

The software implementation of this work is implemented as a unit of code within the latest revision of the SMART verification tool. Its activation is prompted by a `num_eqclass()` command in a model within a SMARTsource file.

The functionality provided to SMART by the software implementation is as follows. SMART calls the function:

```
bigint ComputeNumEQClass(state_model *mdl);
```

passing a `state_model *mdl` , which is a representation of a state transition system. This function, after calculating the relational coarsest partition for the given model, calculates the number of partition elements in the coarsest refinement, and returns that number, as a `bigint`, to SMART. A `bigint` is a data type capable of storing finite integers that may require very many digits for their representation.

1. The `ComputeNumEQClass` function is a glue function, which invokes an internal `ComputeNumEQClass` function.

   The internal `ComputeNumEQClass` function extracts the state space and the transition relations from its input, then, to produce the result, calls the following other internal functions:

2. `Mdd *BooleanInterleavedLumping( Mdd * * Qarray, int NQ, Mdd *S, int L );` calculates the relational coarsest partition, using the interleaved representation for the partition relation, given the array `Qarray` of interleaved MDDs representing the transition relations, the number `NQ` of transition relations, the MDD `S` for the state space, and the number `L` of levels in the MDD `S` .

3. `Mdd *BooleanHybridLumping( Mdd * * Qarray, int NQ, Mdd *S, int L );` calculates the relational coarsest partition, using the non-interleaved representation for the partition relation, given the array `Qarray` of interleaved MDDs representing the transition relations, the number `NQ` of transition relations, the MDD `S` for the state space, and the number `L` of levels in the MDD `S` .

4. `Mdd * SaturationBisimulationNEG( Mdd * * Qarray, int NQ, Mdd *S, int L );` calculates the relational coarsest partition, using the interleaved representation for the partition relation, given the array `Qarray` of interleaved

MDDs representing the transition relations, the number `NQ` of transition relations, the MDD `S` for the state space, and the number `L` of levels in the MDD `S` .

5. `bigint CountEQClasses( Mdd * E, Mdd * S )` calculates the number of partition elements, given the interleaved MDD `E`, representing the partition, and the MDD `S`, representing the state space.

6. `Mdd *sigrenumQ( Mdd *E, int level, int & count )` calculates the number of partition elements, given the non-interleaved MDD `E`, representing the partition, and the MDD level `level`, indicating which levels of `E` store signatures.

The other internal functions operate as follows:

7. The `BooleanInterleavedLumping` function performs the *RefineClosure* algorithm, as shown in Algorithm 1.2. It calls `InsertDontCaresQQ` to construct the initial partition ($S \times S$). It calls `RelationalComposeInterleavedQQ` to perform the composition calculation on line 5 of Algorithm 1.2. It calls `NegEquivClassInterleavedQQ` to perform the class construction calculation on line 6 of Algorithm 1.2.

8. The `BooleanHybridLumping` function performs the *HybridClosure* algorithm, as shown in Algorithm 7. It calls `sigrenumQ` to construct the initial partition ($DC_2(\mathcal{S}, \{1\})$). It calls `InsertDontCaresQQ` to construct $\mathcal{W}$, as described on line 5 of Algorithm 7. It calls the `GenericComposeQQ` function followed by `ProjectUnionQQ` to perform the composition calculations on line 6 of Algorithm 7. It calls `sigrenumQ` to perform the class renumbering calculation on line 8 of Algorithm 7.

9. The `SaturationBisimulationNEG` function performs the *BiSat3* algorithm, as shown in Algorithm A. It calls `InsertDontCaresQQ` to construct $S \times S$, $(\mathcal{S}_\alpha \times (S \setminus \mathcal{S}_\alpha))$, $((S \setminus \mathcal{S}_\alpha) \times \mathcal{S}_\alpha)$, and $\mathcal{T}_{[\alpha]}(i,j) \wedge \mathcal{T}_{[\alpha]}(i',j')$ in algorithm A. It also calls the `ProjectUnionQQ` function to perform the projection $\{s \in S | \exists s' : s \xrightarrow{\alpha} s'\}$. It calls `RelationalComposeInterleavedQQ` to perform the application of function T_α to set $\overline{\mathcal{B}}$ on line 9 of Algorithm *BiSat3*.

10. The `CountEQClasses` function performs the calculation: $N_E \leftarrow mag(\ S \setminus proj_{\vee 2}(\ selectNonMin(E)\ )\ )$. It calls the `selectGT` function to perform the *selectNonMin* algorithm. It also calls the `ProjectUnionQQ` function to perform the projection ($proj_{\vee 2}$).

11. The `Mdd * RelationalComposeInterleavedQQ` function accepts the following parameters:

    `(`

    `Mdd *left,` a MDD representation of a first relation $\mathcal{Q}$

    `bool *Lleft,` a selection of surviving variables

    `Mdd *Uleft,` a MDD for the set of possible states of the surviving variables $\mathcal{S}_\mathcal{Q}$

    `Mdd *right,` a MDD representation of a second relation $\mathcal{E}$

    `bool *Lright,` a selection of surviving variables from the second relation

    `Mdd *Uright` a MDD for the set of possible states of the surviving variables from the second relation $\mathcal{S}_\mathcal{E}$

    `);` and returns a MDD representing the composition $\mathcal{T} \subseteq \mathcal{S}_\mathcal{Q} \times \mathcal{S}_\mathcal{E} = \mathcal{Q} \circ \mathcal{E}^{-1}$,

    such that $\mathcal{T}(q,e) = \bigvee c : (\mathcal{Q}(q,c) \wedge \mathcal{E}^{-1}(c,e)) = \bigvee c : (\mathcal{Q}(q,c) \wedge \mathcal{E}(e,c))$.

    It performs this composition as: $\mathcal{T} \leftarrow proj_{\vee 3}((DC_2(\mathcal{Q}, \mathcal{S}_\mathcal{E})) \cap (DC_1(\mathcal{E}, \mathcal{S}_\mathcal{Q})))$, and so can be used to implement the pseudo-code on line 5 of *RefineClosure* in Algorithm 1.2. It calls `GenericComposeQQ` to compute the MDD for $(DC_2(\mathcal{Q}, \mathcal{S}_\mathcal{E})) \cap (DC_1(\mathcal{E}, \mathcal{S}_\mathcal{Q}))$. It also calls `ProjectUnionQQ` to calculate the projection operation ($proj_{\vee 3}$).

12. The `Mdd *NegEquivClassInterleavedQQ` function accepts the following parameters:

    `(`

    `Mdd *T,` a MDD representing a signature relation $\mathcal{T}$. The set $\{b | \mathcal{T}(a,b)\}$ is the signature of a

    `bool * L,` a selection of variables representing the signature part of the relation $\mathcal{T}$

    `Mdd *U );` a MDD for $\mathcal{S}$ (the state space) the set of objects that have signatures in $\mathcal{T}$

    It returns an MDD representing the partition of those $\mathcal{S}$ into classes of members having the same signature. The returned MDD encodes the complement of the equivalence relation: $\mathcal{E}(s_1, s_2) \in \mathcal{S} \times \mathcal{S} = \bigwedge t : (\mathcal{T}(a,t) = \mathcal{T}(b,t))$. This output is calculated as: $\overline{\Delta\mathcal{E}} \leftarrow proj_{\vee 3}(DC_2(\mathcal{T}, \mathcal{S}) \uplus DC_1(\mathcal{T}, \mathcal{S}))$, and so may be used to implement the pseudo-code on line 6 of *RefineClosure* in Algorithm 1.2. It calls `InsertDontCaresQQ` to compute the MDDs for the expressions $DC_2(\mathcal{T}, \mathcal{S})$ and $DC_1(\mathcal{T}, \mathcal{S})$. It also calls `ProjectUnionQQ` to calculate the projection operation ($proj_{\vee 3}$).

13. The function `Mdd *InsertDontCaresQQ` accepts the following parameters:

    `( int L,` the height of (number of variables in) the MDD to be constructed.

    `Mdd *d,` the input MDD, for the set D of the values of the original variables

    `bool * inserts,` a selection of variables in the output MDD which will be don't-care variables

    `Mdd * U );` an MDD for the universe, from which values of don't care variables may be drawn

    It returns an MDD for a set D', where $D'(\ldots$ D-variables and U-variables $\ldots) = D(\ldots$ D-variables $\ldots) \wedge U(\ldots$ U-variables $\ldots)$. The result is calculated using the algorithm for DC, from the algorithms section, and so can be used to implement any of the forms of DC_1 and DC_2 found in the various algorithm pseudo-codes.

14. The `Mdd * ProjectUnionQQ` function accepts the following parameters:

    `(`

    `Mdd * arg,` a MDD for the set/relation F to be projected

    `bool * projectible );` a selection of levels of the variables to be eliminated by disjunction

    It returns a MDD for the projection F' of the relation F, such that $F'(\ldots$ surviving variables $\ldots) = \bigvee$ projectible variables : $F(\ldots$ surviving variables and projectible variables $\ldots)$. It uses the *projectUnion* algorithm previously given, and so can be used to implement all the various forms of $proj_\vee$ found in the above algorithms. It calls the `OperateUnionQQ` function to implement the *operateUnion* function found on line 11 of the *projectUnion* algorithm.

15. The `Mdd * OperateUnionQQ` function accepts an argument:

    `( Mdd * arg );` an MDD F

    The function returns a MDD for the union of all children of the top node of F. The union is calculated according to the algorithm for *operateUnion*, given above, and is used to implement line 11 of the *projectUnion* algorithm.

16. The `Mdd * selectGT` function accepts the parameter:

    `( Mdd * Eb );` a MDD, representing a set, E_b, of pairs of states that is an equivalence class

    and returns a MDD for a set E' of pairs such that $E'(b, c) = b \not\leq c \wedge E_b(b, c)$ (using lexicographic ordering). This result is calculated using the *selectNonMin* algorithm, previously given, and is used by the `CountEQClasses` function to determine the number of classes in an equivalence class.

17. The `Mdd *GenericComposeQQ` function accepts the parameters:

    `(`

    `Mdd *operand1,` a first operand

    `Mdd *operand0,` a second operand

    `Mdd *range,` a third operand, normally used to limit the function output

    `int N );` the number of levels for the output MDD

    and other parameters, encoding an operation description.

    This function performs a composition operation, among the three operands, producing a composed MDD output having `N` levels. This composition function evaluates compositions such as $(DC_2(Q, S)) \cap (DC_1(E, S))$, and $DC_4(DC_3(Q, [0, |S|]), [0, |S|]) \cap DC_4(DC_2(E, S), [0, |S|]) \cap DC_1(DC_2(E, [0, |S|]), S)$, in a recursive manner without constructing the MDDs representing any of the sub-expressions. Hence, it is able to efficiently implement the *UcompL* method of Algorithm 6, the *UcompH* method of Algorithm 8, and the application $T_\alpha(\overline{\mathcal{B}})$ in algorithm A, and is used for those purposes by the `RelationalComposeInterleavedQQ` function, the `BooleanHybridLumping` function, and the `SaturationBisimulationNEG` function.

18. The `Mdd *sigrenumQ` function accepts the following parameters: `( MDDL::MddNode *E,` A non-interleaved MDD representing a signature relation `int level,` The top level in E at (and below) which the signatures are represented `int & count );` A reference to the global counter as described in Algorithm 9.

    This function returns a MDD analogous to its input `E`, with each different signature (at level `level`) replaced by a unique class number, starting at 1. This replacement of signatures by unique numbers is performed according to the *SigRenum* method of Algorithm 9. The global counter `count` will contain the number of signatures/classes found in `E`, upon return from this function.

Note: Some omissions and contractions occur in the above, to simplify the presentation.

7 Project History

In our previous work, motivated by the need for more efficient lumping algorithms, we surveyed the literature relevant to lumping. Guided by the advisor's idea to use interleaved MDDs to represent partitions, we soon devised Algorithm 1, for bisimulation, and a variant, Algorithm X, for lumping. Imminency of deadlines forced a limitation of the scope of this project. As Algorithm X promised to require more effort, to implement AAMDDs or to convert EV*MDDs to EV+MDDs, we chose to first implement Algorithm 1.2, to gauge the feasibility of using interleaved MDDs to represent partitions. Our first implementation of *RefineClosure* used interleaved representation for all relations. After coding *RefineClosure*, we surveyed the literature relevant to this type of bisimulation, and surprisingly found that Algorithm 1 had been discovered by 1992 by Amar Bouali and Robert De Simone[1]. Amar Bouali and Robert De Simone also provide performance comparison, between use of interleaved BDDs, and use of non-interleaved BDDs. We then chose to compare with the performance of Algorithm H, which uses interleaved MDDs for the transition relations, and non-interleaved MDDs for the equivalence relation representing the partition of the state space. The equivalence relation is constructed in a manner similar to that used in the lumping algorithm described by Derisavi [6]. This is intended to provide a comparison between the use of interleaved and non-interleaved MDDs, that applies specifically to the equivalence relation. We also constructed some additional parametric sets of models to compare these bisimulation algorithms. The models we constructed tended to have few equivalence classes in their bisimulation partitions. The results we obtained indicated that our fully implicit bisimulation algorithm (that we had hoped to extend for use in lumping) performed relatively poorly compared to the hybrid algorithm, in all respects except for output representation size. After finding that several variations on the concept also did not produce significant improvement, we chose to explore the direct use of saturation methods to compute the complement of the maximum bisimulation, as indicated in our previous "Future Work" section. This exploration has produced Algorithm A in the current work. Inconveniently, the models we used to test Algorithm 1 and Algorithm H make use of non-deterministic transition relations. As Algorithm A only works with deterministic (functional) transitions, we are forced to use other models. We found that typical benchmark models, such as Dining philosophers, tend to have very many equivalence classes in their bisimulation partitions, if they are limited to using only deterministic transitions. Consequently we constructed the parameterized "comb" model as an example of a model with few equivalence classes, but many states. We have tested our algorithms on the parameterized Dining philosophers model, and on 2 parameterized comb models. The results of these tests are discussed below.

8 Comb model

Unlike the Dining philosophers model, the comb model is unknown. Consequently we describe it briefly here. As shown in Figure 8, the $M \times N$ comb Petri-net model has N rows of M places, with $N-1$ rows of M transitions between the rows of places. Each transition, when fired, receives a token from each place in the row above it, except the one place shown directly above it. Each transition then deposits one token into each place in the row below it.

The initial marking (see Figure 8, left) has one token in each place shown in the top row, and no other tokens. This class of models operates by firing one transition from each row until no more firings are possible, a state which occurs after $N-1$ firings. Only one transition from each row may fire, because the firing of one transitions takes away all but one token from the row of places above it. Consequently, all other transitions in that row become disabled. A typical terminal state, illustrated in Figure 8, right, shows each row of places having one token remaining in one of its places, preserving a record of which firings occurred. Obviously, these models will terminate after $N-1$ firings, and have $O(M^N)$ different states.

We use 2 variations of this model, parameterized in a single parameter N. In one variation ("$3 \times N$"), $M = 3$ always, and in the other variation ("$N \times N$"), $M = N$ always.

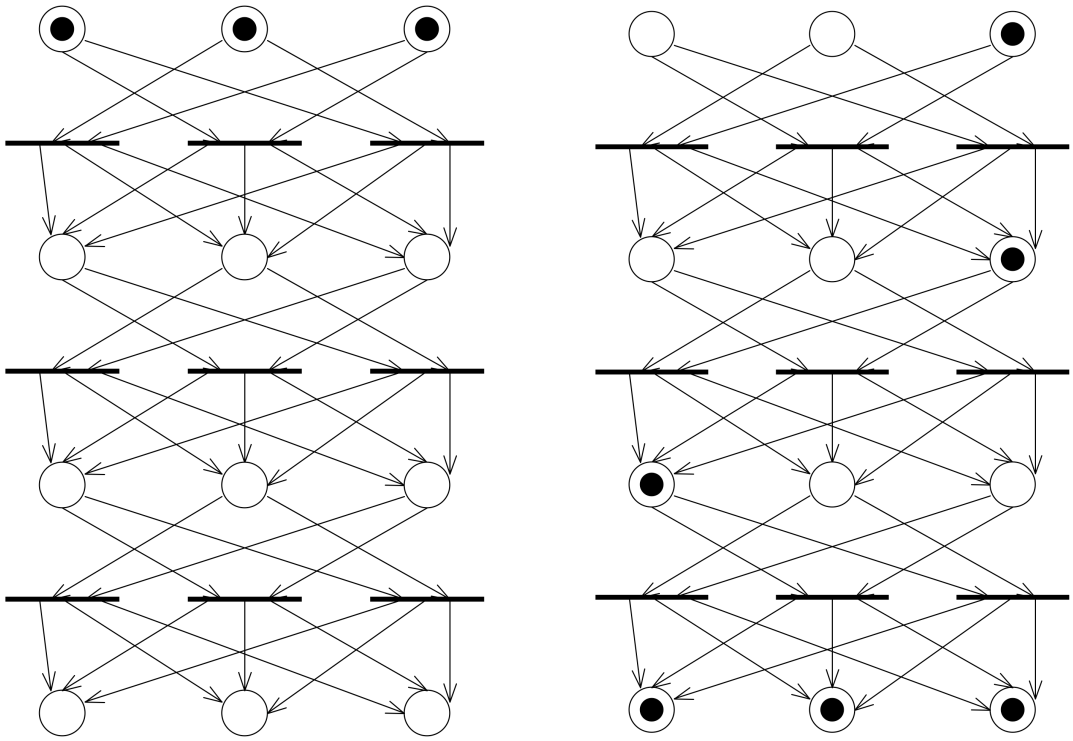


Figure 8: The comb model (left) with a final marking (right)

9 Results

We obtained the following results, summarized graphically, running our novel saturation-based algorithm, the classical fully implicit algorithm and the hybrid algorithm on the parametric Dining philosophers model, the $3 \times N$ comb model, and the $N \times N$ comb model. It should be noted that the Dining philosophers models have very many equivalence classes in their bisimulation partitions, in this case, only one less than the number of states. Those models therefore are expected to represent the ideal case for our algorithm.

9.1 Various Measures for Algorithm A

Figure 9 shows run-time, maximum memory usage, and output size, along with number of states, for our novel saturation-based algorithm applied to the Dining philosophers model.

It is clear that the run-time grows strongly sub-linearly in the number of states (and number of classes, in this case), as we had hoped. The maximum memory and output size also grow strongly sub-linearly in the number of states, for these models. All these cost measures increase very gradually with N, the number of philosophers.

9.2 Comparison of Algorithms

We also tested the other algorithms using the all of the same models, producing the comparisons below. It should be noted that, for Dining philosophers, the saturation-based Algorithm A was able to calculate the bisimulation for all N in $[2, 14]$, while the classic Algorithm 1 had prohibitive run-times when $N > 9$, and the hybrid algorithm H could not be run with $N > 7$. For the comb models, all algorithms were able to run for all $N \in [2, 20]$.

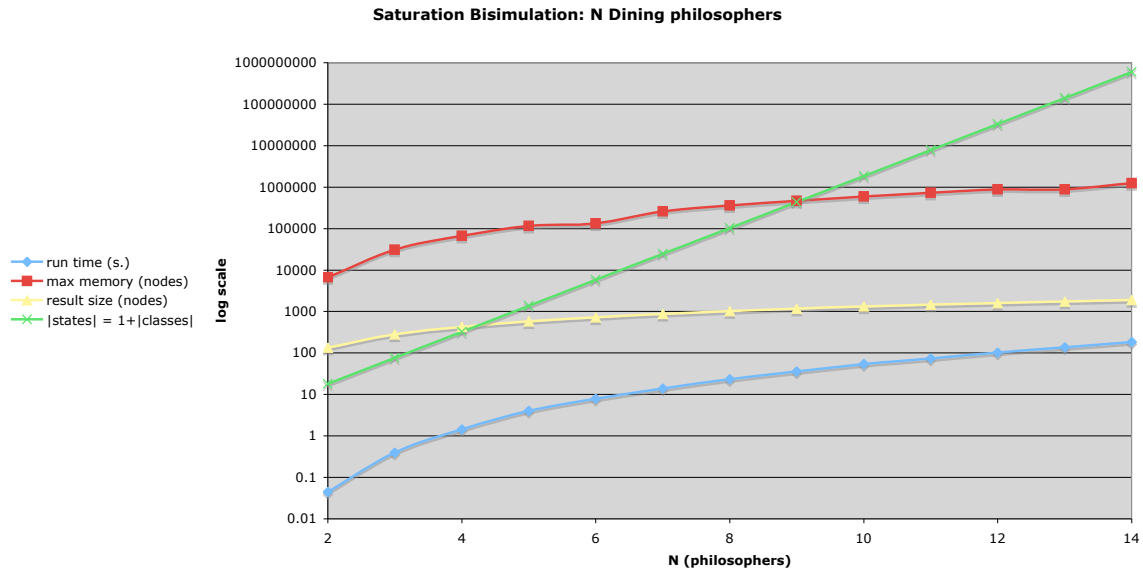


Figure 9: Performance measures for saturation-based bisimulation Algorithm A

9.2.1 Run-Time Comparison of Algorithms

Figure 10 shows run-time, for the 3 algorithms on the Dining philosophers models.

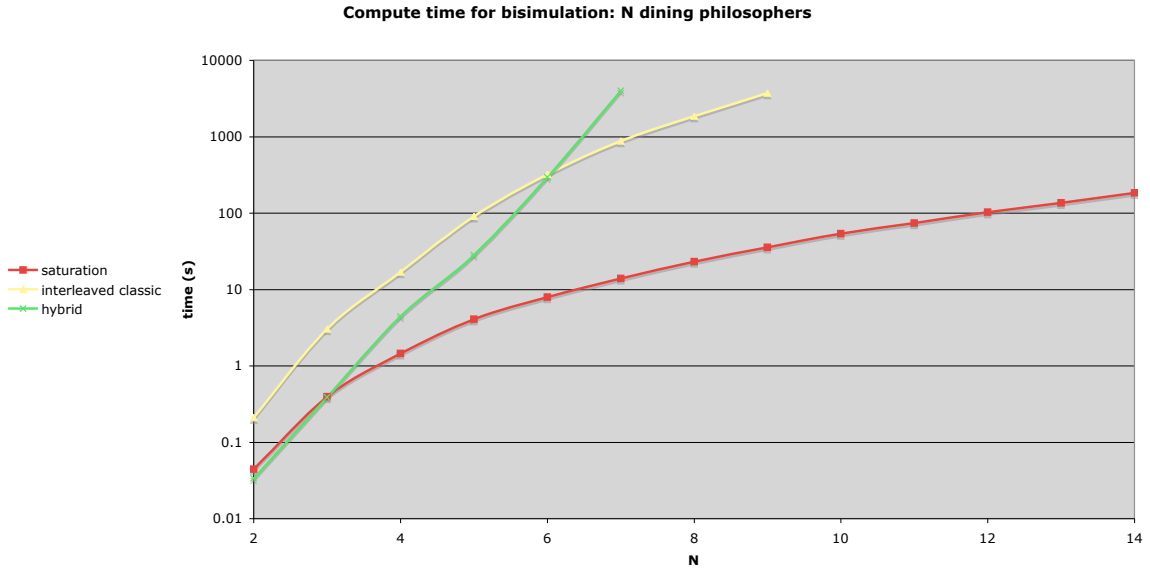


Figure 10: Run-times of 3 algorithms (Dining philosophers)

As we had hoped, on Dining philosophers, our novel algorithm out-performed both other algorithms. Figure 11 shows run-time, for the 3 algorithms on the $3 \times N$ comb models, and Figure 12 shows run-time, for the 3 algorithms on the $N \times N$ comb models. For reference, the number of states ($|S|$) and number of classes is also shown.

With the comb models, we find the novel algorithm is slightly slower than the hybrid algorithm, although its run-time remains reasonable, and does not exceed the run-time of the hybrid algorithm by an order of magnitude.

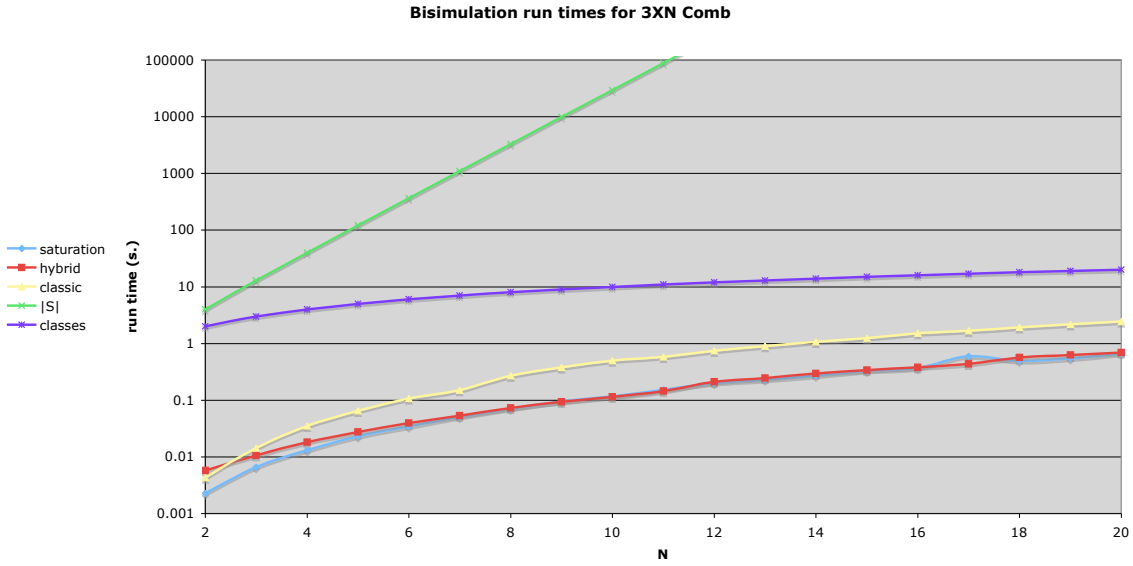


Figure 11: Run-times of 3 algorithms ($3 \times N$ comb)

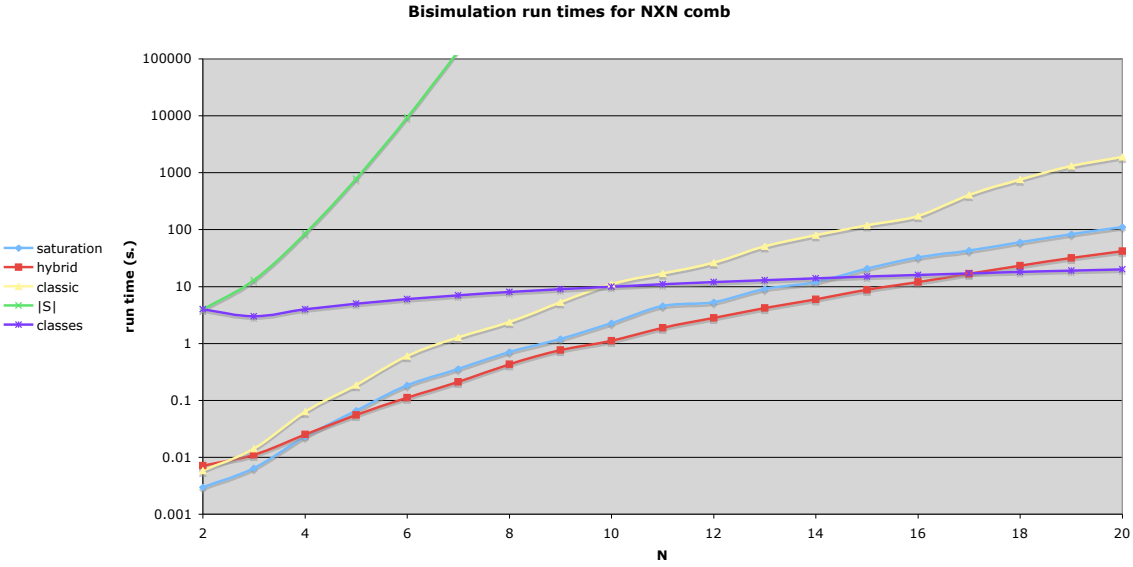


Figure 12: Run-times of 3 algorithms ($N \times N$ comb)

9.2.2 Maximum Memory Usage Comparison of Algorithms

Figure 13 shows maximum memory usage, in nodes, for the 3 algorithms on the Dining philosophers models.

As we had hoped, our novel saturation-based algorithm had much better memory usage than the hybrid algorithm, for the Dining philosophers model. Surprisingly, the classical Algorithm 1 had better memory usage. Figure 14 and Figure 15 show maximum memory usage, in nodes, for the 3 algorithms on the $3 \times N$ comb and $N \times N$ comb models respectively.

With these models, the classic algorithm and the hybrid algorithm tended to have very similar maximum memory usage, while our new algorithm usually used moderately more memory.

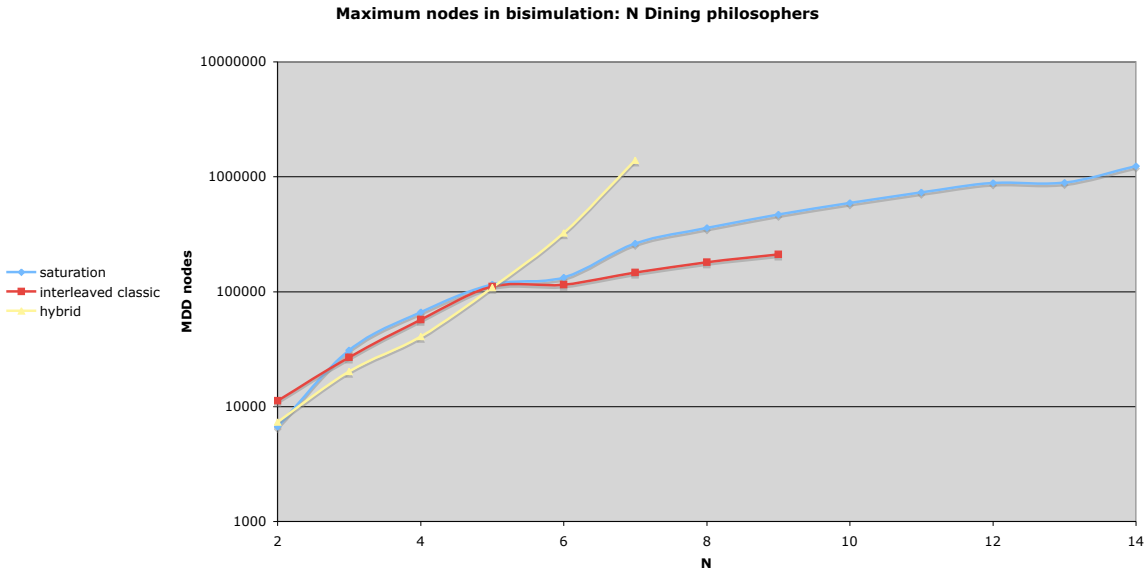


Figure 13: Maximum Memory Usage of 3 algorithms (Dining philosophers)

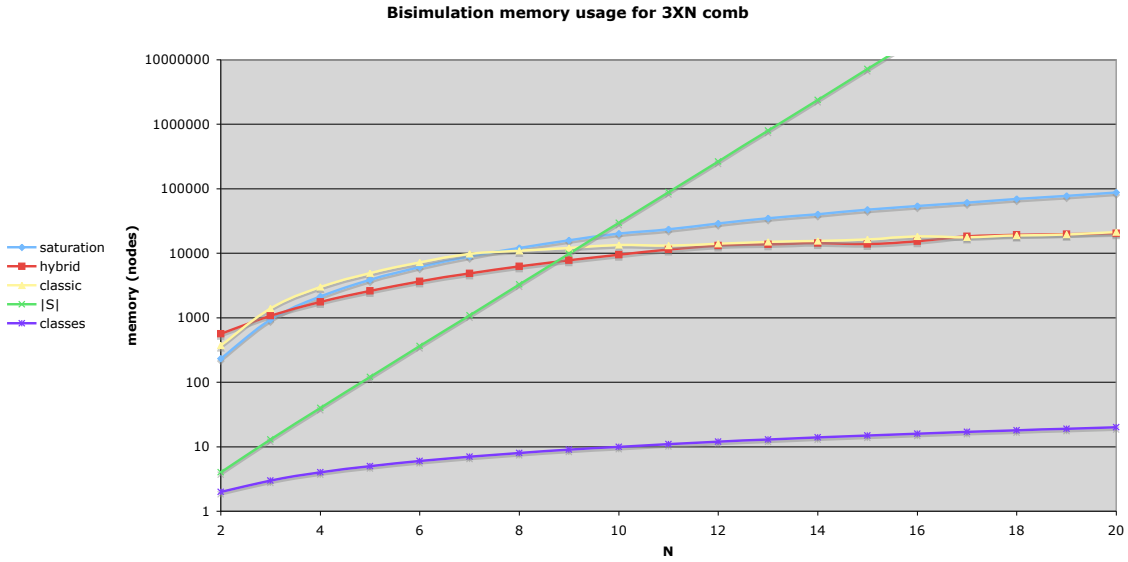


Figure 14: Maximum Memory Usage of 3 algorithms ($3 \times N$ comb)

9.2.3 Output Size Comparison of Algorithms

Figure 16 shows output size, in nodes, for the 3 algorithms on the Dining philosophers models.

As expected, the algorithms (classic, and saturation) with fully implicit representation produced much more compact output than the hybrid algorithm for Dining philosophers. The output size of the hybrid algorithm grows linearly in the number of classes. For the $N \times N$ comb model, there was a slight exception. The output size using the fully implicit representation was slightly larger for several values of N.

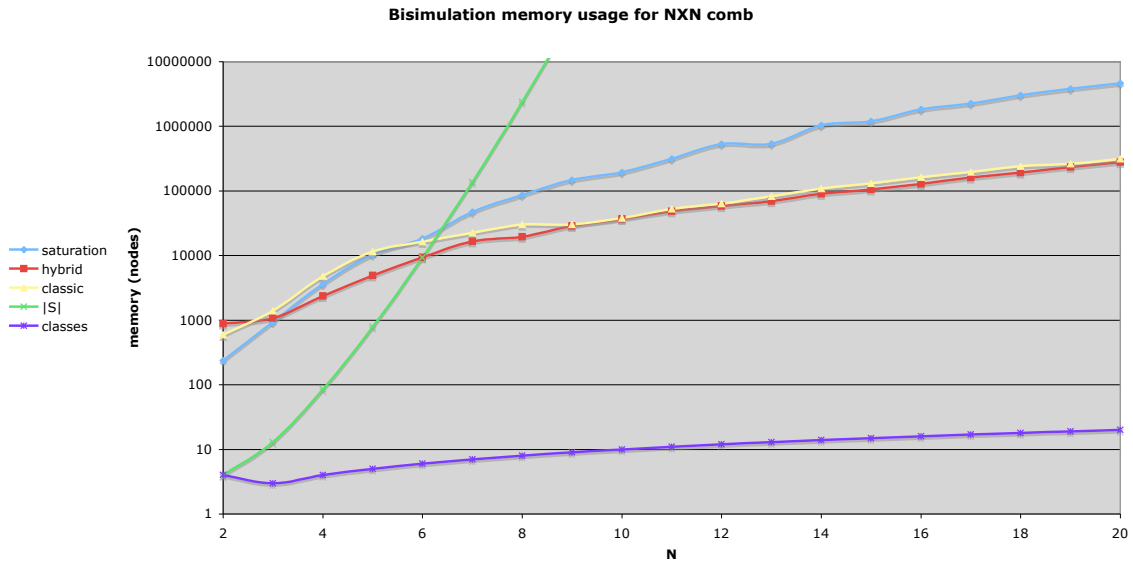


Figure 15: Maximum Memory Usage of 3 algorithms ($N \times N$ comb)

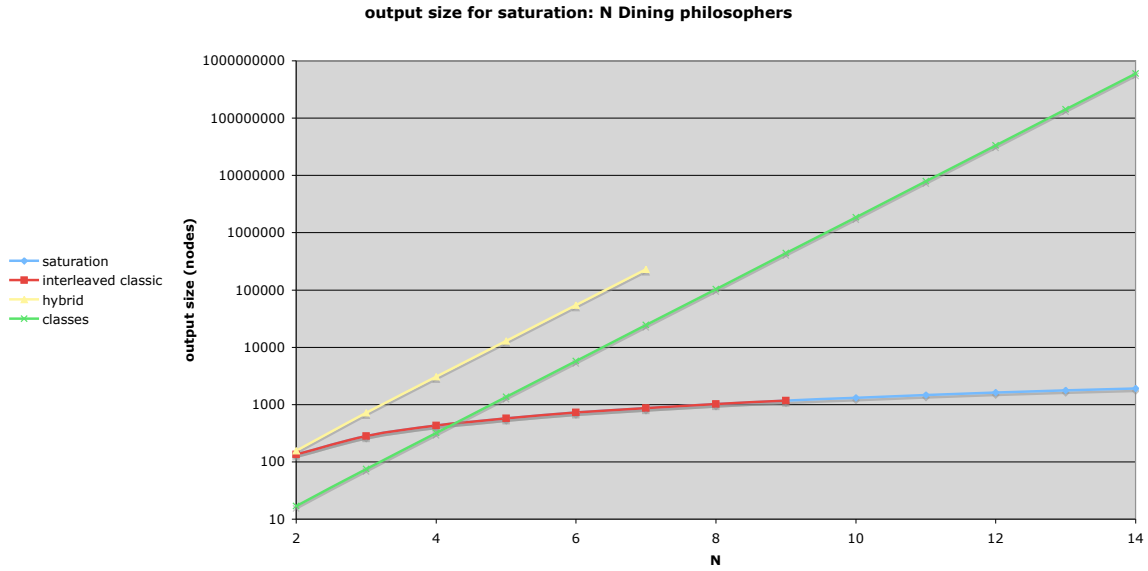


Figure 16: Output Size of 3 algorithms (Dining philosophers)

10 Conclusions

It appears that we have found a saturation-based symbolic bisimulation algorithm which is potentially capable of calculating bisimulations for some large systems beyond the reach of extant algorithms. When applied to the comb models, which have very small numbers of classes, this algorithm is not the fastest, but it still provides reasonable performance for those cases. This novel bisimulation algorithm usually provides the ability to deliver output in a compact (interleaved) representation in time strongly sub-linear in the number of states. This is what is required for further symbolic processing of bisimulation results by other algorithms.

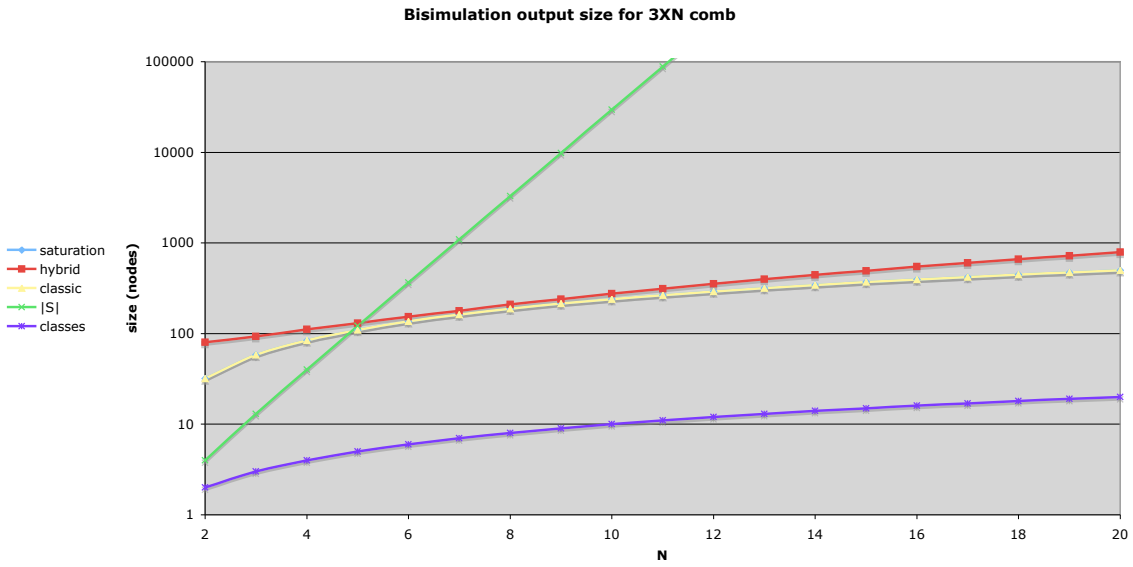


Figure 17: Output Size of 3 algorithms ($3 \times N$ comb)

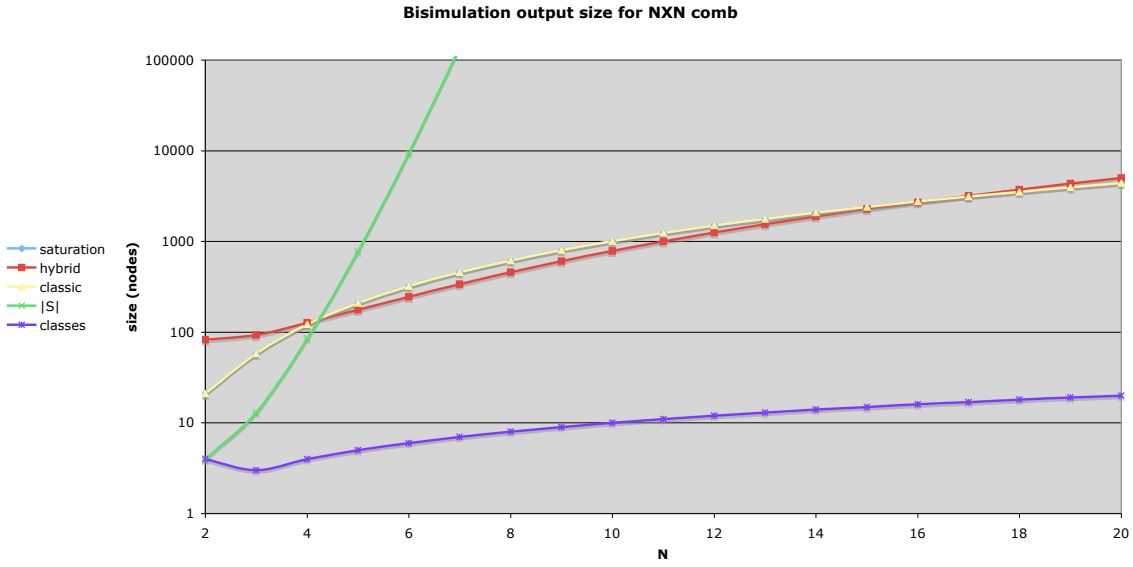


Figure 18: Output Size of 3 algorithms ($N \times N$ comb)

11 Future Work

A variation of our novel algorithm may be applied to systems with transitions of limited non-determinism. We hope this can be achieved by treating some non-deterministic transitions as unions of a small number of deterministic transitions, at the cost of additional set operations at the core of our algorithm.

We also hope to eventually extend the use of the interleaved representation of partitions to Markov Lumping (see Algorithm X in appendix X).

Future updates of this work may be found at:

`http://www.cs.ucr.edu/~mummem/{bisimulation,lumping,MSproject}` .

12 Appendices

12.1 Appendix X

This lumping algorithm is almost identical to bisimulation Algorithm 1, except that three operations are replaced, and an EV$^+$MDD or an AADD is used for the $\mathcal{Q}$ parameter.

The following additional definitions are required:

$DC_2(F,G)(x,a,y) \triangleq F(x,y)$ whenever G(a) inserts don't-care variable in position 2 (extended for EV$^+$MDD F)

$proj_{\Sigma 3}(F)(x,y) \triangleq \Sigma_a F(x,y,a)$ project third variable by summation (EV$^+$MDD F)

$(F \times G)(\ldots args \ldots) \triangleq \left\{ \begin{array}{ll} F(\ldots args \ldots) & \text{iff } G(\ldots args \ldots) \\ 0.0 & \text{iff } \neg G(\ldots args \ldots) \end{array} \right\}$ intersection, analogous to $\cap$

Algorithm X:

MDD *LRefine*($\mathbb{N}$ $\mathcal{N}_{transition\_labels}$, EV$^+$MDD $\mathcal{Q}_{[]}$, MDD $\mathcal{E}$, MDD $\mathcal{S}$) is
local MDD $\mathcal{T}_{[]}$, MDD $\overline{\Delta\mathcal{E}}$, MDD $\mathcal{E}'$,

1 $\overline{\Delta\mathcal{E}} \leftarrow \emptyset$
2 for $t \in [0, \mathcal{N}_{transition\_labels})$ loop
3 $\mathcal{T}_t \leftarrow proj_{\Sigma 3}((DC_2(\mathcal{Q}_t, \mathcal{S})) \times (DC_1(\mathcal{E}, \mathcal{S})))$ • $\mathcal{T}_t(a,b) = \Sigma c : (\mathcal{Q}(a,c) \times \mathcal{E}(b,c))$
4 $\overline{\Delta\mathcal{E}} \leftarrow \overline{\Delta\mathcal{E}} \cup proj_{\vee 3}(DC_2(\mathcal{T}_t, \mathcal{S}) \uplus DC_1(\mathcal{T}_t, \mathcal{S}))$ • $\Delta\mathcal{E}(a,b) = \bigwedge t \in [0,t] : \bigwedge c : (\mathcal{T}_t(a,c) = \mathcal{T}_t(b,c))$
5 end loop
6 $\mathcal{E}' = \mathcal{E} \setminus \overline{\Delta\mathcal{E}}$ • $\mathcal{E}' = \mathcal{E} \cap \Delta\mathcal{E}$
7 return $\mathcal{E}'$

MDD *LumpingRefineClosure*($\mathbb{N}$ $\mathcal{N}_{transition\_labels}$, EV$^+$MDD $\mathcal{Q}_{[]}$, MDD $\mathcal{S}$, $\mathbb{N}$ $\mathcal{N}_{state\_labels}$, MDD $\mathcal{SL}_{[]}$) is
local MDD $\mathcal{E}_0$, MDD $\mathcal{E}$, MDD $\mathcal{E}_{old}$,

1 $\mathcal{E}_0 \leftarrow \emptyset$ • *Construct initial partition* $\mathcal{E}_0$
2 for $l \in [0, \mathcal{N}_{state\_labels})$ loop
3 $\mathcal{E}_0 \leftarrow \mathcal{E}_0 \cup DC_2(\mathcal{SL}_l, \mathcal{SL}_l)$
4 end loop
5 $\mathcal{E} \leftarrow \mathcal{E}_0$ • $\mathcal{E}(a,b) = \exists l \in [0, \mathcal{N}_{state\_labels}) : (a \in \mathcal{SL}_l \wedge b \in \mathcal{SL}_l)$
6 repeat • $\mathcal{E} \leftarrow$ *LRefine*($\mathcal{E}$) *until* $\mathcal{E} =$ *LRefine*($\mathcal{E}$)
7 $\mathcal{E}_{old} \leftarrow \mathcal{E}$
8 $\mathcal{E} \leftarrow$ *LRefine*($\mathcal{N}_{transition\_labels}$, $\mathcal{Q}_{[]}$, $\mathcal{E}$, $\mathcal{S}$)
9 until $\mathcal{E} = \mathcal{E}_{old}$
10 return $\mathcal{E}$ • $E_{n_{max}}$

Algorithm X, Closure of partition refinement (Markov lumping)

13 Acknowledgements

I thank my advisor for sharing his ideas and research.

Thanks to Min Wan for the use of the fork-join model and help with learning the SMART2 MDD library interface.

I thank the Lord for bringing me to the right advisor at the right time.

References

[1] Amar Bouali and Robert De Simone. Symbolic bisimulation minimisation. In *In Computer Aided Verification*, pages 96–108. Springer-Verlag, 1992.

[2] Randy E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[3] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In Tiziana Margaria and Wang Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, April 2001. Springer-Verlag.

[4] Gianfranco Ciardo and Radu Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In Mark D. Aagaard and John W. O'Leary, editors, *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pages 256–273, Portland, OR, USA, November 2002. Springer-Verlag.

[5] S. Derisavi. A symbolic algorithm for optimal Markov chain lumping. In O. Grumberg and M. Huth, editors, *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 139–154. Springer, 2007.

[6] Salem Derisavi. Signature-based symbolic algorithm for optimal markov chain lumping. In *Proc. Quantitative Evaluation of Systems (QEST)*, pages 141–150, 2007.

[7] Salem Derisavi, Holger Hermanns, and William H. Sanders. Optimal state-space lumping in markov chains. *Inf. Process. Lett.*, 87(6):309–315, 2003.

[8] Salem Derisavi, Peter Kemper, and William H. Sanders. Lumping matrix diagram representations of markov models. In *Proc. Int. Conf. on Dependable Systems and Networks (DSN)*, pages 742–751, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[9] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. theoretical computer science. *Theor. Comput. Sci*, 311:1–3, 2004.

[10] M. Fujita, P. C. McGeer, , and J. C.-Y. Yang. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169, 1997.

[11] Holger Hermanns and Markus Siegle. Bisimulation algorithms for stochastic process algebras and their bdd-based implementation. In *ARTS '99: Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, pages 244–264, London, UK, 1999. Springer-Verlag.

[12] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[13] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th Conference on Design Automation*, pages 608–613, Los Alamitos, CA, USA, June 1992. IEEE Computer Society Press.

[14] Faron Moller and Scott A. Smolka. On the computational complexity of bisimulation, redux. In *Principles of Computing and Knowledge: Paris C. Kanellakis Memorial Workshop*, pages 55–59, San Diego, CA, USA, June 2003.

[15] Malcolm Mumme. Fully-Implicit Relational Coursest Partitioning for Faster Bisimulation. Master Project Report, University of California, Riverside, USA, December 2008.

[16] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 6(16), December 1987.

[17] Scott Sanner and David McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *Proc. IJCAI*, 2005.

[18] F. Somenzi. CUDD: CU Decision Diagram Package, Release 2.3.1. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.

[19] Paul Tafertshofer, Lehrstuhl Fur Rechnergestutztes Entwerfen, Prof Dr. ing, K. Antreich, Prof. Dr. Massoud Pedram, Betreuer Ulf Schlichtmann, and Bernd Wurth. Factored edge-valued binary decision diagrams. In *Formal Methods in System Design*, pages 109–132. Kluwer, 1997.

[20] Min Wan, Gianfranco Ciardo, and Andrew S. Miner. Decision-diagram-based approximate steady-state analysis of large structured markov models. [in progress].

[21] Ralf Wimmer, Marc Herbstritt, and Bernd Becker. Forwarding, splitting, and block ordering to optimize bdd-based bisimulation computation.