Generating BDD models for process algebra terms

Ashvin Dsouza and Bard Bloom*

Department of Computer Science,
Cornell University,
Ithaca, NY 14853, USA

Abstract. The Simple systems form a class of process algebras whose operational semantics can be specified using finite state labelled transition systems. In this work, we describe how to efficiently derive the ordered Binary Decision Diagrams (BDDs) corresponding to the operational semantics of the terms of an arbitrary Simple system. Model checking using such BDDs can often significantly speedup the testing of properties such as bisimilarity over direct algorithms. We also introduce a useful extension of Simple providing explicit recursion. For the CCS operators, we show that the corresponding BDD operators we generate automatically are comparable to those coded by hand.

1 Introduction

Process algebras are coming into increasing use as specification tools for concurrent systems. Specifications are given as terms; the appropriate use of the algebra's operations lends useful structure to the specifications. As process algebraic specifications resemble programs, they appeal to programmers' intuitions.

This study is part of an ongoing project to develop tools for the support of process algebraic methods for specification and verification of programs. There is a variety of such tools available for specific process algebras. However, many of these tools are customized for particular process algebras; *e.g.*, the tools of LOTOSphere [LOT] are intended for use with LOTOS.

Building custom tools is certainly appropriate. However, in many cases, it is possible to build general tools without loss of efficiency and with a distinct gain in expressive power. In this study, we give a procedure for model-checking processes over a wide class of process algebras. Preliminary results indicate that our method gives comparable results to custom model-checkers.

The problem of model-checking in general is, given a finite-state program p and a formula φ describing a desired property of p, to determine automatically whether p has property φ, by a complete consideration of all the possible states of p. For a program of any significant size, it is prohibitively expensive to generate all the states of p directly.

* bard@cs.cornell.edu. Supported by NSF grants CCR-9003441, CCR-9003440, CCR-9014363, and CCR-9223183.

So, modern model checking methods are largely based on compact, efficient representations of state spaces. The most successful method[BCM+90] is to take advantage of any innate structure in a state space to obtain a compact representation of it using an ordered Binary Decision Diagram (BDD)[Bry86]. In [BCM+90], a model checker is obtained by encoding temporal logic specifications as μ-calculus formulas, which are then interpreted as BDD's and evaluated. Systems with as many as 10^{120} states have been efficiently verified this way[BCL91], and larger systems have been verified with appropriate abstraction functions.

For specificity, we consider the problem of deciding *bisimulation*: [Par81, Mil89] that is, of deciding whether two processes are identical in a very strong sense. A μ-calculus formula encoding the bisimulation relation between two processes is also presented in [BCM+90].

However, that formulation assumed that the processes were already represented as BDD's. No methods were provided in [BCM+90] for the construction of BDD's from programs. This problem is addressed in [EFT93] for CCS, by providing operators on BDD's representing transition systems to mimic the effect of certain CCS operators: parallel composition, channel-hiding, and renaming. The simple scheduler described in [Mil89] was verified by BDD methods, and a significant improvement in performance over direct methods [dSV89, Fer89, GV90] was obtained. That is, BDD techniques indeed help model-checking for CCS.

Other researchers have used BDD techniques on CCS and related process algebras. For finite automata running in parallel, [BdS92] uses BDD's to improve the conventional fast bisimulation algorithm based on partition refinement. Partitioning for a variety of equivalences is carried out efficiently using BDD's in [FKM93], also using automata running in parallel. In [MM93], BDD's are used to speed up bisimilarity checking for the Circal calculus.

However, these techniques either focus on one particular process algebra and derive by hand a BDD operator for each operator of the process algebra, or assume the setting to be finite automata running in parallel.

In this paper, we present a general method to generate BDD models for the terms of a useful class of process algebras. Our class of calculi includes most finitary process-algebraic operations that have appeared in practice, including large fragments of CCS, CSP, ACP, Meije, and LOTOS and many less-well-publicized ones, and thus our techniques should be widely useful.

1.1 Structural Operational Semantics

The basic semantic model of processes we use is the familiar *labelled transition system* (LTS), in which a process state p is an entity capable of taking actions a and thereafter acting like some other process state q. Such a transition is written $p \xrightarrow{a} q$.

Most process algebras are given behavior by means of a *Structural Operational Semantics* (SOS), in which the behavior of a composite process $f(p, q)$ is determined by rules which can examine the behavior of p and q. For example, the CCS nondeterministic choice of p and q, written $p+q$, can perform an action of p (or one of q). This is given by the rules:

$$\frac{p \xrightarrow{a} r}{p+q \xrightarrow{a} r} \qquad \frac{q \xrightarrow{a} r}{p+q \xrightarrow{a} r}$$

where a ranges over a set of actions. Most processes algebras with LTS semantics use terms for states, and give the transition relation inductively by SOS rules. Since SOS rules are fairly regular in form, it has proven worthwhile to study the *metatheory* of SOS: that is, the properties of process algebras defined by specific formats of SOS rule systems [dS85, Blo90, ABV94]. For example, any language in GSOS format [Blo90] has bisimulation as a congruence; and one may automatically extract a set of equational axioms from the rules which (together with an induction principle) are complete for proving equations between programs.

We selected the class of Simple systems [Ace93], which is a subclass of GSOS guaranteeing finite state behavior for terms. We extended it with recursion, subject to restrictions that keep recursive processes finite-state. Indeed, most process algebras (including CCS, CSP, Meije, core LOTOS, and ACP) are Simple languages with an infinite-state recursion operation added. Obviously, ordinary BDD-based model checking only works on finite state processes, so this seems a reasonable class to consider.

This paper is structured as follows: In the next section, we describe Simple systems and BDD's. In Section 3, we show how to derive the BDD representation of the transition system semantics of a Simple term. In Section 4, we propose an extension of Simple systems with explicit recursion and describe how it can be handled by our BDD construction. We then describe our implementation in Section 5, and end with a discussion of future work in Section 6. A formal construction, with proofs of correctness and a bound on the BDD size, can be found in the full version of the paper[DB95].

2 Preliminaries

2.1 Simple Systems

Simple systems[2] [Ace93] are GSOS systems [Blo90] that guarantee finite state operational semantics for process terms. The main constraint added to GSOS is that no nesting of operators is allowed on the right hand side of the consequent of a rule.

Definition 1. Let Σ be a finite signature, and Act a finite set of actions. A Simple rule ρ is of the form:

$$\frac{\bigcup_{i=1}^{l} \left\{ x_i \xrightarrow{a_{i,j}} y_{i,j} \mid 1 \le j \le m_i \right\} \cup \bigcup_{i=1}^{l} \left\{ x_i \overset{b_{i,j}}{\not\rightarrow} \mid 1 \le j \le n_i \right\}}{g(\vec{x}) \xrightarrow{a} t} \qquad (1)$$

[2] We refer to the *simple GSOS* rules and systems introduced in [Ace93] as Simple rules and systems in this paper.

where all the variables $\vec{x}$ and $\vec{y}$ are distinct, $0 \leq m_i, 0 \leq n_i$, g is an operation symbol from Σ of arity l, and the $a_{i,j}, b_{i,j}$ and a are actions from Act. The term t is either a variable in $\vec{x}, \vec{y}$, or of the form $h(\vec{z})$ where h is an operation symbol from Σ and each z_i is a variable in $\vec{x}$ or $\vec{y}$.

Definition 2. A Simple system over a finite set of actions Act is a pair $G = (\Sigma_G, \mathcal{R}_G)$, where Σ_G is a finite signature and $\mathcal{R}_G$ is a finite set of Simple rules over Σ_G.

Note that Simple rules allow arbitrary copying of arguments, as well as negative antecedents. It follows from GSOS theory [Blo90] that any Simple system G has a well defined operational semantics, which maps each G-term to a transition system over Act. Denoting the labelled transition system of a term t by $\mathsf{TS}(t)$, the following theorem stating the finiteness property of Simple systems is proved in [Ace93]:

Theorem 3. *Let $G = \langle \Sigma_G, \mathcal{R}_G \rangle$ be a* Simple *system. Then, for all $t \in \mathsf{T}(\Sigma_G)$, where $\mathsf{T}(\Sigma_G)$ is the set of terms over Σ_G, $\mathsf{TS}(t)$ is finite.*

Example 1. The checkpointing operator p was q is intended to capture the abstract behavior of the checkpointed version of the running process p, with a prior state q backed up on stable storage. The rules for was are :

$$\frac{x_1 \xrightarrow{a} x_1'}{x_1 \ \mathsf{was}\ x_2 \xrightarrow{a} x_1' \ \mathsf{was}\ x_2} \qquad \frac{x_1 \xrightarrow{\mathsf{f}} x_1'}{x_1 \ \mathsf{was}\ x_2 \xrightarrow{\mathsf{r}} x_2 \ \mathsf{was}\ x_2} \qquad \frac{}{x_1 \ \mathsf{was}\ x_2 \xrightarrow{\mathsf{c}} x_1 \ \mathsf{was}\ x_1} \tag{2}$$

where a ranges over a set of actions A not including f, r or c. The first rule[3] says that the running process p can compute freely on the actions in A, representing proper computation; p's ordinary computation does not modify the stored process q. The second rule says that, if p fails — signalled by the action f - then a new running process is started from the saved state q. The third rule says that, at any time, the system may do a checkpoint, copying p onto stable storage. The actions r and c signal that a restart or checkpoint happened. All three rules are Simple.

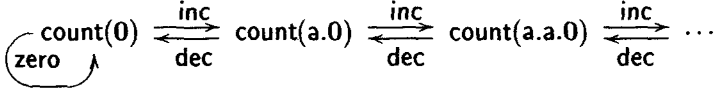
Example 2. Even a slight relaxation of the Simple constraints can violate the finiteness property. Consider the GSOS system $\langle \Sigma, \mathcal{R} \rangle$ over Act$=\{$zero, inc, dec, a$\}$ where $\Sigma = \{\mathsf{count}, \mathsf{a}., 0\}$ and $\mathcal{R}$ is as follows:

$$\frac{}{\mathsf{a}.x \xrightarrow{\mathsf{a}} x} \qquad \frac{x \stackrel{\mathsf{a}}{\not\rightarrow}}{\mathsf{count}(x) \xrightarrow{\mathsf{zero}} \mathsf{count}(x)} \qquad \frac{x \xrightarrow{\mathsf{a}} y}{\mathsf{count}(x) \xrightarrow{\mathsf{dec}} \mathsf{count}(y)} \tag{3}$$

$$\frac{}{\mathsf{count}(x) \xrightarrow{\mathsf{inc}} \mathsf{count}(\mathsf{a}.x)} \tag{4}$$

[3] This is really a rule schema parameterized by A.

The operator 0 has no rules, and thus is a stopped process. The rules (3) are valid Simple rules, but (4) has one level of nesting in its target. A graphical representation of TS(count(0)) is as follows:



This is an infinite state transition system, an unbounded counter, violating the finiteness property.

2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDD's)[Bry86] are graphs representing Boolean functions, in which the variables of the function are ordered. For a given variable ordering, the reduced BDD representation of a Boolean function is canonical, making satisfiability testing and equivalence testing trivial. Also, many common functions have compact BDD representations, which we exploit in implementing transition systems.

A BDD has a distinguished root node, internal nodes containing Boolean variables, and terminal nodes containing either the value 1 or the value 0. Nodes containing variables later in the order are reachable from those with variables earlier in the order but not vice versa. Non-terminal nodes have exactly two outgoing edges, labelled 0 and 1.

To evaluate the Boolean function for a given variable assignment using its BDD representation, start at the root, and take the edge labelled with 1 or 0, depending on the value of the variable at the current node. The value of the function is the value contained in the terminal node that is reached using this process.

In diagrams for BDD's, we mostly omit edge labels, using the convention that edges going left are labelled 0, while edges going right are labelled 1. Each internal node is labelled with its variable. The terminal node with value 0 and its associated edges are omitted.

The size of the BDD for given Boolean function is very sensitive to the ordering of the variables. While finding the optimal variable ordering is co-NP complete, many real world problems admit a heuristic ordering that performs well.

Example 3. Consider the Boolean function φ in disjunctive normal form (CNF)

$$\varphi = \begin{pmatrix} (v_1 \wedge v_2 \wedge v_3) \vee (v_1 \wedge v_5 \wedge v_3) \vee (v_1 \wedge v_2 \wedge v_6) \vee (v_1 \wedge v_5 \wedge v_6) \vee \\ (v_4 \wedge v_2 \wedge v_3) \vee (v_4 \wedge v_5 \wedge v_3) \vee (v_4 \wedge v_2 \wedge v_6) \vee (v_4 \wedge v_5 \wedge v_6) \end{pmatrix}$$

The BDD representation of φ using the variable orderings $v_1 < v_4 < v_2 < v_5 < v_3 < v_6$ and $v_1 < v_2 < v_3 < v_4 < v_5 < v_6$ respectively are shown in Figure 1. The smaller BDD is about as large as the conjunctive normal form of φ:

$$\varphi = (v_1 \vee v_4) \wedge (v_2 \vee v_5) \wedge (v_3 \vee v_6)$$

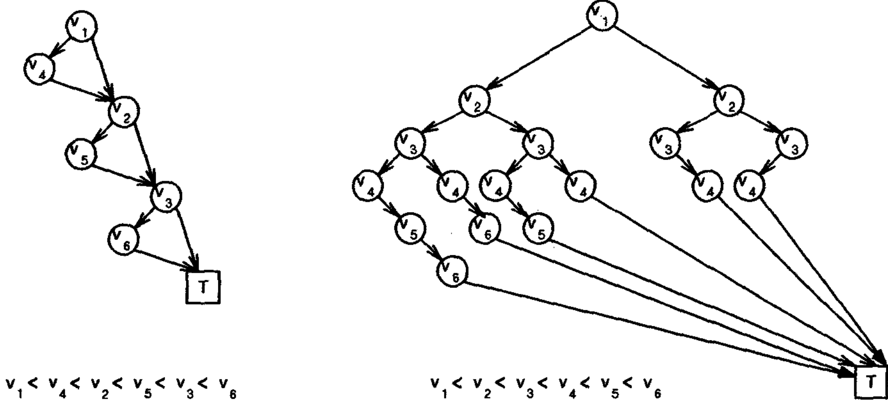


Fig. 1. BDDs for φ using good and bad variable orderings

To represent a LTS as a BDD, we must first encode the states and labels using bit variables, and then construct a BDD in these variables that will recognize encodings of valid transitions.

3 BDD Construction

The standard construction of the transition relation $p \xrightarrow{a} q$ is mathematically elegant, but does not immediately lead to a BDD construction that avoids explicitly enumerating the transitions. The difficulty is that the relation between the structure of p and q may be rather arbitrary. For example, consider $p = f(a.b) + c \xrightarrow{d} g(b) = q$ via the transition $f(a.b) \xrightarrow{d} g(b)$. The structure of q bears little resemblance to that of p. In this section, we represent terms and transitions in a way that preserves more of the structure, and is thus more suitable for BDD's.

3.1 Rules Without Copying Or Elimination

We illustrate the concepts with the Simple system G_0, which is designed so that the structure of terms stays visible and constant. G_0 is very stylized, in ways that illustrate our methods well; but our methods require nothing but the Simple rule format.

G_0 has j-ary identity functions $\mathsf{id}_{i/j}$, with rules such that $\mathsf{id}_{i/j}(x_1, \ldots, x_j)$ behaves like x_i. G_0 also has slightly unusual variants of prefixing $a()$ and choice $+$. In CCS, as $a(x)$ evolves, it forgets the fact that it used to have the shape $a(x)$: the rule is $a(x) \xrightarrow{a} x$.[4] G_0's prefixing keeps some record that of where it

[4] For most purposes, the CCS-style evolution is simpler and superior. However, G_0's method is ideally suited for this example.

started: its rule is the computationally equivalent one,

$$a(x) \xrightarrow{a} \mathsf{id}_{1/1}(x) \tag{5}$$

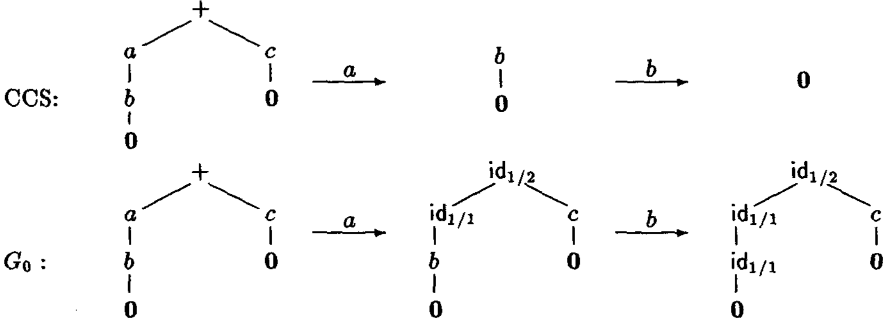
Similarly, the binary form $x + y$ evolves into $\mathsf{id}_{1/2}(x', y)$ or $\mathsf{id}_{2/2}(x, y')$.

$$\frac{x_1 \xrightarrow{a} y_1}{x_1 + x_2 \xrightarrow{a} \mathsf{id}_{1/2}(y_1, x_2)} \qquad \frac{x_2 \xrightarrow{a} y_2}{x_1 + x_2 \xrightarrow{a} \mathsf{id}_{2/2}(x_1, y_2)} \tag{6}$$

$$\frac{x_1 \xrightarrow{a} y_1}{\mathsf{id}_{1/2}(x_1, x_2) \xrightarrow{a} \mathsf{id}_{1/2}(y_1, x_2)} \qquad \frac{x_2 \xrightarrow{a} y_2}{\mathsf{id}_{2/2}(x_1, x_2) \xrightarrow{a} \mathsf{id}_{2/2}(x_1, y_2)} \tag{7}$$

where a ranges over all actions.

In G_0, the parse tree of a term never changes shape, though the operators at the nodes change. For example, the G_0 version of the CCS computation $a.b.\mathbf{0} + c.\mathbf{0} \xrightarrow{a} b.\mathbf{0} \xrightarrow{b} \mathbf{0}$ is $a.b.\mathbf{0} + c.\mathbf{0} \xrightarrow{a} \mathsf{id}_{1/2}(\mathsf{id}_{1/1}(b.\mathbf{0}), c.\mathbf{0}) \xrightarrow{b} \mathsf{id}_{1/2}(\mathsf{id}_{1/1}(\mathsf{id}_{1/1}(\mathbf{0})), c.\mathbf{0})$. We can compare these evaluations using parse trees:



Since the G_0 parse tree never changes shape, we can code the term and its descendants easily, by simply telling which operations are at which nodes of the tree. In this case, the head operator is either $+$, $\mathsf{id}_{1/2}$, or $\mathsf{id}_{2/2}$; let us code these as the bit patterns `11`, `10`, and `01`. Similarly, the first argument starts with either a-prefixing or $\mathsf{id}_{1/1}$; we code these as `1` and `0` respectively. The nullary operator $\mathbf{0}$ never evolves, so it doesn't need a code. $a.b.\mathbf{0} + c.\mathbf{0}$ is coded as `11111`:

11 $+$ $s_1 s_2$	1 a s_3	1 b s_4	$\mathbf{0}$	1 c s_5	$\mathbf{0}$
oper.	left subterm			right subterm	

(8)

The transitions above are represented:

11 $+$	1 a	1 b	$\mathbf{0}$	1 c	$\mathbf{0}$

$\xrightarrow{a}$

10 $\mathsf{id}_{1/2}$	0 $\mathsf{id}_{1/1}$	1 b	$\mathbf{0}$	1 c	$\mathbf{0}$

$\xrightarrow{b}$

10 $\mathsf{id}_{1/2}$	0 $\mathsf{id}_{1/1}$	0 $\mathsf{id}_{1/1}$	$\mathbf{0}$	1 c	$\mathbf{0}$

(9)

For the BDD construction, we must encode the transition relation $p \xrightarrow{a} q$. We do this in the obvious way. We have the five-bit coding of p and q given in the diagrams above, so we will have five bit variables $s_1, \ldots, s_5$ for p, and a distinct

set $s'_1, \ldots, s'_5$ for q. We will have a third set of variables for the action: two bits, b_1, b_2, suffice for the action set $\{a, b, c\}$. Then the transition relation $p \xrightarrow{a} q$ can be expressed as a logical formula in terms of $\vec{s}$, $\vec{s}\,'$, and $\vec{b}$; as this is a finite logical formula, it induces a BDD in the standard way.

Suppose s represents some descendant q of p. The transition formula must first check the leading operator of q, as coded by $s_1 s_2$. If the leading operator is $\mathsf{id}_{1/2}$, then q takes a transition iff the left subterm (given by $s_3 s_4$) can take the same transition. So, suppose that φ_l is a formula over $s_3 s_4$, $s'_3 s'_4$, and $b_1 b_2$ which gives the transition system for the left subterm. We use the formula φ_l to make the transition in the subterm, and a clause about $s'_1 s'_2$ to explain that the main operator $\mathsf{id}_{1/2}$ does not change state. Then the $\mathsf{id}_{1/2}$ case is covered by:

$$s_1 s_2 = 10 \Rightarrow (\varphi_l \wedge s'_1 s'_2 = 10)$$

The entire transition relation is:

$$\begin{array}{lll} & s_1 s_2 = 10 \Rightarrow (\varphi_l \wedge s'_1 s'_2 = 10) & /^*\ \mathsf{id}_{1/2}\ ^*/ \\ \wedge & s_1 s_2 = 01 \Rightarrow (\varphi_r \wedge s'_1 s'_2 = 01) & /^*\ \mathsf{id}_{2/2}\ ^*/ \\ \wedge & s_1 s_2 = 11 \Rightarrow \left(\begin{array}{l} (\varphi_l \wedge s'_1 s'_2 = 10) \\ \vee\ (\varphi_r \wedge s'_1 s'_2 = 01) \end{array} \right) & /^*\ +\ ^*/ \\ \wedge & s_1 s_2 \neq 00 & \text{unused bit pattern} \end{array} \tag{10}$$

where φ_r is the formula for the transition system of the right subterm, using variables s_5, s'_5 and $b_1 b_2$.

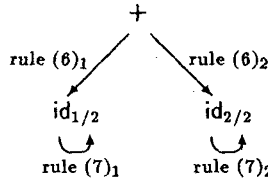
The construction in general We can construct such an encoding of states and transition formulas in general for G_0 terms. For concreteness, we consider terms of the form $p = f(t_1, t_2)$. All descendants of p can be represented as triples

$$\langle \underline{g}, s_1, s_2 \rangle \tag{11}$$

where $\underline{g}$ is an operator which f may evolve into, and s_1 and s_2 are representations of descendants of t_1 and t_2. We apply this scheme recursively, representing s_i by its operation symbols in nested tuples. As above, the operation symbols may change as the term evolves; but the nesting structure does not.

So, if we can discover the possible descendants of each operator f, then we have a good representation for each G_0 term. We could simply take the set of operators of the right arity — *e.g.*, $+$ can only evolve into binary operators — but, for a less trivial language than G_0, most of these will not actually be possible descendants.

We summarize the descendants for each operator by computing a reachability graph, which will serve in this case as the control structure we call the Operator Occurrence Graph (OOG). The OOG is more complicated when we handle general Simple systems, but for now it just has operators for nodes, and edges labelled by rules linking the corresponding operators. For example, the OOG for the G_0-operator $+$ is as follows:



Thus, no matter how many binary operations G_0 had, we would only need two bits to represent the three possible states corresponding to a + operation. The OOG mechanism is excessive for G_0, but is helpful in general.

3.2 Handling Copying And Elimination

The rules for G_0 were very stylized, in a way that made representing terms convenient. General Simple languages are not so stylized, but the same intuitions apply with suitable changes.

For example, Simple rules may have conclusions of the form $x + y \xrightarrow{a} x'$. We may transform the language slightly, adding an identity operator id to the language, and using it in all such conclusions: $x + y \xrightarrow{a} \mathsf{id}(x')$. We henceforth assume this transformation.

G_0 is also unrealistically simple in that it has neither copying, permutation, nor elimination: that is, each descendant of $f(x, y)$ has the form $g(x', y')$ where x' and y' are descendants or copies of x and y respectively. We must handle more complex operations, like $\cdot$ was $\cdot$ of (2), which have a more complex pattern of propagation of arguments.

Since Simple rules do not allow nesting of operators in the targets of rules, *i.e.* they have consequents of the form $f(\vec{x}) \xrightarrow{a} g(\vec{z})$, we examine how the rules relay their arguments to handle copying and elimination. For example, consider checkpointing: $p = (q \text{ was } r)$. The operator was is static, like $|$ and restriction and relabelling in CCS; that is, the descendants of q was r all have leading was operators. So, we don't need to waste bits saying that a descendant of p starts with was. However, we can't simply represent a state of p with a state of q and one of r, because we have transitions that copy one and eliminate the other, such as

$$p = (q \text{ was } r) \xrightarrow{\mathsf{c}} (q \text{ was } q) = p'. \tag{12}$$

Clearly, p''s representation should include two states of q and none of r. Conversely, after the f transition of the third rule for was, the state should include two states of r and none of q.

So, the representation of p should have three fields:

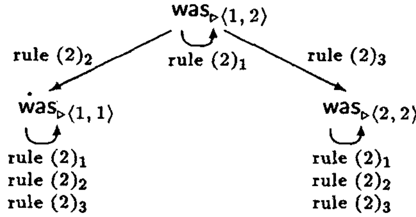
ww	$\vec{b}_1$	$\vec{b}_2$

where ww is two bits long, and $\vec{b}_1$ and $\vec{b}_2$ are each long enough to hold either a state of q or a state of r. We use, say, $ww = 00$ for terms of the form q was r, where $\vec{b}_1$ is a state of q and $\vec{b}_2$ a state of r; $ww = 10$ when both are states of q, and $ww = 01$ when both are states of r.

In general, then, we must record for each rule how the indices of the target variables are related to the indices of the source variables : this is the *index mapping* of the rule. The index mapping of a rule with conclusion $f(\vec{x}) \to g(\vec{z})$ is a vector $\langle i_1, i_2 \ldots i_n \rangle$whose first element tells which source variable x_{i_1} the target variable z_1 is associated with, and so on. Index mappings can be composed, which enables us to trace arguments along execution paths.

For example, the first checkpointing rule of (2) does not modify the structure of its arguments, and thus has the identity $\langle 1, 2 \rangle$ as its index mapping. However, the second rule has conclusion x_1 was $x_2 \to x_2$ was x_2, replacing both arguments by members of x_2's state space; this is represented by index map $\langle 2, 2 \rangle$. Similarly, the third rule has index mapping $\langle 1, 1 \rangle$.

We calculate the index mappings in the OOG, along with the operators. OOG nodes are now pairs $g_{\triangleright I}$, where g is an operator and I is an index mapping. To construct an OOG, we start with the initial head operator and the identity index map. From node $g_{\triangleright I}$, there is an edge for each each rule ρ for g to a node $h_{\triangleright J}$ where h is the target operator of ρ and J is the composition of the index mapping for ρ and I. For example, the checkpointing operator $\cdot$ was $\cdot$ described in Section 2.1 has the following OOG:



In general, the term $f(p_1, \ldots, p_n)$ will evolve into another term $g(q_1, \ldots, q_l)$; the q_j's are descendents of the p_i's, with the relation between i and j given by some index mapping I. Accordingly, $g_{\triangleright I}$ appears in the OOG for f.

So, when we represent $g(q_1, \ldots, q_l)$ we need a bit-field of sufficient size to identify a node of the OOG — that is, $\lg |V_{OOG}|$ bits, where V_{OOG} is the OOG's vertex set. The OOG node determines g and an index mapping I. We also need k bit-fields to represent the arguments of g, where k is the maximum arity of any operator in the OOG. We have each argument bit-field long enough to represent a state of any p_i[5]; the index map I tells which p_i each of the bit-fields should be interpreted as a state of.

In practice, the size of the OOG is often small: for example, the largest OOG for CCS is for summation and has just 3 nodes. However, in the worst case, a Simple system with n operators and maximum arity m may have as many as $n \cdot m^m$ nodes.

The transition relation is coded in just the manner suggested by (10). By induction, we have formulas φ_i which describe the transition relation for the

[5] A more refined analysis could save a few bits here, as it may not be the case that every process can get to every argument position.

i'th subterm p_i. We code the transition relation for $p = f(\vec{p})$ by, first, a case analysis on the bits representing the main operator g and index mapping I. Within each case, we have one clause for each rule for g, using the formulas φ_{I_j} to find transitions of the j'th subterm.

All in all, we have one clause for each rule that could ever be used to calculate the behavior of a descendant of p — which, obviously, is just the information we need. This logical formula may be calculated as a BDD in canonical form in the usual way, giving a BDD representation of the transition system.

Theorem 4. *The construction sketched in this section yields a transition system isomorphic to the ordinary SOS-calculated transition system.*

3.3 Optimizations

The construction we have outlined is correct, but sometimes uses too many state bits. There are a number of common special-case optimizations which apply, in many cases dramatically reducing the size of the BDD's. Some of these have been pointed out in the construction, but the most important optimization deals with BDD's for nested terms.

The problem with generating the BDD for a CCS term such as a.b.c.0, is that the construction will first generate the BDD for 0, then pass this as an argument to c.x which adds a bit of state, and passes the result as an argument to b.x, which adds another bit of state and so on. The result has three bits of state encoding only three states. For larger terms, this problem gets worse. Our solution is to extend the OOG construction to ruloids[Blo90]; ruloids are similar to rules, but describe when terms can fire, rather than defining how operators behave. Hence, for a.b.c.0 we would first construct the ruloids for the open term a.b.c.x and its derivatives:

$$\frac{}{\mathsf{a.b.c.}x \xrightarrow{\mathsf{a}} \mathsf{b.c.}x} \qquad \frac{}{\mathsf{b.c.}x \xrightarrow{\mathsf{b}} \mathsf{c.}x} \qquad \frac{}{\mathsf{c.}x \xrightarrow{\mathsf{c}} x} \qquad \frac{x \xrightarrow{a} y, \quad a \in \mathsf{Act}}{x \xrightarrow{a} y}$$

and then generate the three-state OOG for a.b.c.x (treating it like an operator with argument x), and finally apply it to the BDD for 0 to generate a BDD with just two state bits instead of three.

The ruloid construction just described in effect explores the whole state space of the term, which could be prohibitively expensive in the case of a larger term, such as a.b.c.$(P_1 \mid \ldots \mid P_n)$, with the P_i's large. However, there is no need to determine the ruloids for the whole term; in this case, we would generate the ruloid OOG for a.b.c.x, and pass the BDD generated in the standard way from $(P_1 \mid \ldots \mid P_n)$ as the argument to this OOG. At the moment, the extent to which the ruloid construction is applied is specified by the user.

4 Handling Explicit Recursion

In [Ace93], it is described how a Simple system (Σ_G, R_G) may be extended with a finite set of names, $\mathcal{N}$, and a mapping, Δ, from $\mathcal{N}$ to terms over $\Sigma_G \bigcup \mathcal{N}$.

$\Delta(X)$ must be of the form $g(\vec{Y})$, where $\vec{Y}$ are names, and all the rules for every operator reachable from g must have no antecedents. The system is also extended with the rule:

$$\frac{\Delta(X) \xrightarrow{a} t}{X \xrightarrow{a} t}$$

Unfortunately, this extension excludes many useful recursive terms, such as the following CCS definition:

$$X = \mathsf{a}.X + \mathsf{b}.X$$

since both rules for the CCS operator $+$ have an antecedent. We therefore present an alternative extension of a Simple system with explicit recursion that overcomes this problem and is amenable to our method for constructing BDD models of terms.

Definition 5. Given a Simple system (Σ_G, R_G), an operator $f \in \Sigma_G$ is *projective* if every rule ρ for f in R_G is of one of the following forms:

$$\frac{}{f(\vec{x}) \xrightarrow{a} x} \qquad \frac{}{f(\vec{x}) \xrightarrow{a} g} \qquad \frac{x_i \xrightarrow{a} y}{f(\vec{x}) \xrightarrow{b} z}$$

Definition 6. An argument position i is *guarded* for a projective operator f if there is no rule for f of the form

$$\frac{x_i \xrightarrow{a} y}{f(\vec{x}) \xrightarrow{b} y}$$

A *non-guarded* position is one that is not guarded.

In CCS, the parallel composition, renaming and restriction operators are not projective, the summation operator is projective but not guarded in any position, while the prefixing operators are both projective and guarded in their only argument position. In general, static operators are not projective.

Definition 7. A term t is projective-recursive with respect to a Simple system (Σ_G, R_G) and a finite set of variables $\mathcal{X}$ if

1. It is generated using the grammar:

$$t ::= \mu X.t \mid f(\vec{t}) \mid X$$

   where $f \in \Sigma_G$ is a projective operator, and $X \in \mathcal{X}$
2. Every occurrence of a variable $X \in \mathcal{X}$ in t appears in a guarded position of an operator, and
3. t has no free variables.

We also add the standard rule for recursion:

$$\frac{t[X := \mu X.t] \xrightarrow{a} t'}{\mu X.t \xrightarrow{a} t'}$$

Theorem 8. *Let G be a* Simple *rule system, and t a closed* projective-recursive *term over G. Then the transition system of t is finite state.*

Proof. (Sketch) Consider any execution sequence $t = t_0 \xrightarrow{a_0} t_1 \xrightarrow{a_1} t_2 \ldots$. We show that the size of the terms in this sequence is bounded: The only time there is an increase in size after a transition $t_i \xrightarrow{a_i} t_{i+1}$ is when a subterm of t_i of the form $\mu X.t'$ fires. Now, a guarded subterm of a projective-recursive term remains guarded unless the term evolves into that subterm. Since all occurrences of X in t' must be guarded, the next time $\mu X.t'$ fires will be when the execution sequence evolves into the term $\mu X.t'$. Since the number of subterms of the form $\mu X.t'$ that can fire in the execution sequence is determined by the number of such subterms in t, there are only finitely many terms t_i that are strictly larger than all t_j for $j < i$.

We have extended our BDD construction to handle these terms as well, by redirecting transitions that would lead to occurrences of recursive variables back to the states representing points where they are bound.

Theorem 9. *The BDD construction extended for projective recursion yields BDD's corresponding to the transition system semantics.*

5 Implementation

We have implemented our algorithm using C++ and a BDD library[Lon93]. Given the SOS rules of a Simple system G, our program will generate a BDD model for any G-term. We also implemented the BDD interpreter for the μ-calculus described in [BCM+90], so that we could use the μ-calculus encoding of the bisimulation relation to check bisimilarity.

In Figure 5, we compare the performance of our system with that of [EFT93], where CCS terms representing the specification and implementation of a simple distributed scheduler [Mil89] are checked for bisimilarity. In [EFT93], the static CCS operators are built into the system as operators on BDD's for transition systems, and the dynamic operators are not available as operators. In contrast, our system took as inputs (1) an SOS specification of CCS, (2) the system to verify, written in CCS and (3) an indication of which subterms of the system to apply the ruloid construction to; it can handle any program in any Simple process algebra. In both cases, the tool requires a modest amount of hand-tuning of the inputs. We used input (3) to optimize the dynamic components, while [EFT93] explicitly provided the transition system BDD's of the dynamic components as part of the input.

Using the optimizations described in Section 3.3, we actually generate slightly smaller BDD's than in [EFT93]. We also have better run times, though this should not be taken seriously as we are running on slightly different hardware and using different BDD libraries. However, this does suggest that our method is probably competitive with hand-coded algorithms tuned to the most common special case.

N	States	Transitions	Size [EFT93]	Size Simple	Time[6] [EFT93]	Time[6] Simple
6	577	2 017	427	419	21	5
8	3 073	13 825	651	633	40	13
10	15 361	84 481	907	882	87	30
12	73 729	479 233	1 200	1 167	145	55
14	~ 300 000	2 000 000	1 528	1 488	233	95
16	~ 1 200 000	8 000 000	1 897	1 861	348	196
18	~ 4 800 000	32 000 000	2 297	2 254	569	522

Fig. 2. Comparative BDD sizes/verification times (seconds) for the scheduler.

Buffer Capacity	States	BDD size	Time Simple	Time CWB (v6.1)
1× 2	126	314	4	4
2× 2	750	479	18	200
3× 2	3850	639	88	12050
4× 2	18278	799	465	> 1 week

Fig. 3. Verification times (seconds) for the alternating bit protocol.

We are verifying other protocols to see how well our methods scale to larger problems; *e.g.*, the alternating-bit protocol with various buffer capacities[Mil89], for which a comparison with version 6.1 of the Concurrency Workbench[CWB92] is shown in Figure 3.

6 Further Work

There are several ways in which we can extend this work. First, we could carry out this program for other classes of finitary process algebras. We used Simple because it has a very clean syntactic characterization, which makes it easier to deal with than classes of algebras determined by term rewriting techniques. We hope to eventually handle higher order process algebras, such as the π-calculus, for which a model checking procedure has appeared[Dam93]. In a different direction, we could look for optimizations and heuristics to tune the model checking process for bisimulation, for example, by abstracting systems. As part of a larger project, we are integrating this tool into a general process algebra toolkit that will use a variety of techniques, including equational theorem proving, for bisimilarity checking.

[6] For Simple, the time includes computing the τ-closure and bisimulation relation on a SUN 670, using the model checking algorithm and μ-calculus formula for bisimulation in [BCM+90]. For [EFT93], the times are for the same operations, but on a Sun Sparcstation 2.

References

[ABV94] Luca Aceto, Bard Bloom, and Frits Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, May 1994. (Special Issue for LICS '92).

[Ace93] Luca Aceto. GSOS and finite labelled transition systems. Technical Report 6/93, University of Sussex at Brighton, March 1993.

[BCL91] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *VLSI 91*, Edinburgh, Scotland, 1991.

[BCM+90] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science*, 1990.

[BdS92] A. Bouali and R. de Simone. Symbolic bisimulation minimization. In *Computer Aided Verification*, volume 663 of *LNCS*, 1992.

[Blo90] Bard Bloom. *Ready Simulation, Bisimulation and the semantics of CCS-like languages.* PhD thesis, MIT, Cambridge, Massachusetts, october 1990.

[Bry86] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 8 1986.

[CWB92] The Edinburgh Concurrency Workbench, version 6.1, 1992. Available by ftp from ftp.dcs.ed.ac.uk, directory export/cwb/CWB6.1.

[Dam93] Mads Dam. Model checking mobile processes. *CONCUR 93*, 1993.

[DB95] A. Dsouza and B. Bloom. Applying symbolic model checking to process algebras, 1995. http: //www.cs.cornell.edu/Info/People/dsouza/pa2bdd.ps.

[dS85] R. de Simone. Higher-level synchronizing devices in MEIJE-SCCS. *Theoretical Computer Science*, 37(3):245–267, 1985.

[dSV89] R. de Simone and D. Vergamini. Aboard Auto. Technical Report Rapports Techniques 111, INRIA, Sophia Antipolis, 1989.

[EFT93] R. Enders, T. Filkhorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6:155–164, 1993.

[Fer89] J-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989.

[FKM93] J-C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic equivalence checking. In *Computer Aided Verification*, 1993.

[GV90] J.F. Groote and F. Vandraager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *ICALP '90*, Lecture Notes in Computer Science. Springer Verlag, 1990.

[Lon93] David Long. bdd: A Binary Decision Diagram (BDD) package, 1993. Available by FTP from emc.cs.cmu.edu (pub/bdd/bddlib.tar.Z).

[LOT] LOTOSphere is available by FTP from ftp.cs.utwente.nl.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1989.

[MM93] G. Milne and G. McCaskill. Sequential circuit analysis with a BDD based process algebra system. Technical Report HDV-25-93, University of Strathclyde, 1993.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, Lect. Notes in Computer Sci., page 261. Springer-Verlag, 1981.