Checking for Equivalence of Communication Behaviors using BDD

by

O. Grumberg, T. Djerassi-Shintel, N. Shintel, E. Aharonson

Technical Report #729
May 1992
CHECKING FOR EQUIVALENCE OF COMMUNICATION BEHAVIORS USING BDD

Orna Grumberg Taly Djerassi-Shintel Noam Shintel
Eran Aharonson

Computer Science Department
Technion
Haifa 32000, Israel
email: orna@cs.technion.ac.il, shintel@cs.technion.ac.il, taly@cs.technion.ac.il

May 21, 1992
We would like to thank David Long for his help and useful advice.
# Contents

1 Introduction 1

2 The Model 2
   2.1 Processes and Communication Behaviors 2
   2.2 Equivalence of Communication Behaviors 3

3 Checking for Equivalence of States in Communication Behaviors 4
   3.1 Deterministic Structures 4
   3.2 Checking for Equivalence of States 5

4 The Implementation 6
   4.1 The Representation of an Automaton 6
   4.2 Notations and Conventions 6
   4.3 Communication Behaviors 7
   4.4 Building the Deterministic Structure 8
      4.4.1 $\text{inf}_c$ Transitions 9
      4.4.2 The Deterministic Structure 9
      4.4.3 $\varepsilon$-Closure 9
      4.4.4 Renaming 10
      4.4.5 Finding the Name of an Existing State 11
   4.5 Calculating State Equivalence 11
      4.5.1 State Equivalence 11
      4.5.2 The Separating Communication Sequence 12

5 Input and Output 13
   5.1 Input 13
   5.2 Output 14

6 About the C Source Code 15
   6.1 About the Functions 15
   6.2 The Hash Implementation 15
   6.3 The Modules 15

List of Figures

4.1 BDD Variables Ordering 7
4.2 The Determinization Algorithm 9
4.3 Naming Scheme 10
4.4 Algorithm for State Equivalence 12
5.1 Example: Description of an Automaton 13
6.1 Calculating the Hash Key 15
6.2 The Hash Table Structure 16
1 Introduction

In his Master Project Thesis, Rami Marelly, under the supervision of Dr. Orna Grumberg, presented a tool for automatic verification of distributed algorithms. In the different phases of the algorithm, huge automata are created which caused a problem of very large memory consumption, exceeding the capabilities of many computer systems. In order to reduce the memory consumption (at least in the average case), a realization using BDD was suggested.

A Binary Decision Diagram (abbreviated BDD), presented by Bryant in [Bry86], is a canonical data structure for the representation of boolean functions. A binary function is represented by a directed acyclic graph. In the worst case this graph might be exponential in size, but in typical applications it is much smaller. The time complexity of most manipulations on this structure is proportional to its size.

Our project is intended to check the feasibility of an implementation using BDD, on three of the major bottle-necks of the algorithm:

- Given two automata, representing two processes, build an automaton which represents their communication behavior.

- Convert non-deterministic automata which represent communication behaviors into deterministic ones. This step is a necessity for the next one.

- For a given pair of states, in two, possibly different communication behaviors, check whether they are state equivalent.

The project is implemented in C using a BDD - library written by David Long.

This document contains three major parts:

1. A brief description of the model and the problem to which the project relates.

2. Description of the algorithms used.

3. C - source documentation.

This project was performed in the frame of the Software Verification Lab under the supervision of Dr. Orna Grumberg.
2.2 Equivalence of Communication Behaviors

Definition 2.5: A path \( \Pi \) in \( P : Q \) and a path \( \Pi' \) in \( P' : Q' \) are communication equivalent if their communication sequences (\( \varepsilon \)-transitions are ignored) are equal, and in addition, if the communication sequences are finite then:

- \( \Pi \) is finite iff \( \Pi' \) is finite.
- \( \Pi \) contains a sink (sink) state iff \( \Pi' \) contains a sink (sink) state.
- \( \Pi \) contains an infinite number of \( \varepsilon \) (\( \varepsilon \)) transitions iff \( \Pi' \) contains an infinite number of \( \varepsilon \) (\( \varepsilon \)) transitions.

Definition 2.6: Let \( P : Q \) and \( P' : Q' \) be communication behaviors such that \( ACT_{P:Q} = ACT_{P':Q'} \). The states \( s \in P : Q \) and \( s' \in P' : Q' \) are equivalent iff:

- For each path \( \Pi \) from \( s \) there exists a communication equivalent path \( \Pi' \) from \( s' \).
- For each path \( \Pi' \) from \( s' \) there exists a communication equivalent path \( \Pi \) from \( s \).
3 Checking for Equivalence of States in Communication Behaviors

The algorithm presented here is the relevant part of the algorithm in [Mar91].

3.1 Deterministic Structures

Definition 3.1: Let us define the following three predicates:
\[ \text{loop}_\varepsilon(s) = \text{TRUE} \iff \text{there exists a state } s' \in C L_\varepsilon(s) \text{ from which there is a loop containing } \varepsilon \text{ edges only} \]
\[ \text{loop}_\varepsilon(s) = \text{TRUE} \iff \text{there exists a state } s' \in C L_\varepsilon(s) \text{ from which there is a loop containing } \varepsilon \text{ edges only} \]
\[ \text{loop}_{\varepsilon\varepsilon}(s) = \text{TRUE} \iff \text{there exists a state } s' \in C L_\varepsilon(s) \text{ from which there is a loop containing } \varepsilon \text{ and } \varepsilon \text{ edges only, and at least one of each kind} \]
\[ C L_\varepsilon(s) \text{ is the set of states reachable from } s \text{ along } \varepsilon \text{ and } \varepsilon \text{ edges only.} \]

Definition 3.2: Given a communication behavior \( P : Q \) the deterministic structure is \( D_{P,Q} = (U,V) \) where:

- \( U \) is the set of states of \( D_{P,Q} \):
  \[ U = 2^{s_{P,Q}} \cup \{ \emptyset \} \]
  \[ I = \{ \text{sink, sink} \} \cup \{ \inf_{\varepsilon}, \inf_{\varepsilon}, \inf_{\varepsilon\varepsilon} \} \]

- \( V \), the transition relation, contains the following transitions:
  \[ (U_1, \alpha, U_2) \iff U_2 = \{ s' \mid \exists s \in U_1, \exists s'' \in C L_\varepsilon(s) : (s'', \alpha, s') \in R_{P,Q} \} \]
  \[ (U_1, \varepsilon, \inf_{\varepsilon}) \iff \exists s \in U_1 : \text{loop}_\varepsilon(s) = \text{TRUE} \]
  \[ (U_1, \varepsilon, \inf_{\varepsilon\varepsilon}) \iff \exists s \in U_1 : \text{loop}_{\varepsilon\varepsilon}(s) = \text{TRUE} \]
  \[ (U_1, \alpha, \text{sink}) \iff \exists s \in U_1, \exists s' \in C L_\varepsilon(s) : (s', \alpha, \text{sink}) \in R_{P,Q} \]
  \[ (U_1, \alpha, \text{sink}) \iff \exists s \in U_1, \exists s' \in C L_\varepsilon(s) : (s', \alpha, \text{sink}) \in R_{P,Q} \]

Definition 3.3: A structural equivalence \( E \) on the states of \( D \) is defined inductively as follows:

\[ E_0 = \{ (U, U') \mid U \in D - I \land U' \in D - I \} \cup \{ (\text{sink, sink}), (\text{sink, sink}), (\inf_{\varepsilon}, \inf_{\varepsilon}), (\inf_{\varepsilon}, \inf_{\varepsilon}), (\inf_{\varepsilon\varepsilon}, \inf_{\varepsilon\varepsilon}) \} \]

\[ E_{n+1} = \{ (U, U') \mid U \in E_n U' \land \forall U_1(U, \alpha, U_1) \implies (\exists U'_1 : (U', \alpha, U'_1) \land U_1 E_n U'_1) \land \forall U'_1(U', \alpha, U'_1) \implies (\exists U_1 : (U, \alpha, U_1) \land U_1 E_n U'_1) \} \]

\[ E = \bigcap_n E_n \]
3.2 Checking for Equivalence of States

Given two communication behaviors $P : Q$ and $S : T$, do:

1. Build $D_{P,Q}$ and $D_{S,T}$.
2. Partition the states in $V_{P,Q} \cup V_{S,T}$ into structural equivalence classes.
3. Two states, $U, U' \in V_{P,Q} \cup V_{S,T}$ are equivalent iff $UEU'$. 
4 The Implementation

In this section we present the algorithms, data and BDD structures used in the implementation of the project.

4.1 The Representation of an Automaton

The BDD representation of an automaton $M$ is a set of relations $(s, \alpha, s')$, where each state of $M$ is encoded by a unique binary vector of a constant length $n$, and each action label is encoded by a unique binary vector of a constant length $m$.

For the representation of a state $s$ we use $2n$ variables, $v_0, \ldots, v_{n-1}$ are used for the representation of $s$ as a source state, $v_n, \ldots, v_{2n-1}$ which are the primed variables of $v_0, \ldots, v_{n-1}$, respectively, are used for the representation of $s$ as a destination state. Unpriming a BDD which contains primed variables only, results in a BDD in which each variable is replaced with its unprimed variable. Therefore, unpriming a BDD representing $s$ as a destination state, results in a BDD representing $s$ as a source state, and vice versa.

In a BDD representing a transition, there are $2n + m$ variables, $v_0, \ldots, v_{n-1}$ which are used to represent the source state $s$, $v_n, \ldots, v_{2n-1}$ which are used to represent the destination state $s'$, and $v_{2n}, \ldots, v_{2n+m-1}$ which are used to represent the action label. Let $x_0x_1\ldots x_n$ be the binary encoding of a state $s$. Let $x_nx_{n+1}\ldots x_{2n-1}$ be the binary encoding of a state $s'$, and let $x_{2n}x_{2n+1}\ldots x_{2n+m-1}$ be the binary encoding of a label $\alpha$, then the BDD representing $(s, \alpha, s')$ is the BDD that represents the formula:

$$\bigwedge_{i=0}^{2n+m-1} y_i, \quad y_i = \begin{cases} \neg v_i & x_i = 0 \\ v_i & x_i = 1 \end{cases}$$

(4.1)

Let $R_0, \ldots, R_k$ be the BDD representing all transitions of $M$, the BDD representing $M$ is:

$$\bigvee_{i=0}^{k} R_i$$

(4.2)

The ordering of variables in a BDD is a major factor in its efficiency. Therefore, the variables ordering in our implementation is different from the straight forward one presented above, and is shown in figure 4.1. The interleaving between source and destination states is due to the implementation of the BDD package.

4.2 Notations and Conventions

- We write $P(p, l, p')$ to denote a boolean function $P$ with parameters $p, l, p'$.

- Here, a label (e.g. $\epsilon$) or a state (e.g. $s_0$) is not a constant but a boolean function. The BDD $\exists l [P(p, l, p') \land \alpha]$ represents all couples $(p, p')$ such that $(p, \alpha, p')$ is a transition in $P$. $\exists l$ quantifies out the label variables.

- In our implementation, a BDD encodes a set of transitions i.e. an automaton. It is sometimes necessary to perform a negation ($\neg$) operation on a BDD encoding a partial set of transitions, in order to get a BDD representing the complementary set. In most cases this results in a set containing illegal states and transitions which are not part
The computation here is different from definition 3.2 in that here the transition relation is defined for reachable states only.

4.4 Building the Deterministic Structure

We build a slightly different structure than the one described in section 3.1. We do not distinguish between \( \varepsilon \) and \( \overline{\varepsilon} \) since these actions lead to different states in the deterministic structure, therefore there is no loss of information in using the same label. The deterministic structure is built in two stages:

1. Calculating the \( \varepsilon \)-loop predicates and adding inf \( \varepsilon \) transitions to the non-deterministic structure.

2. Building the deterministic structure.
4.4.1 \( \inf_\varepsilon \) Transitions

The set of all states with an infinite outgoing \( \varepsilon \) path is given by the following equation:

\[
L_\varepsilon(p) = \nu X[\exists l, p'((\text{prime}(X))(p') \land P(p, l, p') \land \varepsilon)]
\]  

(4.8)

The following equation adds \( \inf_\varepsilon \) transitions to \( P : Q :\)

\[
P : Q := P : Q(s, l, s') \lor \exists l, s' (P : Q(s, l, s')) \land \\
[ (L_\varepsilon(p) \land \varepsilon \land \inf_\varepsilon) \lor \\
(L_\varepsilon(q) \land \varepsilon \land \inf_\varepsilon) \lor \\
(L_\varepsilon(p) \land L_\varepsilon(q) \land \varepsilon \land \inf_\varepsilon) ]
\]  

(4.9)

4.4.2 The Deterministic Structure

The algorithm for building the deterministic structure, presented in figure 4.2, is basically straight-forward. The operators \( \varepsilon \)-closure and rename will be further discussed later on.

\[
D(s, l, s') := 0 \\
todo := \{s_0, p, q\} \\
done := {} \\
\text{while } todo \neq {} \\
\quad \text{get state from todo} \\
\quad \text{for each } \alpha \in \Sigma \\
\quad \quad \text{if } \inf_\varepsilon \in \varepsilon\text{-closure(state) then } D := D \lor (\text{rename(state)}, \alpha, \inf_\varepsilon) \\
\quad \quad \text{if } \inf_\varepsilon \in \varepsilon\text{-closure(state) then } D := D \lor (\text{rename(state)}, \alpha, \inf_\varepsilon) \\
\quad \quad \text{if } \inf_\varepsilon \in \varepsilon\text{-closure(state) then } D := D \lor (\text{rename(state)}, \alpha, \inf_\varepsilon) \\
\quad \quad \text{dest} := \exists s, l (P : Q(s, \alpha, s') \land \varepsilon\text{-closure(state)}) \\
\quad \quad \text{if sink } \in \text{dest then } D := D \lor (\text{rename(state)}, \alpha, \text{sink}) \\
\quad \quad \text{if } \text{sink in dest then } D := D \lor (\text{rename(state)}, \alpha, \text{sink}) \\
\quad \quad D := D \lor (\text{rename(state), \alpha, rename(dest)}) \\
\quad \quad todo := todo \cup \{dest\} \\
\quad \quad done := done \cup \{state\} \\
\quad todo := todo \setminus done
\]

Figure 4.2: The Determinization Algorithm

4.4.3 \( \varepsilon \)-Closure

Given a set of states \( S \), \( \varepsilon \)-closure(\( S \)) is found using the following equation:

\[
\varepsilon\text{-closure}(S) = \mu X[ S \lor \text{unprime}(\exists s, l (X(s) \land P : Q(s, l, s') \land \varepsilon))] 
\]  

(4.10)
4.4.4 Renaming

Since the number of states in the deterministic automaton might be larger than in the non-deterministic one, it is necessary to transform the encoding of each non-deterministic state into a new encoding. There are three major requirements of the name generator, namely:

1. Each name generated is unique.
2. Each deterministic state (which is a subset of the non-deterministic states) has only one name.
3. The number of variables used to represent the names is minimal.

We use the following naming scheme:

\[
\begin{align*}
cname & := 0 \\
\#vars & := 0 \\
\text{for a new state } s, \text{ do:} \\
\text{name}(s) & := cname \\
cname & := cname + 1 \\
\text{if } cname > 2\#vars \text{ then } \#vars & := \#vars + 1 \\
\end{align*}
\]

Figure 4.3: Naming Scheme

It is mandatory that given a deterministic state (i.e. a subset of non-deterministic states), its generated name can be found. Therefore some kind of a list containing the generated states and their names must be kept. A logical solution would be to keep the BDD representing the non-deterministic states which the deterministic one contains, together with the BDD representing its name. Using this method, an addition of a new variable to the names of deterministic states requires updating the names generated so far, i.e. each BDD representing an already generated name should be changed so that it includes the new variable. We also have to update the previously generated transitions. We avoid the high time complexity involved in going over a long list of names by keeping an integer representing the name instead of a BDD. A generation of a BDD representing a name, from the integer representation, is done in a straightforward way using its binary representation and the number of existing variables (e.g. if the number of existing variables is 4 (i.e. there are \(v_0, v_1, v_2, v_3\)), then the BDD representing the name 5 is \((\neg v_3 \land v_2 \land \neg v_1 \land v_0)\). When a new variable is added, we update the set of already generated transitions, by anding the BDD which represents this set with the negation of the new variable. Saving integers instead of BDD's also results in a lower space complexity.

Note that since there are two sets of variables used to represent a name (primed and unprimed) we actually add two variables instead of one. Also, in our implementation, the first five names represent the special states (\(\text{inf}_e\) etc.).
4.5.2 The Separating Communication Sequence

During the computation of $E$ we keep a set $SEP$ defined as follows:

$$SEP = \{(s_1, s_2, l) \mid \exists E_i, s'_1, s'_2 \ ( (s_1, s_2) \in E_i \land (s_1, l, s'_1) \in M_1 \land (s_2, l, s'_2) \in M_2 \land (s'_1, s'_2) \notin E_i) \}$$ (4.15)

In our implementation, for each separated couple of states, only a single separating label is kept. Also, the couples separated in $E_1$ are not kept.

Given two states $s_1, s_2$ as above, and the sets $E$ and $SEP$ the algorithm for checking whether they are equivalent is as follows:

```plaintext
if $(s_1, s_2) \in E$ return TRUE
else, while there is $\alpha$ such that $(s_1, s_2, \alpha) \in SEP$
    print $\alpha$
    $s_1 := \text{unprime}(\exists s_1, l \ (M_1(s_1, l, s'_1) \land \alpha))$
    $s_2 := \text{unprime}(\exists s_2, l \ (M_2(s_2, l, s'_2) \land \alpha))$
    find $\alpha$ such that there are $s'_1, s'_2$ satisfying $[(s_1, \alpha, s'_1) \in M_1 \land (s_2, \alpha, s'_2) \notin M_2 \lor (s_1, \alpha, s'_1) \notin M_1 \land (s_2, \alpha, s'_2) \in M_2]$
    print $\alpha$
return FALSE
```

Figure 4.4: Algorithm for State Equivalence
5 Input and Output

5.1 Input

The input to the program is a representation of four automata: $P$, $Q$, $S$, $T$, from which $D_{P:Q}$ and $D_{S:T}$ are built. The program gets its input from stdin which means that it can be a file or some kind of a compiler which generates the automata.

The input format for each automaton is as follows:

- Number of bits used to represent a label ($m$)
- Newline
- Number of bits used to represent a state ($n$)
- Newline
- A list of relations in the form:
  - A binary vector of length $n$ representing the source state
  - A binary vector of length $n$ representing the destination state
  - A binary vector of length $m$ representing the label
  - Newline
- The number 2 which marks the end of the automaton definition

The definition of each automaton follows the definition of the previous one. An example of an automaton's encoding is in figure 5.1.

![Automaton Diagram](image)

Figure 5.1: Example: Description of an Automaton

We assume the following:

1. The initial state of an automaton is encoded as $[0]_2$. 
2. \( \varepsilon \) is encoded as \([0]_2\).

3. Each state in the automaton has at least one outgoing edge, which is true for communication processes.

4. All four automata have the same alphabet, therefore the same number of bits, and the same variables set, are used for encoding a label.

5. In each automaton there are at least three bits for the representation of a state. In different automata, the variables used for encoding states, are disjoint.

Since the program is intended to be used as part of a larger program, state pairs which should be checked for equivalence are encoded in the program and are not an input. At this stage, this part is used for debugging purpose only.

5.2 Output

The program has two output streams:

1. stdout to which equivalence results and errors are directed.

2. A file named proj.out to which data on the state of the program, and average running times of various parts, is written.
6 About the C Source Code

In this section we describe some of the major properties of the program functions, we describe the hash implementation, and we provide a short description of each of the C - modules.

In the C - implementation we have reversed the role of primed and unprimed variables (i.e. a source state is represented by primed variables).

6.1 About the Functions

Most of the memory allocation is done while a new BDD is created. Therefore, when a BDD is created as an intermediate result, it should be freed as soon as it is no longer needed. Most of the functions in our implementation follow this rule. The only exception is the function det() which builds a new deterministic automaton from a non-deterministic one. This function assumes that the BDD - manager of the old automaton will be dropped.

In the process of creating the deterministic automata we use three BDD managers: one is used for \( P, Q \) and \( P : Q \), the second for \( S, T \) and \( S : T \) and the third for \( D_{P,Q} \) and \( D_{S,T} \). The first two are dropped after the deterministic representation is built.

There are no global variables in the implementation, with the exception of the output file descriptor. All the parameters a function requires should be passed by the caller.

6.2 The Hash Implementation

The module hash.c is the one in which the hash functions are. The function hash_func10() calculates the key to the hash table from the input BDD (which is a unique pointer) as described in figure 6.1

Let the hash table size be \( 2^k \). Do:
- Cast off the two least significant bits (most 32-bit machines use 4-byte alignment)
- From the remaining bits, use the \( k \) least significant ones as the key.

Figure 6.1: Calculating the Hash Key

The hash table structure is described in figure 6.2. The todo pointer links the todo list.

6.3 The Modules

proj.c

This module contains the function: main(). The state pairs which are checked for equivalence are encoded in main().

behav.c

In this module there are functions performing two tasks:

1. Calculating communication behaviors.
2. Calculating the equivalence class - $E$ (see sub-section 4.5).

$\textit{det.c}$
Contains the functions which determinize a communication behavior.

$\textit{build.c}$
Contains functions to build the non-deterministic automata.

$\textit{state.c}$
Contains utility functions which build BDD's representing states.

$\textit{eps.c}$
Deals with $\varepsilon$, i.e. $\varepsilon$-closure and $\varepsilon$ loops.

$\textit{hash.c}$
Implements the hash table as described above.
References

