Contents

Abstract

Chapter 1

Introduction

Verifying computer systems is essential. They have become such an important part of our lives that unforeseen software bugs and malfunction can lead to disastrous situations. One of many examples is the 1996 Arianne V satellite launcher, whose internal software caused the self-destruction of the module as a result of a simple but erroneous float-to-integer conversion. Another example of bad software is the Therac-25 radiation therapy machine, which massively overdosed six patients between 1985 and 1987, causing the death of at least three of them.

Our dependency on computer systems is increasing, but so is their *complexity* which, more often than not, demands a large amount of skilled human maintenance. Financially, this has motivated the need for systems to be more independent, and consequently, a large number of the critical systems that we rely on are now *autonomous*: they are able to "make decisions" without human intervention, based on a variety of factors such as their current state or environment.

For the majority of current software systems, the verification is done by manual or automated *testing*, with limited efficiency, as it is generally impossible to test *all* possible situations. This is even more true in the case of autonomous systems, where the number of situations which might occur is much higher than usual. In the context of safety-critical systems, we want to be able to *prove* that they meet their specification, and function correctly without bugs.

The verification of autonomous systems has been a very active research area in the past few decades due to their rising importance, and a range of verification techniques have been developed based on formal methods for modelling systems, including automated theorem proving or *model checking*, the latter being the focus of this project.

Model checking was introduced in 1981 by E. M. Clarke and E. A. Emerson [1], and independently by J. P. Queille and J. Sifakis [3], and is found to have major advantages [2]: no correctness proofs are required (it is an automatic process), it finds counterexamples (i.e. bugs), and it allows for many different properties to be

checked, by means of temporal logics.

Unfortunately, an important disadvantage of model checking, known as the *state explosion problem*, is the difficulty in verifying larger systems efficiently. In the past 20 years, several techniques attempting to solve the state explosion problem have been introduced, such as predicate abstraction, bounded model checking, symmetry reduction, and (perhaps more notably, at least for this project) symbolic model checking with *ordered binary decision diagrams* (OBDDs, [20]).

The idea of symbolic model checking, introduced in 1992 by K. L. McMillan [4], is to encode states and transitions of the system as *Boolean functions*, and represent these symbolically using efficient and compact data structures such as OBDDs. Symbolic model checking with OBDDs has allowed for very large systems to be verified, and has therefore seen a rise in the popularity of model checking as a verification technique. Although OBDD-based techniques have significantly improved the efficiency of model checking, they have limitations and only allow systems up to about 10^{20} states to be verified [5]. Beyond, the OBDDs generated become too large to be efficiently handled by today's computers.

Several alternatives have been proposed to the use of OBDDs in symbolic model checking. In particular, a number of SAT-based methods have been introduced [27, 25, 26], demonstrating potential to outperform OBDDs in some situations, though these tend to focus on *bounded* model checking, which only verifies part of a system.

On the other hand, no other *representations* of Boolean functions have been as influential as OBDDs in symbolic model checking. Experiments have been conducted with

, and some recent results [10, 28] have proposed the symbolic encoding of systems using multi-valued logics, leading to potentially more compact representations.

In this project, we explore the possibility of replacing OBDDs with a new symbolic representation of (classical two-valued) Boolean functions known as the *sentential decision diagram* (SDD), first introduced in 2011 by A. Darwiche [7]. SDDs were originally introduced as an alternative data structure in the field of *knowledge compilation* and, until this project, they had never been experimented with in the context of model checking.

1.1 Contributions

This project constitutes a first investigation of the use of sentential decision diagrams in symbolic model checking. Our contributions include:

1. A theoretical approach to model checking with SDDs, determining the different aspects of the process and the choices involved;

2. An implementation of a model checker for multi-agent systems entirely based on SDDs;

3. Some heuristics for a more efficient use of SDDs in model checking multi-agent systems;

4. A thorough comparison of the above with OBDD-based model checking.

Chapter 2

Background

2.1 Logics for Multi-Agent Systems

2.1.1 Multi-Agent Systems

Autonomous multi-agent systems are computer systems which are made up of several intelligent "agents" acting within an "environment". Intuitively, an *agent* is:

- Capable of *autonomous* action
- Capable of *social* interaction with its peers
- Acting to *meet* their design objectives

Suppose we have a multi-agent systems consisting of n agents and an environment e.

Definition An agent i in the system consists of:

- A set L_i of local states representing the different configurations of the agent,
- A set Act_i of local actions that the agent can take,
- A protocol function $P_i : L_i \to 2^{Act_i}$ expressing the decision making of the agent.

We can define the environment e as a similar structure (L_e, Act_e, P_e) where P_e represents the functioning conditions of the environment.

2.1.2 Interpreted Systems

We present a formal structure to represent multi-agent systems. Consider a multi-agent system Σ consisting of n agents $1, ..., n$ and an environment e.

Definition An *Interpreted System* IS for Σ is a tuple $(G, \tau, I, \sim_1, ..., \sim_n, \pi)$, where

- $G \subseteq L_1 \times ... \times L_n \times L_e$ is the set of global states that Σ can reach. A global state $g \in G$ is essentially a picture of the system at a given point in time, and the local state of agent i in g is denoted $l_i(g)$.
- $I \subseteq G$ is a set of intial states for the system
- $\tau : G \times Act \to G$ where $Act = Act_1 \times ... \times Act_n \times Act_e$ is a deterministic transition function (we can define $\tau : G \times Act \to 2^G$ to model a non-deterministic system)
- $\sim_1, ..., \sim_n \subseteq G \times G$ are binary relations defined by

  $$g \sim_i g' \Leftrightarrow l_i(g) = l_i(g') \quad \forall g, g' \in G, \forall i = 1, ..., n$$

  i.e iff agent i is in the same state in both g and g'.
- $\pi : PV \to G$ is a valuation function for the set of atoms PV, i.e for each atom $p \in PV$, $\pi(p)$ is the set of global states where p is true

We also need a formal definition for the "execution" of a system. A *run*, as defined below, represents one possible execution of a MAS.

Definition A *run* of an interpreted system $IS = (G, \tau, I, \sim_1, ..., \sim_n, \pi)$ is a sequence $r = g_0, g_1, ...$ where $g_0 \in I$ and such that for all $i \geq 0$, $\exists a \in Act$ such that $\tau(g_i, a) = g_{i+1}$.

Interpreted systems as defined above are used as semantic structures for a particular family of logics, presented in the next section.

2.1.3 Linear Temporal Logic

In order to verify properties of multi-agent systems, we first need to find a logic allowing us to describe these properties accurately.

A good candidate is the Linear Temporal Logic (LTL), a modal temporal logic in which one can write formulas about the future of *paths*. Here we use paths to represent infinite runs of an interpreted system.

Definition The syntax of LTL formulas is given by the following BNF:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid G\varphi \mid \varphi U \varphi$$

The intuitive meanings of $X\varphi, G\varphi$ and $\varphi U \psi$ are respectively

- φ holds at the ne**X**t time instant
- φ holds forever (**G**lobally)
- φ holds **U**ntil ψ holds

We define the unary operator F to be the dual of G, i.e $F\varphi := \neg G \neg \varphi$ for any LTL formula φ. $F\varphi$ represents the idea that φ will hold at some point in the **F**uture.

Semantics

A model for LTL is a Kripke model $M = (W, R, \pi)$ such that the relation R is serial, i.e $\forall u \in W, \exists v \in W$ such that $(u, v) \in R$. The worlds in W are called the *states* of the model.

Definition A *path* in an LTL model $M = (W, R, \pi)$ is an infinite sequence of states $\rho = s_0, s_1, ...$ such that $(s_i, s_{i+1}) \in R$ for any $i \geq 0$. We denote ρ^i the suffix of ρ starting at i (note that ρ^i is itself a path since ρ is infinite).

It is easy to see how such a model can be used to represent a computer system, and how an execution of this system can be written as a path.

Our objective is to be able to verify that a system S has property P, so if we encode P as an LTL formula φ_P and S as a model M_S, then we need to be able to check whether φ_P is *valid* in M (or at least true in a set of initial states). This technique is called *model checking*, and we do this by using the following definition for the semantics of LTL:

Definition Given LTL formulae φ and ψ, a model M and a state $s_0 \in W$, we say that

$$
\begin{aligned}
(M, s_0) \vDash p &\Leftrightarrow s_0 \in \pi(p) \\
(M, s_0) \vDash \neg\varphi &\Leftrightarrow (M, s_0) \nvDash \neg\varphi \\
(M, s_0) \vDash \varphi \wedge \psi &\Leftrightarrow (M, s_0) \vDash \varphi \text{ and } (M, s_0) \vDash \psi \\
(M, s_0) \vDash X\varphi &\Leftrightarrow (M, s_1) \vDash \varphi \text{ for all states } s_1 \text{ such that } R(s_0, s_1) \\
(M, s_0) \vDash G\varphi &\Leftrightarrow \text{for all paths } \rho = s_0, s_1, s_2, ..., \text{ we have } (M, s_i) \vDash \varphi \quad \forall i \geq 0 \\
(M, s_0) \vDash \varphi U \psi &\Leftrightarrow \text{for all paths } \rho = s_0, s_1, s_2, ..., \exists j \geq 0 \text{ such that } (M, s_j) \vDash \psi \\
&\qquad \text{and } (M, s_k) \vDash \varphi \quad \forall 0 \leq k < j
\end{aligned}
$$

The expressive power of LTL is limited to quantification over *all* possible paths. For example:

- $FG(\text{deadlocked})$

  In every possible execution, the system will be permanently deadlocked.

- $GF(\text{crash})$

  Whatever happens, the system will crash infinitely often.

Hence some properties cannot be expressed in LTL, as in certain applications we might want to quantify explicitely over paths. The Computation Tree Logic (CTL) can express this.

2.1.4 Computation Tree Logic

Definition The syntax of CTL formulae is defined as follows:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi U \varphi)$$

Intuitively, $EX\varphi, EG\varphi$, and $E(\varphi U \psi)$ represent the fact that there exists a possible path starting from the current state such that, respectively, φ is true at the next state, φ holds forever in the future, and φ holds until ψ becomes true.

The dual operator $AX\varphi := \neg EX\neg\varphi$ can be used to represent the fact that in all possible paths from the current state, φ is true at the next state. Connectives $AG\varphi, AF\varphi$, and $A(\varphi U \psi)$ can be defined in the same way.

We also use models (as defined in 2.1.3) for the semantics of CTL, as follows:

Definition Given CTL formulas φ and ψ, a model $M = (W, R, \pi)$ and a state $s_0 \in W$, the satisfaction of formulas at s_0 in M is defined inductively as follows:

$$\begin{aligned}
(M, s_0) \vDash p &\Leftrightarrow s_0 \in \pi(p) \\
(M, s_0) \vDash \neg\varphi &\Leftrightarrow (M, s_0) \nvDash \neg\varphi \\
(M, s_0) \vDash \varphi \wedge \psi &\Leftrightarrow (M, s_0) \vDash \varphi \text{ and } (M, s_0) \vDash \psi \\
(M, s_0) \vDash EX\varphi &\Leftrightarrow \exists \text{ a path } s_0, s_1, s_2, \ldots \text{ such that } (M, s_1) \vDash \varphi \\
(M, s_0) \vDash EG\varphi &\Leftrightarrow \exists \text{ a path } s_0, s_1, s_2, \ldots \text{ such that } (M, s_i) \vDash \varphi \quad \forall i \geq 0 \\
(M, s_0) \vDash E(\varphi U \psi) &\Leftrightarrow \exists \text{ a path } s_0, s_1, s_2, \ldots \text{ for which } \exists i \geq 0 \text{ such that } (M, s_i) \vDash \psi \\
&\qquad \text{and } (M, s_j) \vDash \varphi \quad \forall 0 \leq j < i
\end{aligned}$$

The quantifiers allow for more properties to be expressed, for example:

- $EF(AG(\text{deadlocked}))$

  It is possibly to reach a point where the process will be permanently deadlocked.

- $AG(EX(\text{reboot}))$

  From any state it is possible to reboot the system.

Again, some formulas can be expressed in LTL but not in CTL. For instance, the property that *in every path where p is true at some point then q is also true at some point* is expressed in LTL as $Fp \rightarrow Fq$ but there is no equivalent CTL formula. The logic CTL* combines the syntax of LTL and CTL to provide a richer set of connectives. We will not go into any more details regarding CTL*, but we refer the reader to [21] for more information.

2.1.5 The Epistemic Logic CTLK

In the case of multi-agent systems, we are interested in describing the system in terms of individual agents, and in particular their *knowledge*.

For this reason, we can add [22] a family of unary operators K_i for $i = 1, ..., n$ to the modal connectives defined previously. Each K_i will represent the intuitive notion of knowledge for agent i. This enables us to define the temporal-epistemic logics LTLK and CTLK, which are extensions of LTL and CTL, respectively. Here we leave out details about LTLK, as the practical applications we present later on only support CTLK.

Definition The syntax of CTLK is defined by the following BNF:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi U \varphi) \mid K_i\varphi \quad (i \in \{1, ..., n\})$$

We use interpreted systems (2.1.2) as semantic structures for CTLK. The satisfaction of a CTL formula on an interpreted system IS is defined analogously to its satisfaction on a model M whose worlds W are the global states of IS, and whose relation function R is the global transition function of IS. For example, $(IS, g_0) \vDash EX\varphi$ iff there is a run $r = g_0, g_1, g_2, ...$ of IS such that $(IS, g_1) \vDash \varphi$.

The following definition completes the semantics of CTLK formulae:

Definition Given an interpreted system IS, a global state g, an agent i of IS, and a CTLK formula φ, we define

$$(IS, g) \vDash K_i\varphi \text{ iff } \forall g' \in G, g \sim_i g' \Rightarrow (IS, g') \vDash \varphi$$

The connective K_i expresses that agent i *knows* of the property φ when the system's global state is g.

We extend the syntax and semantics of CTLK by adding two extra unary operators: E (Everybody knows) and C (Common knowledge), whose semantics are defined as follows:

$$\begin{aligned}
(IS, g_0) \vDash E\varphi &\Leftrightarrow (IS, g_0) \vDash K_i\varphi \quad \forall i = 1, ..., n \\
(IS, g_0) \vDash C\varphi &\Leftrightarrow (IS, g_0) \vDash \bigwedge_{k=1}^{\infty} E^{(k)}\varphi \\
&\qquad \text{where } E^{(1)} = E \text{ and } E^{(j+1)} = EE^{(j)} \quad \forall j \geq 1
\end{aligned}$$

2.2 Model Checking

Model checking was briefly introduced in 2.1.3 as a automated verification technique, which can be used to check that a system S satisfies a specification P. The technique

involves representing S as a logic system L_S which captures all possible computations of S, and encoding the property P as a temporal formula φ_P.

The problem of verifying P is then reduced to the problem of checking whether $L_S \vdash \varphi_P$. But we can now build a Kripke model $M_S = (W_S, R_S, \pi)$ such that L_S is sound and complete over (the class of) M_S, so that

$$L_S \vdash \varphi_P \Leftrightarrow M_S \vDash \varphi_P.$$

M_S is the Kripke model representing all possible computations of S, i.e. W_S contains all the possible computational states of the system and the relation R_S represents all temporal transitions in the system.

In the case of a multi-agent system as defined above, encoding S as an interpreted systems of agents will satisfy the equivalence, and we can use CTLK to encode properties of the system to be checked.

2.2.1 Explicit Model Checking

In this section we present a first approach to model checking, the so-called *explicit* approach. Although the explicit model checking algorithm for CTL was in the original article on model checking by Clarke et al.[1], its extension to multi-agent systems and CTLK is due to the work of Lomuscio et al. in [6].

Suppose that we want to check that a multi-agent system Σ satisfies a property P. If IS is an interpreted system representing Σ, and φ the CTLK formula corresponding to P, we need to verify that $(IS, s_0) \vDash \varphi$, for all initial states $s_0 \in I$.

Algorithmically it is more efficient [**?**] to compute the set of global states $[\![\varphi]\!]$ of IS where φ is true, and check that $I \subseteq [\![\varphi]\!]$. Algorithm 2.1 returns $[\![\varphi]\!]$ for any CTLK formula φ.

Notice that this covers all formulae φ, as $\{EX, AF, EU, K_i, E, C\}$ is a minimum set of connectives for CTLK. The respective auxilliary functions are defined below. Notation: for any global states g_0, g_1 of IS we write $g_0 \to g_1$ iff $\exists a \in Act$ such that $\tau(g_0, a) = g_1$ (i.e. there is an run of IS starting with $g_0, g_1, ...$).

The complexity of `SAT` is linear in the size of the model. However, the size of the model (i.e. the number of reachable states) grows exponentially in the number of variables used to describe the system Σ, therefore the explicit approach is not always viable in practice. This is the main difficulty in model checking and it is known as the *state explosion problem*.

Recall that in order to use the algorithm above we also need a way of computing the set of reachable states, which is rarely given explicitely. In most cases all that is given is the set I of initial states of the system. The reachable state space can then be obtained by computing the fixed point of the transition relation function, as shown in the algorithm below:

In the next section we introduce a model checking technique aiming to improve the efficiency of the approach.

```
function SAT(φ)
// returns ⟦φ⟧
if φ = ⊤: return G
if φ = ⊥: return ∅
if φ = p: return π(p)
if φ = ¬φ1: return G∖SAT(φ1)
if φ = φ1 ∧ φ2: return SAT(φ1)∩SAT(φ2)
if φ = EXφ1: return SAT_EX(φ1)
if φ = AFφ1: return SAT_AF(φ1)
if φ = E(φ1Uφ2): return SAT_EU(φ1, φ2)
if φ = Kiφ1: return SAT_K(i, φ1)
if φ = Eφ1: return SAT_E(φ1)
if φ = Cφ1: return SAT_C(φ1)
end
```

Figure 2.1: test

2.2.2 Symbolic Model Checking

Symbolic model checking is an approach to model checking which involves representing sets of states and functions between them as Boolean formulas. The algorithm presented in 2.2.1 is then reduced to a series of operations on Boolean formulas. In this section we go through the process of encoding sets and functions as propositional formulas, and we explain how this encoding facilitates model checking.

Symbolic Representation of Sets of States

We use an example [14] to illustrate the encoding process. Consider the model in Figure 2.2, representing a system with three states labelled s_0, s_1, s_2.

We consider two propositional variables, namely x_1 and x_2. If S is the whole state space (so $S = \{s_0, s_1, s_2\}$), we can represent subsets of S using Boolean formulae, as shown in the following table:

```
function SAT_EX(φ)
// returns ⟦EXφ⟧
  X := {g0 ∈ G | g0 → g1 for some g1 ∈SAT(φ) }
  return X
end

function SAT_AF(φ)
// returns ⟦AFφ⟧
  X := G
  Y := SAT(φ)
  repeat until X = Y:
    X := Y
    Y := Y ∪ {g0 ∈ G | for all g1 with g0 → g1, g1 ∈ Y }
  end
  return Y
end


function SAT_EU(φ1, φ2)
// returns ⟦E(φ1Uφ2)⟧
  W := SAT(φ)
  X := G
  Y := SAT(ψ)
  repeat until X = Y:
    X := Y
    Y := Y ∪ (W ∩ {g0 ∈ G | ∃g1 ∈ Y such that g0 → g1})
  end
  return Y
end
```

```
function SAT_K(i, φ)
// returns [[K_i φ]]
  X := SAT(¬φ)
  Y := {g_0 ∈ G | ∃g_1 ∈ X with g_0 ~_i g_1}
  return G \ Y
end

function SAT_E(φ)
// returns [[E φ]]
  X := SAT(¬φ)
  Y := {g_0 ∈ G | ∃g_1 ∈ X with g_0 ~_i g_1 for all i = 1,...,n}
  return G \ Y
end

function SAT_C(φ)
// returns [[C φ]]
  X := G
  Y := SAT(¬φ)
  repeat until X = Y:
    X := Y
    Y := {g_0 ∈ G | ∃g_1 ∈ X with g_0 ~_i g_1 for all i = 1,...,n}
  end
  return G \ Y
end
```

```
function compute_reach()
// returns the reachable state space G
  X := ∅
  Y := I
  repeat until X = Y
    X := Y
    Y := Y ∪ {g_1 ∈ G | g_0 → g_1 for some g_0 ∈ X}
  end
  return Y
end
```

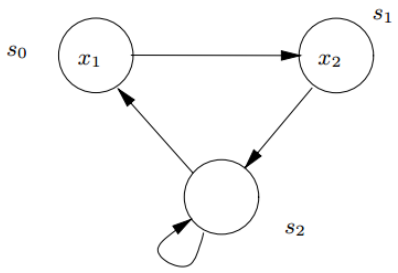


Figure 2.2: A model

set of states	representation by boolean formula
$\varnothing$	$\bot$
$\{s_0\}$	$x_1 \wedge \neg x_2$
$\{s_1\}$	$\neg x_1 \wedge x_2$
$\{s_2\}$	$\neg x_1 \wedge \neg x_2$
$\{s_0, s_1\}$	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$
$\{s_1, s_2\}$	$(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$
$\{s_0, s_2\}$	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge \neg x_2)$
$\{s_0, s_1, s_2\}$	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$

Note that for this representation to be unambiguous, we must ensure that no two states satisfy the same set of Boolean variables. If this is the case, new variables can be added which will be used to differenciate between the ambiguous states.

Symbolic Representation of a Transition Relation

The transition relation $\rightarrow$ of a model is a subset of $S \times S$. Taking two copies of our set of propositional variables, we can then associate a Boolean formula to the transition relation, as follows.

Firstly, notice that in the example above the transition relation $\rightarrow$ of the model is

$$\{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_2)\}.$$

Our set of Boolean variables was $\{x_1, x_2\}$; we now create a copy and use *primed* variables to represent its elements. We get another set $\{x_1', x_2'\}$, which will be used to represent the *next* state in our encoding of the transition relation. Now, for each pair element in the set, we take the conjunction of the Boolean representation of each state in the pair, the first one using the original set of variables, the second the primed set. For example, with (s_0, s_1) is associated the formula $(\neg x_1 \wedge \neg x_2) \wedge (\neg x_1' \wedge \neg x_2')$ (using the Boolean representation for s_0, s_1 derived above – see the table).

As in the representation of sets of states, we represent sets of pairs by taking the conjunction of the representation of each pair element. Thus we compute the

representation of $\rightarrow$ to be $(\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge \neg x_2') \vee (\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2') \vee (x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \vee (\neg x_1 \wedge x_2 \wedge \neg x_1' \wedge \neg x_2')$.

Set Operations with Boolean Functions

If sets of states are represented by Boolean functions, we need a way of applying equivalent operations to the Boolean functions in order to represent their union, intersection, or complement. It turns out that the definition of the symbolic representation of sets of states gives a natural correspondence between the basic set operations and the basic Boolean operations, as follows. Suppose that X, Y are sets represented by Boolean functions f_X and f_Y. Then

$$f_{X \cup Y} = f_X \vee f_Y$$
$$f_{X \cap Y} = f_X \wedge f_Y$$
$$f_{G \setminus X} = \neg f_X.$$

The Functions $\mathrm{pre}_\exists$ and $\mathrm{pre}_\forall$

We have seen that the explicit model checking algorithm relies on two functions on sets of states,

$$\mathrm{pre}_\exists(X) = \{g_1 \in G \mid \exists g_0 \in X \text{ with } g_0 \rightarrow g_1\}$$

and

$$\mathrm{pre}_\forall(X) = \{g_1 \in G \mid \text{ if } g_0 \rightarrow g_1, \text{ then } g_0 \in X\}.$$

We therefore need a way of implementing those using the symbolic representations of sets of states introduced above. First note that

$$\mathrm{pre}_\forall(X) = G \setminus \mathrm{pre}_\exists(G \setminus X),$$

so it is enough to have an algorithm for computing $\mathrm{pre}_\exists(X)$ on a given set X. Let us first recall a basic definition about Boolean functions:

Definition Let f be a Boolean function. The *conditioning* of f on a variable x, written $f|_x$, is the Boolean function obtained by changing x to $\top$ in f, and similarly $f|_{\neg x}$ is obtained from f by setting x to $\bot$. The *existential quantification* of f with respect to x, denoted $\exists x f$, is used to relax the constraint on variable x in f, and it is defined by

$$\exists x f = f|_x \vee f|_{\neg x}.$$

Suppose now that $X \subseteq G$ is a set of states encoded by function f_X, and that the transition relation $\rightarrow$ of the system is encoded by function $f_\rightarrow$. The algorithm has the following steps:

1. Rename variables in f_X to their primed counterparts, and call the resulting function $f_{X'}$.

2. Compute $f_{\rightarrow X'} = f_{X'} \wedge f_{\rightarrow}$.

3. Existentially quantify all primed variables away from $f_{\rightarrow X'}$, i.e. compute $\exists x'_1 ... \exists x'_n f_{\rightarrow X'}$. The resulting function is $f_{\mathrm{pre}_\exists(X)}$, the symbolic representation of $\mathrm{pre}_\exists(X)$.

In the `compute_reach` algorithm above, we use a different operator, which computes the set of states *following* X in the transition relation. This can be encoded in terms of Boolean functions in a very similar way: start with the Boolean encoding of X, conjoin with the encoding of the transition relation, existentially quantify all non-primed variables, and "un-prime" the variables in the resulting function.

The Case of Multi-Agent Systems

Notice that the procedure described above is only valid for global states and does not distinguish between different agents. In the case of multi-agent systems, we can encode the local states for agents using the same method as for states in the general case, making sure to use a different set of variable for each agent. A Boolean representation for a global state in the system will then be the conjunction of the formulas for the local states of which it consists. The separate encoding of the states for each agent will also enable the agent protocols to be encoded independently.

Moreover, in order to successfully implement our model checking algorithm using symbolic expressions, we need a way of representing the agent accessibility relations $\sim_i$. But each of these is a binary relation on states, i.e. a subset of $S \times S$. Hence we can use the exact same method as for the global transition relation.

How does this help us? Well, several techniques exist which allow us to represent these Boolean formulas in a very concise form. This is the topic of the next section.

2.3 Representing Boolean functions

It is important to make the point that the Boolean formulas computed in 2.2.2 can be regarded as Boolean *functions*, i.e. functions $\{0,1\}^n \rightarrow \{0,1\}$ for some n. For example, the formula $x_1 \wedge \neg x_2$ is associated to the function $f(x_1, x_2) = x_1 \wedge \neg x_2$.

In this section we introduce various representations of Boolean functions using directed acyclic graphs (DAG).

2.3.1 Ordered Binary Decision Diagrams

One the most basic ways of representing a Boolean function is by using a *binary decision tree* (BDT). A BDT is a binary tree where we label non-terminal nodes with Boolean variables $x_1, x_2, ...$ and terminal nodes with the values 0 and 1. Each non-terminal node has two outgoing edges, one solid and one dashed (one for each value

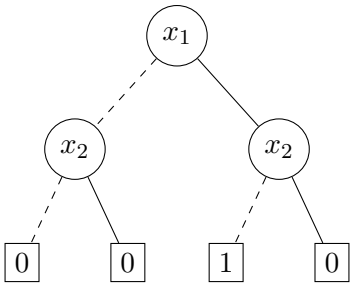


Figure 2.3: BDT for $f(x_1, x_2) = x_1 \wedge \neg x_2$ under the ordering $[x_1, x_2]$

of the variable the node is labelled with). An example is shown in Figure 2.3. Note that the initial ordering of the variables affects the resulting BDT; for convenience, we write $[x_1, x_2, ..., x_n]$ to denote the order $x_1 < x_2 < ... < x_n$.

BDTs are a relatively inefficient way of storing Boolean formulas, since a BDT for a formula with n variables has $2^{n+1} - 1$ nodes. A binary decision diagram (BDD) is a directed acyclic graph, similar to a BDT but allowing nodes to have more than one incoming edge. A BDD is called an *ordered binary decision diagram* (OBDD) if every path from the root to a terminal node, variables appear in the same order (although not all variables have to occur in the path). An extremely useful property of OBDDs is that they can be *reduced* to a ROBDD which is, in most cases, a far smaller data structure than the BDT for the same function. There are three steps in the reduction of BDTs to OBDDs:

1. Removal of duplicate terminals: we merge all the 0-nodes and 1-nodes into two unique terminal nodes;

2. Removal of redundant tests: if both outgoing edges of a node n point to the same node m, we remove n from the graph, sending all its incoming edges directly to m;

3. Removal of duplicate non-terminals: we merge any two subtrees with identical BDD structure.

Steps (2) and (3) are repeatedly applied until no further reduction is possible, and the resulting diagram is said to be a reduced ordered binary decision diagram. Figure 2.4 illustrates this algorithm with an example. The following key result makes the use of OBDDs viable in practice:

Theorem 2.3.1. *[20] If f is a Boolean function over the variables $x_1, ..., x_n$, then f has a unique ROBDD for each order of the variables $x_1, ..., x_n$.*

The immediate consequence of Theorem 2.3.1 is that one can easily compare two Boolean functions by comparing their respective ROBDDs (provided both ROBDDs have the same variable order).

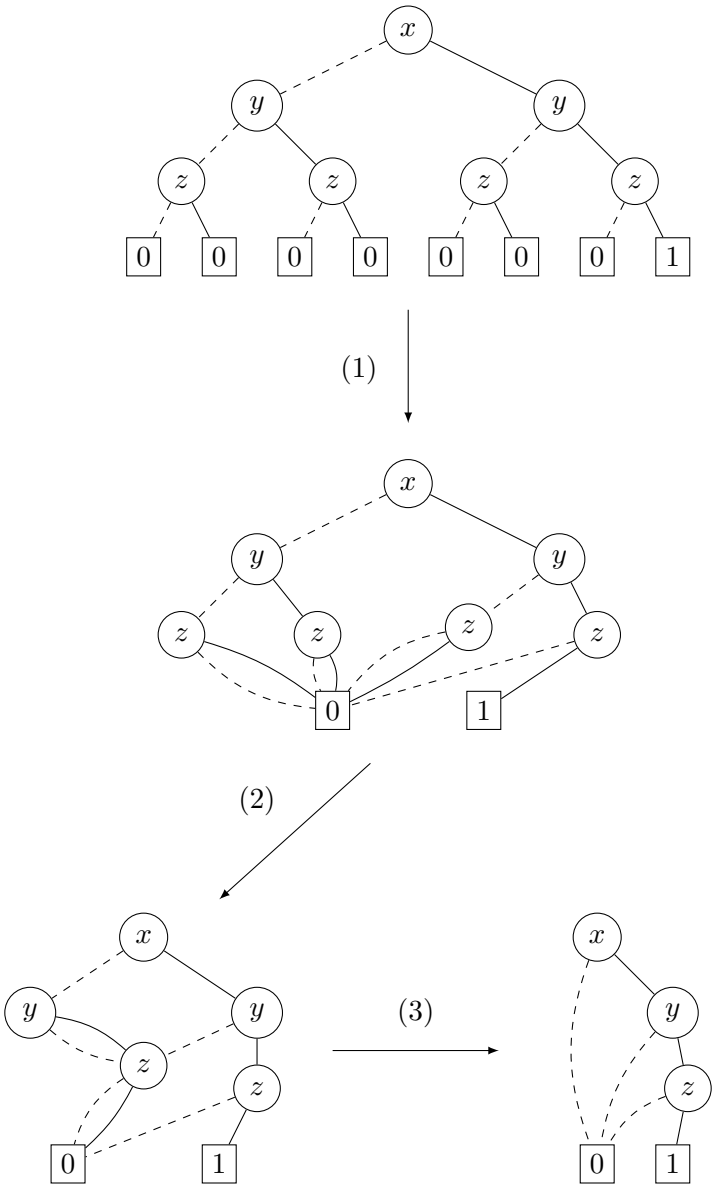


Figure 2.4: Reduction to the OBDD for function $f(x, y, z) = x \wedge y \wedge z$ with variable order $[x, y, z]$. In this case, applying Steps (2) and (3) once is sufficient as the resulting diagram is reduced.

Another important observation to make is that ROBDDs resulting from two different variable orders may present a *significant* difference in size, and therefore a large amount of work has been done in the search for suitable variable orders.

ROBDDs help us to manipulate Boolean functions with a high number of variables, allowing us to use our model checking algorithm (2.2.1) on systems with much larger state-spaces, which has led researchers in the past 15 years to explore various graph-based representations of Boolean functions.
Note to the reader: comforming to the generally accepted denomination, throughout this report we often drop the 'RO' and refer to ROBDDs as BDDs.

2.3.2 Project Directions

The initial objective of this project was to investigate new representations of Boolean functions suitable for application to symbolic model checking, seeking a potential improvement on BDDs (which constitute so far the "industry standard").

In computer science, the field of *knowledge compilation* is concerned with finding compact and efficient representations of *propositional knowledge bases*, which correspond to sets of propositional formulae. Such a representation is refered to as a *target compilation language*, of which BDTs and OBDDs are examples.

In order for an target compilation language to be suitable for knowledge compilation, it must be *succinct* (i.e. representations must be small), and it must support a number of important *queries* in polynomial time. OBDDs, for instance, satisfy these two properties (with succinctness remaining highly dependent on the variable order).

Symbolic model checking also recognises the need for succinct representations of propositional formulae, however an important difference must be noted: the model checking algorithm consists in a large series of *operations* on Boolean functions, whereas knowledge compilation focuses on encoding a propositional knowledge base, in order to *query* it more efficiently later on.

In 2002, A. Darwiche and P. Marquis published [11], a comparative analysis of most existing DAG-based target compilation languages in terms of their succintness and the polytime operations they support. They show that all of these languages are subsets of a broad language called *negation normal form* (NNF).

Definition A sentence in *NNF* is a rooted directed acyclic graph where each leaf node is labelled with $\top$, $\bot$, X or $\neg X$ for some propositional variable X, and each internal node is labelled with $\wedge$ or $\vee$ and can have arbitrarily many children.

Observe that OBDDs are NNF sentences. Figure 2.5 shows an OBDD and the corresponding NNF sentence.

This "knowledge compilation map" was the starting point for this project. Taking into account the various criteria for suitable representations described above, we searched the literature aiming to find a good candidate to replace BDDs.

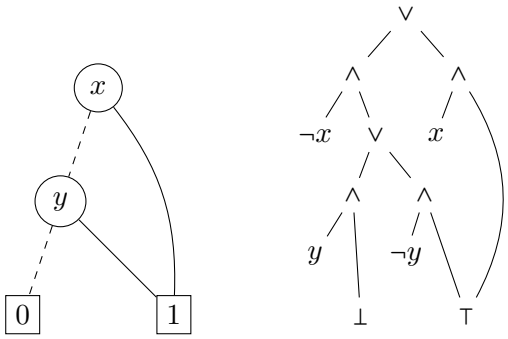


Figure 2.5: An OBDD (left) and the corresponding NNF sentence (right). Although the former seems much more compact, the difference in size is only linear and both representations are essentially the same.

After considering a number of different representations including tree-of-BDDs [12], BDD-trees [13], and several other subsets of NNF described in [11], we opted for a relatively novel target compilation language called *sentential decision diagram* (SDD).

SDDs are a subset of NNF, and they satisfy essential properties such as succintness and polynomial-time Boolean operations. Also, they had not yet been experimented with in the context of symbolic model checking, for state-space representation. Moreover, some of the experimental results presented in [7] and [9] demonstrate that SDDs can lead to a significant improvement on BDDs in terms of computation time and memory usage.

In the next section we present SDDs in full details, delaying our own experimental results to the following chapters.

2.4 Sentential Decision Diagrams

Most of the content in that section is taken from the work of A. Darwiche in [7], the first paper written on SDDs.

2.4.1 Preliminaries

To define SDDs formally we must start with some preliminary definitions and results related to Boolean functions.

Definition We say that a function f *essentially depends* on a variable x iff $f|_x \neq f|_{\neg x}$, and we write $f(X)$ if f essentially depends on variables in X only. If $f(Z)$ is a function, we also write $f(X, Y)$ when X, Y are sets forming a partition of Z.

The following definition is the basis for the construction of SDDs:

Definition (Decompositions and partitions) An *(X,Y)-decomposition* of a function $f(X,Y)$ is a set of pairs $\{(p_1,s_1),...,(p_n,s_n)\}$ such that

$$f = (p_1(X) \wedge s_1(Y)) \vee ... \vee (p_n(X) \wedge s_n(Y)).$$

The decomposition is said to be *strongly deterministic* on X if $p_i(X) \wedge p_j(X) = \bot$ for $i \neq j$. In this case, each ordered pair (p_i, s_i) in the decomposition is called an *element*, each p_i a *prime* and each s_i a *sub*.

Let $\alpha = \{(p_1,s_1),...,(p_n,s_n)\}$ be an (X,Y)-decomposition, and suppose α is strongly deterministic on X. Then α is called an *X-partition* iff its primes form a partition (i.e primes are pairwise mutually exclusive, each prime is consistent, and the disjunction of all primes is valid). We say that α is *compressed* if $s_i \neq s_j$ for $i \neq j$.

Example Let $f(x,y,z) = (x \wedge y) \vee (x \wedge z)$. Then $\alpha = \{(x, y \vee z)\}$ is an $(\{x\},\{y,z\})$-decomposition of f which is strongly deterministic (as there is only one prime). It is however not an $\{x\}$-partition, but $\beta = \{(x, y \vee z), (\neg x, \bot)\}$ is, since $x, \neg x$ form a partition. Note that β is compressed.

Remark that in an X-partition $\bot$ can never be prime, and if $\top$ is prime then it is the only prime. Moreover primes determine subs, so two X-partitions are different iff they contain distinct primes.

Theorem 2.4.1. *Let $\circ$ be a Boolean operator and let $\{(p_1,s_1),...,(p_n,s_n)\}$ and $\{(q_1,r_1),...,(q_m,r_m)\}$ be X-partitions of Boolean functions $f(X,Y)$ and $g(X,Y)$ respectively. Then*

$$\{(p_i \wedge q_j, s_i \circ r_j) \mid p_i \wedge q_j \neq \bot\}$$

is an X-partition of $f \circ g$.

Proof. Since $p_1,...,p_n$ and $q_1,...,q_m$ are partitions, the $(p_i \wedge q_j)$ also form a partition for $i = 1,...,n$, $j = 1,...,m$ and $p_i \wedge q_j \neq \bot$. So the given decomposition is an X-partition of some function.

Consider now an instantiation $\mathbf{z}$ of variables $X \cup Y$, so $\mathbf{z}$ consists of an instantiation $\mathbf{x}$ of X, and an instantiation $\mathbf{y}$ of Y. Then, since $\{p_k\}$ and $\{q_l\}$ are partitions, there must exist a unique i and a unique j such that $\mathbf{x} \models p_i$ and $\mathbf{x} \models q_j$, i.e $p_i(\mathbf{x}) = q_j(\mathbf{x}) = \top$. Then $f(\mathbf{x},\mathbf{y}) = s_i(\mathbf{y})$ and $g(\mathbf{x},\mathbf{y}) = r_j(\mathbf{y})$, so $(f \circ g)(\mathbf{x},\mathbf{y}) = s_i(\mathbf{y}) \circ r_j(\mathbf{y})$.

Evaluating the given decomposition at instantiation $\mathbf{z}$ also gives $s_i(\mathbf{y}) \circ r_j(\mathbf{y})$, hence result. □

As we see later on, an important consequence of Theorem 2.4.1 is the polynomial time operations available on SDDs. Canonicity of SDDs is due to the following result:

Theorem 2.4.2. *A function $f(X,Y)$ has exaclty one compressed X-partition.*

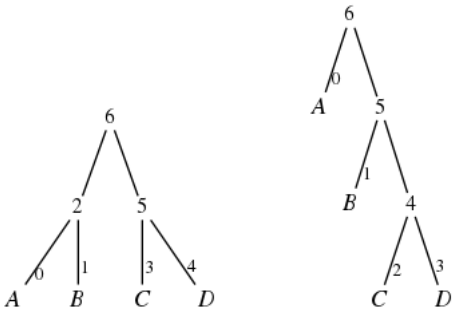


Figure 2.6: Two vtrees for $X = \{A, B, C, D\}$, both inducing the order $[A, B, C, D]$.

Proof. Let $\mathbf{x}_1, ..., \mathbf{x}_k$ be *all* instantiations of variables X. Then $\{(\mathbf{x}_1, f|_{\mathbf{x}_1}), ..., (\mathbf{x}_k, f|_{\mathbf{x}_k})\}$ is an X-partition of f. Let $s_1, ..., s_n$ be the distinct elements of $f|_{\mathbf{x}_1}, ..., f|_{\mathbf{x}_k}$, and for each s_i define $p_i = \bigvee_{f|_{\mathbf{x}_j} = s_i} \mathbf{x}_j$. Then $\alpha = \{(p_1, s_1), ..., (p_n, s_n)\}$ is a compressed X-partition of f.

Suppose now that $\beta = \{(q_1, r_1), ..., (q_m, r_m)\}$ is another compressed X-partition of f. Then α and β must have different partitions, i.e. $\{p_i\} \neq \{q_j\}$. Since they are both *partitions*, there must exist a prime p_i of α which overlaps with two different primes q_j and q_k of β, i.e. there exist distinct instantiations $\mathbf{x}$, $\mathbf{x}$' of X such that $\mathbf{x} \vDash p_i \wedge q_j$ and $\mathbf{x}' \vDash p_i \wedge q_k$. Then we have $f|_{\mathbf{x}} = \alpha_{\mathbf{x}} = s_i = r_j = \beta_{\mathbf{x}}$, and $f|_{\mathbf{x}'} = \alpha_{\mathbf{x}'} = s_i = r_k = \beta_{\mathbf{x}'}$. So $r_j = r_k$, a contradiction as β is compressed. □

Vtrees

Vtrees (for "variable trees") are to SDDs what variable orders are to BDDs. A vtree completely determines the structure of an SDD, so they are crucial to the viability of SDDs in practice.

Definition A *vtree* for variables X is a full binary tree whose leaves are in one-to-one correspondence with the variables in X. We will often not distinguish between a vtree node v and the subtree rooted at v, and the left and right children of a node v will be denoted v^l and v^r, respectively.

Each vtree *induces* a variable order, namely the order in which leaves are visited during a left-right traversal of the vtree. Note that a vtree on a set X is stronger than a total order of the variables in X, as two different vtrees may induce the same order; an example is shown in Figure 2.6.

2.4.2 Definition and Construction

The construction of an SDD for a Boolean function f with respect to a vtree v is done by a recursive algorithm on the children nodes of v.

Let v be the vtree on the left of Figure 2.6, and let

$$f(A, B, C, D) = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D).$$

The decomposition of f at the vtree node v goes as follows: we split the variables in X into two subsets by separating variables in v^l from those in v^r: we obtain $\{A, B\}$ and $\{C, D\}$. We compute the unique compressed $\{A, B\}$-partition α of f:

$$\begin{aligned} f(A, B, C, D) &= (A \wedge B) \vee (B \wedge C) \vee (C \wedge D) \\ &= ((A \wedge B) \wedge \top) \vee ((\neg A \wedge B) \wedge C) \vee (\neg B \wedge (C \wedge D)). \end{aligned}$$

So $\alpha = \{(\neg B, C \wedge D), (\neg A \wedge B, C), (A \wedge B, \top)\}$. The next step is to decompose the primes of α with respect to v^l, and its subs with respect to v^r. The vtree node v^l splits the set $\{A, B\}$ into two subsets $\{A\}$ and $\{B\}$, so the non-trivial primes of f, namely $A \wedge B$ and $\neg A \wedge B$ have SDDs $\{(A, B), (\neg A, \bot)\}$ and $\{(\neg A, B), (A, \bot)\}$, respectively. Similarly, we compute a $\{C\}$-partition of the only non-trivial sub of f, namely $C \wedge D$, and get $\{(C, D), (\neg C, \bot)\}$, which is also an SDD since all its elements are constants or literals. The resulting SDD for f is

$$\{(\neg B, \{(C, D), (\neg C, \bot)\}), (\{(\neg A, B), (A, \bot)\}, C), (\{(A, B), (\neg A, \bot)\}, \top)\},$$

which is very hard to read. But SDDs are generally written in their graphical representation, as shown below.

Graphical Representation of SDDs

The SDD constructed for function f is represented on the left in Figure 2.7. On the right is the SDD for f constructed with respect to the vtree on the right in Figure 2.6.

A decomposition is represented by a circle with outgoing edges pointing to its elements, and an element is represented by a pair of boxes where the left box represents the prime and the right box represents the sub. If one of them is another decomposition, we leave the box empty and draw an edge pointing to the circle node representing it.

The next two definitions formally define the syntax and semantics of SDDs.

Definition (Syntax) Let v be a vtree. α is an SDD that respects v iff:

- $\alpha = \top$ or $\alpha = \bot$
- $\alpha = X$ or $\alpha = \neg X$, and v is a leaf with variable X
- v is an internal node (i.e. it has children), and α is a partition $\{(p_1, s_1), ..., (p_n, s_n)\}$ such that for all i, p_i is an SDD that respects v^l and s_i is an SDD that respects v^r.

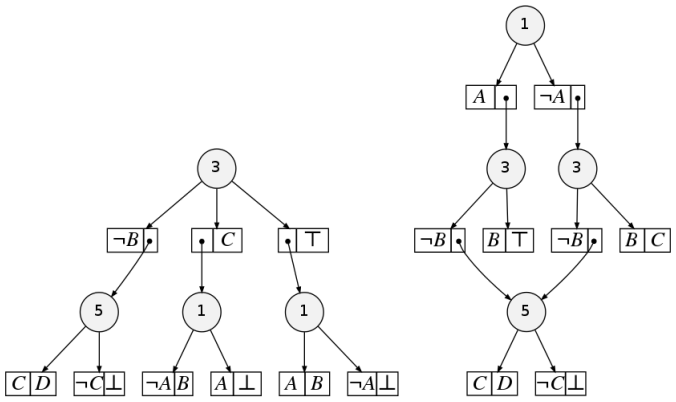


Figure 2.7: SDDs for $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ corresponding to the vtrees in Figure 2.6. Notice that identical SDD nodes have been merged.

In the first two cases we say that α is *terminal*, and in the third case α is called a *decomposition*. For SDDs α and β, we write $\alpha = \beta$ iff they are *syntactically equal*.

Definition (Semantics) Let α be an SDD. We use $\langle . \rangle$ to denote the mapping from SDDs to Boolean functions, and we define it inductively as follows:

- $\langle \top \rangle = \top$ and $\langle \bot \rangle = \bot$
- $\langle X \rangle = X$ and $\langle \neg X \rangle = \neg X$, for all variables X
- $\langle \{(p_1, s_1), ..., (p_n, s_n)\} \rangle = \bigvee_{i=1}^{n} \langle p_i \rangle \wedge \langle s_i \rangle$

We say two SDDs α and β are *equivalent* (written $\alpha \equiv \beta$) if $\langle \alpha \rangle = \langle \beta \rangle$.

2.4.3 Canonicity and Operations

It is obvious that if SDDs α and β are equal, then they are equivalent. We would however like to impose conditions on the construction of α and β so that $\alpha \equiv \beta \Rightarrow \alpha = \beta$, which would make SDDs a *canonical* representation, a crucial property. We begin with a few definitions and lemmas.

Definition Let f be a non-trivial Boolean function. We say f *essentially depends* on vtree node v if v is a deepest node that includes all variables that f essentially depends on.

Lemma 2.4.3. *A non-trivial function essentially depends on exactly one vtree node.*

Definition An SDD is *compressed* iff all its decompositions are compressed. It is *trimmed* iff it does not have decompositions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$ for some SDD α.

These two properties are very accessible. An SDD is compressed as long as all X-partitions used during its construction are compressed, and it can be trimmed by traversing it bottom-up and replacing decompositions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$ by α. Theorem 2.4.5 below shows that they are in fact sufficient for the representation to be canonical. To prove it we first need another lemma.

Lemma 2.4.4. *Suppose α is a non-trivial, compressed and trimmed SDD. Then α respects a unique vtree node v, which is the unique node that the Boolean function $f = \langle\alpha\rangle$ essentially depends on.*

Theorem 2.4.5. *Let α and β be compressed and trimmed SDDs. Then*

$$\alpha = \beta \Leftrightarrow \alpha \equiv \beta.$$

Proof. ($\Rightarrow$) is clear. For ($\Leftarrow$), suppose that $\alpha \equiv \beta$ and let $f = \langle\alpha\rangle = \langle\beta\rangle$. If f is constant, then α and β are trivial SDDs, therefore they are equal. Suppose now that f is non-trivial, and let v be the vtree node that f essentially depends on (it is unique by Lemma 2.4.3). Then by Lemma 2.4.4, α and β respect v. We continue the proof by structural induction on v.

If v is a leaf, then α and β are terminals. But f is non-trivial so α and β are equivalent literals, and so they must be equal. Suppose now that v is internal, and that the theorem holds for v^l and v^r. Let X be the variables in v^l and Y be the variables in v^r. Write $\alpha = \{(p_1, s_1), ..., (p_n, s_n)\}$ and $\beta = \{(q_1, r_1), ..., (q_m, r_m)\}$, where the p_i, q_j are SDDs with respect to v^l and the s_i, r_j are SDDs with respect to v^r. Then $\{(\langle p_1\rangle, \langle s_1\rangle), ..., (\langle p_n\rangle, \langle s_n\rangle)\}$ and $\{(\langle q_1\rangle, \langle r_1\rangle), ..., (\langle q_n\rangle, \langle r_n\rangle)\}$ are X-partitions of f, and they are compressed since α and β are compressed SDDs. So by Theorem 2.4.2, they are the same. So $n = m$, and for all i we have $\langle p_i\rangle = \langle q_i\rangle$ and $\langle s_i\rangle = \langle r_i\rangle$, possibly after reordering. Then by definition $p_i \equiv q_i$ and $s_i \equiv r_i$, which by induction implies that $p_i = q_i$ and $s_i = r_i$. So $\alpha = \beta$. □

Operations on SDDs

We start right away by giving the pseudo-code for the `apply` algorithm on SDDs, which combines two SDDs α and β using a Boolean operator $\circ$, provided they respect the same vtree node.

If α and β are compressed, then this algorithm returns a compressed SDD for $\alpha \circ \beta$. Theorem 2.4.1 ensures that `apply(`α`,` β`,` $\circ$`)` is in fact an SDD, while the if condition on line 13 checks that its subs are distinct, thereby making it a compressed SDD. The use of the `cache` in the pseudo-code emphasises the fact that any implementation of `apply` could be significantly improved by keeping the set of computed SDDs in a cache in memory.

If lines 13 - 16 were replaced by `add (`p`,` s`) to` γ (i.e. if we omitted the code checking that the obtained SDD is compressed), the time complexity of `apply` on SDDs α and β would be $O(|\alpha||\beta|)$ and all Boolean operations could be done in

```
 1 function apply(α, β, ∘)
 2  if α and β are constants or literals
 3    return α ∘ β
 4  else if cache(α, β, ∘) != null
 5    return cache(α, β, ∘)
 6  else
 7    γ = {}
 8    for all elements (p_i, s_i) in α
 9      for all elements (q_j, r_j) in β
10        p = apply(p_i, q_j, ∧)
11        if p != ⊥
12          s = apply(s_i, r_j, ∘)
13          if ∄ element (q, s) in γ
14            add (p, s) to γ
15          else
16            add (apply(p, q, ∨), s) to γ
17          end if
18        end if
19      end for
20    end for
21    return cache(α, β, ∘) = γ
22  end if
23 end function
```

polynomial time. The additional recursive call on line 16 makes the algorithm harder to analyze, considering `p` and `q` are not part of either α or β. In practice however, this additional call makes `apply` much more efficient since it is then dealing with compressed SDDs, easier to manipulate [7].

The other SDD operation which will be essential for symbolic model checking is `condition`, which computes the conditioning of an SDD on a literal. Although [7] doesn't supply pseudo-code for `condition`, it can be done in polynomial time. Let α be an SDD for function f, and let $\mathbf{x}$ be a literal; The SDD for function $f|_{\mathbf{x}}$, i.e. the result of `condition(x,` α`)` can be obtained by applying the following operations, bottom-up:

- For each element $(\beta, \mathbf{x})$ of α, we replace it with $(\beta, \top)$ if no other element has $\top$ as a sub, or by $(\texttt{apply}(\beta,\ \gamma,\ \vee), \top)$ if $(\gamma, \top)$ was another element with sub $\top$. In the latter case we also remove the element $(\gamma, \top)$ and direct all its incoming edges to the newly created node. (Note that γ would have been unique, since α is compressed).
- For each element $(\beta, \neg\mathbf{x})$ of α we do the same as the above, replacing $\top$ with $\bot$.
- For each element $(\mathbf{x}, \beta)$ of α, we replace the partition of which it is an element with β, essentially "removing one level" of this particular SDD.
- We remove each element $(\neg\mathbf{x}, \beta)$ of α. Note that the remaining decomposition will still be an X-partition for some set X, since the previous steps will have been applied on the rest of the elements.

Observe that although this can be done in polynomial time, the `condition` operation on SDD is slightly more time-consuming than the equivalent on BDDs, which simply involves removing each node labelled $(\neg)\mathbf{x}$, redirecting its incoming edges to the corresponding child.

2.4.4 OBDDs are SDDs

To end the background section of this report, we would like to focus on the fact that *OBDDs are SDDs*, i.e. if we regard them as subsets of NNF, then OBDD $\subseteq$ SDD. We first give a simple explanation for why this is the case.

Consider an SDD α respecting a vtree v such that every left child is a leaf. Such a vtree is said to be *right-linear*, and every decomposition in α will be of the form $\{(A, \beta_1), (\neg A, \beta_2)\}$ for some variable A and SDDs β_1, β_2. But this is the same decomposition as that occuring in a BDD at the node labelled with A.

Suppose now that B is a BDD for function f, and that v is the *only* right-linear vtree inducing the variable order for B. Then the SDD for f respecting v is *syntactically equal* to B, when both are regarded as NNF sentences. Hence there is a

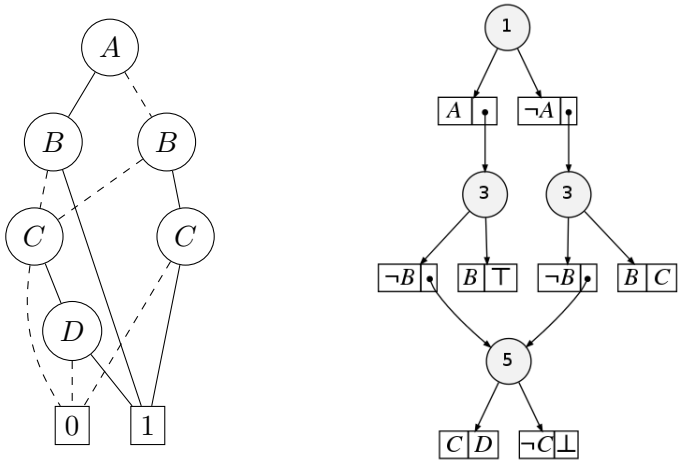


Figure 2.8: The BDD for $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ with variable order $[A, B, C, D]$ (left), and the SDD respecting the right-linear vtree inducing the same order (right). Note that this is the vtree depicted on the right of Figure 2.6.

one-to-one correspondence between BDDs and SDDs respecting right-linear vtrees. Figure 2.8 illustrate this with an example.

A very general consequence of this is that any application of BDDs in computer science could be implemented using SDDs instead, as it would retain the attractive properties of BDDs while benefiting from the potential reduction in size that SDDs present. In the more specific case of this project, we will see in a later chapter that this correspondence was extremely useful in helping us compare the efficiency of SDDs with that of BDDs in model checking: by restricting our SDD implementation to right-linear vtrees, we were able to compare both programs more accurately by taking into account the overhead due to the difference in *implementation* (as opposed to a difference in size between the two structures – we knew these were the same!).

Chapter 3

A First Model Checker Based on SDDs

For the first time, SDDs are being used for symbolic model checking. Implementing a model checker based on SDDs therefore required some prior reflection. In this chapter, we start by presenting the existing code base and libraries which our implementation relies upon (Sections 3.1 & 3.2). Using these "tools", we consider our options, and devise a theoretical approach which will guide the implementation of our model checker, as well as our evaluation strategy in the next chapter (3.3). Finally, we describe the design, algorithms, and challenges of our implementation (3.4 to 3.6).

3.1 Preliminary I: The SDD Package

3.1.1 Overview

The SDD Package [31] is a C library which can be used to create and manipulate SDDs. It was developed at UCLA by the Automated Reasoning group, who first introduced SDDs. Its current version is 1.1.1.

The SDD Package API contains most of the functions required for the use of SDDs in model checking. This includes basic manipulations such as conjunction, disjunction, and negation of Boolean functions represented by SDDs, conditioning a function on a literal, and quantifying out variables (the SDD equivalent of $\exists$ and $\forall$). Additionally, the API makes possible a number of operations on vtrees, including *rotating* and *swapping* (these are crucial for navigating the space of vtrees in the context of SDD minimisation, see below for details).

The *SDD manager* is the focal point for all the SDD operations in the program. It is there to ensure that all SDDs in the program have been built with respect to the same vtree, and it handles SDD conversions in the case of a vtree modification. It also gives the user access to a number of statistics, helpful for tracking a program's

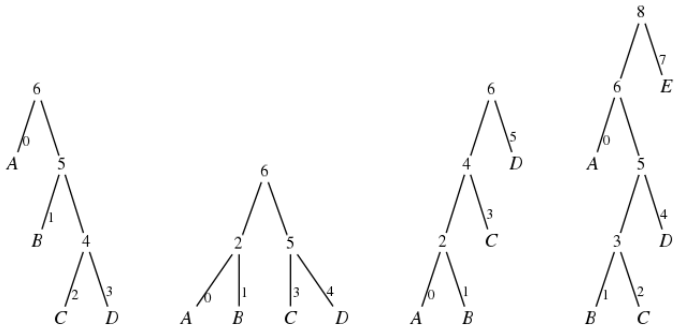


Figure 3.1: The four standard vtrees, in order: right-linear, balanced, left-linear, and vertical.

memory usage or the size of some SDD nodes.

3.1.2 Standard Vtrees

Four different "classes" of vtrees are pre-implemented in the SDD package, i.e. can be created in one function call, provided a variable order is supplied. They are:

- right-linear: all left children are leaves
- left-linear: all right children are leaves
- balanced: the vtree is a balanced binary tree, i.e. both children of a node have the same number of leaves (if total number is even)
- vertical: every node has a leaf child, which is alternatively the right child or the left child

Throughout this report we refer to these as the *standard* vtrees. An example of each is given in Figure 3.1.

3.1.3 Dynamic SDD Minimisation

One particularly important feature of the SDD Package is *dynamic SDD minimisation*. When enabled, this feature automatically attempts to minimise the size of a manager's SDD nodes by searching for a better vtree. Unfortunately this is not always a more efficient solution as searching the space of vtrees is a very lengthy process: there are $\frac{(2n-2)!}{(n-1)!}$ distinct vtrees over n variables!

An vtree search algorithm was proposed in [9] for more efficient dynamic minimisation of SDDs. This algorithm relies on three standard binary tree operations: left-rotation, right-rotation, and swap; these operations can be applied on any node of a vtree, and they are sufficient for navigating the space of all vtrees [15]. They are

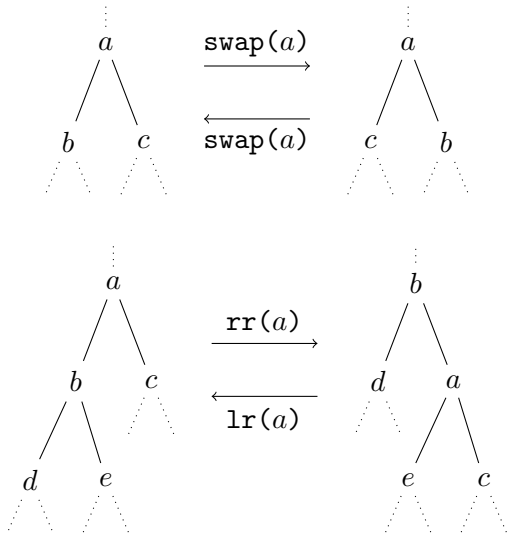
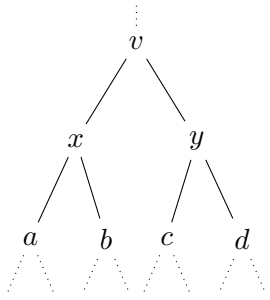


Figure 3.2: The main vtree operations and their effect.

implemented in the SDD Package as `swap_vnode()`, `lr_vnode()`, and `rr_vnode()`. Figure 3.2 shows in what way these operations affect vtrees.

We now give an informal description of the algorithm in [9] (the one implemented in the SDD Package). We start with a vtree node v (typically, the current vtree of the manager). The algorithm first makes two recursive calls on the children of v, ensuring that their structure is optimal. Suppose that the vtree obtained is the one represented below:



Then, the algorithm considers two vtree *fragments* of v, the *l-vtree* and the *r-vtree*. The l-vtree is the binary tree with leaves a, b and y, and the r-vtree is the binary tree with leaves x, c and d. The algorithm attempts to compute the 24 distinct vtree fragments obtained by applying rotations and swaps to the l- and r-vtrees, and then selects the one leading to the most effective reduction in size for the SDD nodes depending on v.

For each of the three vtree operations, users of the SDD package can set limits

on the time needed to compute them, and the increase in size that they induce; this means that not all 24 vtree fragments above are necessarily computed.

3.1.4 Comparison with CUDD

CUDD is the C++ BDD library used by MCMAS (see below) for BDD manipulation, and it is therefore our reference for all practical comparisons between SDDs and BDDs.

Here we simply give an account of the differences between CUDD and the SDD package, in order for our comparisons to be fair: we are interested in the relative efficiency of the data structures, not the packages, and therefore it is important that we take into account any differences in design and implementation.

Fortunately, the libraries have a very similar flavour. For instance both are organised around a manager handling all internal operations. A few differences are worth mentioning:

- Both libraries have an automated garbage collection feature, and both are based on node reference counts. However the SDD Package requires users to manually increment and decrement the reference counts of each SDD node they create, whereas in CUDD this is optional (MCMAS doesn't use it).
- In the SDD Package the automated garbage collection and dynamic minimisation features are not independent, i.e. one cannot be enabled without the other being too. This means that when experimenting with SDDs without dynamic minimization, we find that a lot of unused nodes are kept in memory, sometimes preventing the program from generating any more SDDs in an efficient way (once the memory has run out, which happens relatively often in model checking when no garbage collection takes place). This is not the case in CUDD.
- An important difference in implementation is in the way the libraries handle existential quantification (the function $\exists$). We devote an entire subsection to this issue (3.4.2).

3.2 Preliminary II: MCMAS

MCMAS (Model Checker for Multi-Agent Systems) is a BDD-based model checker written in C++, which was developed, and is currently maintained at Imperial College in the Verification of Autonomous Systems group. It was specifically designed for the verification of multi-agent systems, and users can write system descriptions in a language called ISPL (Interpreted System Programming Language), whose syntax is very much inspired from the definition of interpreted systems (2.1.2). MCMAS is able to verify CTLK formulae, and a recent version also incorporates ATL (the

Alternating-Time Temporal Logic [19], not discussed here), however it does not support the verification of LTL(K) formulae.

In order to better understand the implementation of the model checker, the reader is strongly advised to start by getting an overview of the main components in an ISPL file. For this purpose we supply a full example of a simple interpreted system encoded in ISPL: the *Bit Transmission Problem*, available in Appendix A. ISPL is a fairly straightforward language, we therefore will not spend any more time describing it here, but should anything remain unclear, the reader is refered the MCMAS User Manual [29].

The model checker built for this project is (for a large part) based on the MCMAS code base. In this section we outline some of the MCMAS internal implementation details in order to help the reader understand the steps undertaken when replacing BDDs with SDDs. Moreover, MCMAS will be the basis of our BDDs/SDDs comparisons in the next chapter so it is important for the reader to understand the different configuration options.

3.2.1 Important Classes and Methods

The ISPL parser in MCMAS creates a model of the system using the following important classes:

- `basic_agent`: an agent in the system, consisting of a protocol, an evolution, a set of variables and a set of actions
- `evolution_line` and `protocol_line`: a line in the evolution or in the protocol of an agent, consisting of a Boolean expression and an assignment or (respectively) an action
- `bool_expression` and `assignment`
- `variable`, `basic_type`, `int_value`, `enum_value`, `bool_value`, `rangedint`, `atomic_proposition`, `laction`: agent variables, their types and their values
- `modal_formula`: a CTLK formula to be checked in the model.

Throughout the model checking procedure the *BDD parameters* are carried by the program and passed as argument to the various methods. They are encapsulated in a structure (`struct bdd_parameters`) and contain all the important data required by the algorithm, in particular:

- three vectors `v`, `pv`, and `a`, containing the Boolean variables used in the model for encoding states, next states, and actions respectively
- a vector `vRT`, containing for each agent the BDD for its transition relation
- the BDD `in_st` representing the set of initial states, and (once computed) the BDD `reach` for the reachable state space

- a pointer to the BDD cache
- a vector `is_formulae` containing the `modal_formula` objects for each CTLK formula to check

MCMAS is (quite literally) an implementation of the model checking algorithm described in 2.2.1. However, as a symbolic model checker it also contains procedures for encoding protocols, transition relations, sets of states and actions, etc., and in fact these constitute a very large part of the code. We list the main steps in the execution of the program:

1. Parse ISPL file, allocate variables and set up CUDD (see 3.2.2 below)
2. Compute the BDD for the global transition relation
3. Compute the BDD for the reachable state space
4. For each modal formula φ, compute `SAT(`φ`)` and check that it is a subset of I
5. Output result and free memory

3.2.2 Variable Allocation

Variable allocation is the process of allocating manager variables (created with the manager) to agents and determining the various sets of variables needed for symbolic representation within each agent: state variables, primed state variables (a copy of the state variables representing the *next* states), and action variables.

For each agent, MCMAS first computes the number of variables needed in each of the aforementioned sets by calls to the functions `state_BDD_length()` and `action_BDD_length()`.

The `basic_agent` functions `allocate_BDD_2_variables()` and `allocate_BDD_2_actions()` are then used for variable allocation: they assign a portion of both `v` and `a` to each agent, giving them start and end *indices*. Note that this forces all of an agent's state variables (and similarly, action variables) to be next to each other in `v` (and similarly, `a`).

It may seem confusing that variable allocation happens before the user has been able to select a particular variable order, but in fact no actual variables have yet been allocated, only their position in the arrays. The user's choice will then affect the way the *manager*'s variables are dispatched across `a`, `v`, and `pv`.

In MCMAS, the user can choose between four *standard* different variable orders. We take the time to present them here, not only because of the impact that this choice has on the overall performance, but also for reference in the future chapters (where we compare these orders with various SDD vtrees).

3.2.3 Standard Variable Orders

Suppose MCMAS is running on an example requiring n state variables denoted $x_1, ..., x_n$, and m action variables denoted $a_1, ..., a_m$. By definition there are n primed state variables (the next state is a state, so it can be represented with the n state variables), these are denoted $x'_1, ..., x'_n$. Suppose also that there are k agents, and that for each i, agent i has been allocated variables $x_{i_1}, ..., x_{i_{n_i}}$, $x'_{i_1}, ..., x'_{i_{n_i}}$ and $a_{j_1}, ..., a_{j_{m_i}}$, for some $n_i, m_i \in \mathbb{N}$ and where $i_1, ..., i_{n_i}$ and $j_i, ..., j_{m_i}$ are sequences of consecutive integers.

The manager will then have $2n + m$ variables in total, and the following are the possible four ordering options with respect to which BDDs will be constructed throughout the process. In options 2 to 4, variables are ordered using the agent order, so that we get an ordering of the form

$$[\text{variables for agent } 1, \text{variables for agent } 2, ..., \text{variables for agent } k],$$

and the difference lies in the way variables are ordered within each agent set.

- Ordering option 1 (no interest in agent allocation):

$$[x_1, ..., x_n, x'_1, ..., x'_n, a_1, ..., a_m]$$

- Ordering option 2 (variables for agent i):

$$[x_{i_1}, x'_{i_1}, x_{i_2}, x'_{i_2}, ..., x_{i_{n_i}}, x'_{i_{n_1}}, a_{i_1}, ..., a_{i_{m_1}}]$$

- Ordering option 3 (variables for agent i):

$$[x_{i_1}, ..., x_{i_{n_i}}, a_{i_1}, ..., a_{i_{m_1}}, x'_{i_1}, ..., x'_{i_{n_1}}]$$

- Ordering option 4 (variables for agent i):

$$[x_{i_1}, ..., x_{i_{n_i}}, x'_{i_1}, ..., x'_{i_{n_1}}, a_{i_1}, ..., a_{i_{m_1}}]$$

These different orderings can have a great impact on the size of the resulting BDD, and consequently on MCMAS computation times. It is generally accepted that the ordering option 2 is the best to use in model checking multi-agent systems. It is based on the concept of *variable interleaving* in OBDDs [18], and adapted to agents.

3.2.4 Algebraic Decision Diagrams

MCMAS supports bounded integer variables (e.g. `x : 0..3`), and allows Boolean conditions to be numeric identities (e.g. `(x > 2)` or `(x + y = 3)` for integer vars `x` and `y`).

If an agent has an integer variable with a large range of values, then the number of Boolean variables needed to represent its state is also large (if variable `x` has n possible values, the corresponding agent needs at least $\log_2(n)$ state variables).

To avoid this, MCMAS uses alternative data structures called *algebraic decision diagrams* (ADD, [16]) to represent these variables and expressions. The CUDD manager also handles ADDs, and the ADD variables needed are stored in global vectors `addv` and `addpv`.

As there is no SDD equivalent for ADDs (yet!), our model checker does not support examples containing numerical values and expressions. Note that all these examples *could* technically be implemented in ISPL so that our model checker supports them, by simply replacing an integer range by an *enum* containing all the possible values that the variable can take: for example

```
x :  0..3
```

could be declared as

```
x :  {zero, one, two, three}
```

and expressions such as

```
if (x > 1)
```

could be translated to

```
if (x = two) or (x = three).
```

With ADDs being beyond the scope of this project, we did not study them further (in particular we did not look into the ADD reduction done by CUDD in the background), and therefore thought better not to implement this to keep the comparison fair between MCMAS and our model checker.

3.3 Model Checking with Sentential Decision Diagrams

In this section, we discuss the feasibility of implementing a model checker based on SDDs and the choices involved. We also reflect on the *potential* that SDDs have to outperform BDDs, and we explore from a theoretical point of view the different ways of using them.

3.3.1 Suitability of SDDs for Model Cheking

SDDs are representations of Boolean functions, and as such, they can theoretically be used to represent state spaces and transition relations of a model. We have seen in the background section of this report that there exist polynomial-time algorithms for Boolean operations (AND, OR, NOT) on SDDs, as well as for conditioning

an SDD on a literal, both of these being *essential* to an efficient model checker implementation. We showed also that these two operations (generally refered to in this report as `apply` and `condition`) are sufficient for implementing the entire model checking algorithm defined in 2.2.1. Hence, the SDD data structure is *a priori* a suitable representation for symbolic model checking, and its early success [9] in knowledge compilation gives us hope that it might even constitute an improvement on BDDs.

One of the directions of our investigation will be to search the space of vtrees in order to compare them, and give a first assessment of some specific vtree constructions in terms of their efficiency in the context of model checking. Before we launch into this search, we start by analysing *theoretically* the impact that the vtree will have on the overall performance of the model checker. This should give us some insights into how to structure this search, which would otherwise be very unpromising considering the size of the space of vtrees.

3.3.2 The Importance of Vtrees

We have seen that the size of an SDD can be significantly affected by the vtree with respect to which is is constructed. But in addition, the vtree has a huge impact on the efficiency of some SDD operations such as `apply` or `condition`.

In particular, a characteristic of a vtree v that we will look at very carefully is the size and configuration of its *left subtrees*, by which we mean the left children of the nodes of v, and their descendents. This is not a formally defined vtree characteristic, but it is of fundamental importance to the suitability of a particular vtree for model checking.

Recall the construction of an SDD for function f with respect to a vtree v. It involves creating an X-partition of f, where X is the set of variables in v^l. If X contains a lot of the variables in f, the *primes* of f will be large. But in SDDs, primes are given more importance than subs, as shown for example in the `apply` algorithm in 2.4.3, where the recursive call on subs is only made if the recursive call on corresponding primes did not return $\bot$.

The search for a good vtree does *not* amount to reducing primes as much as possible: if this were the case, there would be absolutely no point in trying to replace BDDs (whose *primes* are as small as possible – they are just one literal!) with SDDs. But the ultimate objective of vtree optimisation is to construct primes in such a way that:

- The number of decisions to make is less than it would be in a BDD;
- Decisions remain quick enough.

The word "decision" is deliberately used in a vague sense here, to incorporate both the combination of two SDD elements in an execution of `apply`, or the evaluation of an SDD on a variable instantiation (for `condition`).

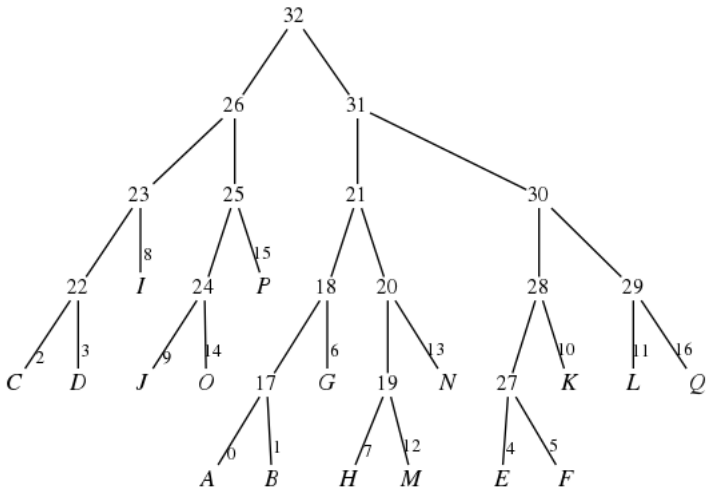


Figure 3.3: The l-subtrees of this vtree are the subtrees rooted at nodes 26, 21, 28, and 29.

3.3.3 A New Perspective on Vtrees

Taking into account the fact that the efficiency of a vtree fundamentally depends on the composition of its left children, we introduce a more precise way of describing vtrees which emphasises this point and facilitates our explanations in the rest of this report. Recall first that for a vtree node v, its left and right children are respectively refered to as v^l and v^r. By slight abuse of notation, we write v^{r^2} for the right child of the right child of v, etc.

Definition Let v be a vtree. An *l-subtree* of v is an element of the sequence

$$v^l, (v^r)^l, (v^{r^2})^l, ..., (v^{r^{k-1}})^l, v^{r^k},$$

where $k \geq 0$ is the smallest integer such that $(v^{r^{k+1}})$ is a leaf.

An example is given in Figure 3.3. Observe that for a right-linear vtree v, l-subtrees of v are all leaves (expect for the last one which is a node with two leaves). Also, if v is left-linear or vertical, the only l-subtree of v is v itself (we set $v^{r^0} = v$).

We also want to emphasise the fact that each variable of v lies on one of its l-subtrees, and furthermore, the sets of variables induced by the l-subtrees form a partition of the set of variables of v.

Our "new perspective" on a vtree consists in identifying it using the partition of the variables that its l-subtrees induces. That is, from now on we will be *comparing* vtrees by comparing their l-subtrees, and we will be *constructing* vtrees by creating a partition of the variable set, which will be used as a basis for constructing l-subtrees that we can then arrange and link together.

3.3.4 Vtree Options in Practice

The formalism defined in the previous section will facilitate the task of exploring the space of vtrees for applications to model checking, as we can now more easily characterise "good" vtrees. In practice however, whatever these good vtrees turn out to be, we need to be able to *generate* them automatically from a model of the multi-agent system we wish to verify. This means that this characterisation needs to have some *meaningful* links to the model itself (as a random partition is unlikely to do the job well, and doesn't constitute a very good heuristic!).

Recall that during the encoding process of an interpreted system for a multi-agent system, we assign a set of variables to each agent, separated into state variables, "primed" state variables, and action variables. This gives us a *natural* splitting of the full set of variables, made up of three different sets for each agent.

We now have a wide range of easily generated, meaningful options to investigate, as a variable partition can lead to a lot of different vtrees: we can change the shape of the l-subtrees we create from it, their order, or even combine them, split them further, etc. These options will be thoroughly explored in the Evaluation chapter of this report.

3.3.5 Dynamic Minimisation in Symbolic Model Checking

Dynamic minimisation is an essential element to symbolic model checking in practice. When state spaces grow larger, model checking without dynamic minimisation is often very impractical.

In 3.1.3, we described the dynamic vtree search algorithm implemented in the SDD Package, which our model checker relies upon. Recall that this algorithm automatically searches through the *whole* space of vtrees, i.e. it may lead to a complete re-structuring of the vtree if it improves the size of the SDDs considered. Although this might be a very good option for knowledge compilation, where the goal is to find the most compact representation possible, it seems to be a considerable effort in symbolic model checking, where a large number of different SDDs will be used at the same time (by the same *manager*), and modified a lot within a short period of time.

For this reason, we introduce the idea of limiting this automatic search to only *some* vtrees. The main option we will look at consists in re-structuring the l-subtrees only. It seems that this would constitute a faster search (as far less vtrees will be considered), while retaining the potential advantage which was originally brought by the particular choice of l-subtrees. So far this is of course just an idea, and we will need to either demonstrate, or reject this idea using real-life examples.

3.4 Implementation Specifics

3.4.1 Adapting MCMAS

We implemented a model checker for multi-agent systems entirely based on SDDs for symbolic representations.

Our starting point was the MCMAS source code, in which we proceeded to replace every call to CUDD with equivalent calls to the SDD Package. Throughout, we managed the reference counts of our SDD nodes by calls to `sdd_ref()` and `sdd_deref()`. Note that this often requires the introduction of temporary nodes; specifically, this occurs when updating the value of a node based on its previous value. For instance, the CUDD statement

```
// f, g are BDDs
f = f * g;
```

setting the value of f to be the conjunction of the BDDs for f and g, becomes

```
// f, g are SddNode*
SddNode* tmp;
f = sdd_conjoin(tmp = f, g, manager);
sdd_ref(f, manager);
sdd_deref(tmp, manager);
sdd_deref(g, manager); // (if this is the last time we use g)
```

This is to ensure that the previous SDD node pointed to by f, which the code can no longer access, is garbage collected. As a more complete example we provide the code for the `compute_reach()` function in appendix (To Do).

The main steps of our model checker are the same as those of MCMAS described in 3.2.1, and therefore the important functions which had to be adapted for SDDs are:

- the functions `encode_protocol()` and `encode_evolution()` for each agent, used to compute the global transition relation
- the `compute_reach()` function for computing the SDD representing the global reachable state space
- the `check_formula()` method (the implementation of `SAT`)

Additionally, new data types and functions had to be defined in order to deal with the SDD-specific aspects of the program. We replaced the MCMAS BDD parameters by our own *SDD parameters* (see appendix) containing the results of the variable allocation process in three different vectors: `variable_sdds`, `primed_variable_sdds`, `action_variable_sdds`.

```
function restrict(f, c):
// pre: c != ⊥ and variables of c are all variables of f
  if c = ⊤ or f = ⊥ or f = ⊤
    return f
  x := top variable in f // the root of the BDD
  if (x = top variable in c)
    return (restrict(f|x, c|x) ∨ restrict(f|¬x, c|x))
  else // x does not appear in c
    return (x ∧ restrict(f|x , c)) ∨ (¬x ∧ restrict(f|¬x , c))
end
```

3.4.2 Existential Quantification

Recall (2.2.2) that the model checking algorithm relies on the functions $\text{pre}_\exists$ and $\text{pre}_\forall$, which themselves depend on existential quantification, or the process of relaxing the constraint on one or more variables in a function.

In CUDD, the function responsible for existential variable quantification is `Cudd_bddExistAbstract()`. This takes as input two BDDs f and c, and returns the BDD for f after existential quantification of all variables in c. The implementation of this function is based on the following recursive algorithm called `restrict`, originally described in [23]:

In the context of symbolic model checking, existential quantification of a set of variable X from a function f can be done in CUDD by a call to `Cudd_bddExistAbstract(f, c)`, where `f` is the BDD for f and `c` is a function essentially depending on all the variables in X (in MCMAS `c` is set to be the conjunction of all the variables in X).

In the SDD Package, existential quantification can only be done one variable at a time, by calling `sdd_exists(x, f, manager)` which returns $\exists x f$. Our first solution was to iteratively call this function on every variable in X, but our first experiments revealed that this method was very slow. We therefore implemented the above algorithm in our model checker as an attempt to reduce computation time: In the code above, we have omitted the calls to `sdd_ref()` and `sdd_deref()` for clarity. Also, `ite` refers to an implementation of the ITE (If-Then-Else) operator on Boolean functions, namely

$$\text{ITE}(f, g, h) = (f \wedge g) \vee (\neg f \wedge h).$$

Another point worth discussing is the notion of "top variable" for an SDD. In a BDD, this is unambiguously refering to the variable with which the root is labelled, and this choice of variable for the start of the algorithm is clearly induced by the conditioning algorithm on BDDs, which consists in removing the *false* outcoming edge for each node labelled with the variable in question – if this is the top variable, then this amounts to simply removing the right (or left) subtree of the BDD.

```
SddNode* restrict_sdd(SddNode* f, SddNode* c, SddManager* manager)
{
  if(sdd_node_is_false(c)) // not allowed
    return NULL;
  if(sdd_node_is_true(c) || sdd_node_is_false(f) || sdd_node_is_true(f))
    return f;
  SddLiteral x = get_top_variable(f);
  SddLiteral y = get_top_variable(c);
  if (x == y) {
    SddNode* res1 = restrict_sdd(sdd_condition(x, f, manager),
        sdd_condition(x, c, manager), manager);
    SddNode* res2 = restrict_sdd(sdd_condition(-x, f, manager),
        sdd_condition(x, c, manager), manager);
    return sdd_disjoin(res1, res2, manager);
  } else {
    SddNode* res1 = restrict_sdd(sdd_condition(x, f, manager), c, manager);
    SddNode* res2 = restrict_sdd(sdd_condition(-x, f, manager), c, manager);
    return ite(sdd_manager_literal(x, manager), res1, res2, manager);
  }
}
```

In an SDD however, this is not so obvious. It would make sense for the "top variable" of a vtree to correspond to the top variable in the variable order induced by that vtree. In that case, it would be in that same order that the variables of `c` would be existentially quantified from f in our implementation.

But looking at the algorithm for `condition` described in 2.4.3 (which we *assume* the SDD Package uses, or at least a similar version), it is unclear what order would be the most efficient. (TODO say that we try both?)

3.5 SDD-Specific Features

3.5.1 Vtrees and Variable Orders

Recall that in the SDD Package, the use of the standard vtrees is only possible when a variable order is supplied. For convenience, we implemented the four standard MCMAS variable orders in our model checker, for use with one of the standard vtree, thereby making 16 different vtrees available. These orderings are implemented within the `get_var_order()` function, whose code is documented in appendix.

As previously mentioned, being able to create our own "non-standard" vtrees will be crucial for the utility of the project. The SDD Package API is lacking some basic functions in that area, most probably because it is generally sufficient for users to rely on the standard vtrees, together with the dynamic minimisation feature. There are, for example, no ways of constructing a vtree bottom-up, as one would construct an SDD node. Hence, to create our own customised vtrees, we had the following two options:

- Generate a variable order to create one of the standard vtrees, and apply swaps, left- and right-rotations until the vtree obtained is the desired vtree
- Generate a *vtree format* text file and use `sdd_vtree_read()` to create a `Vtree` object. According to the SDD manual, this file format is normally used to save vtrees to file in order to re-use them in the future. The function `sdd_vtree_save_to_file()` is the way to generate the file from an existing vtree.

We chose the second option, simply because it seemed more convenient than computing the set of operations to apply to get from one vtree to another. The vtree format, without being particularly readable, has the advantage of being relatively simple in syntax and therefore easy to generate.

The API of the SDD Package was clearly not designed to edit and manipulate vtrees practically, so as a replacement we created a new tree structure, the `vtree_node`, defined thus:

```
struct vtree_node
{
```

```
  // the children
  vtree_node * left;
  vtree_node * right;
  // more data for file generation
  int size;
  bool isleaf;
  int id;
};
```

This very simple structure significantly facilitated the process of constructing vtrees. We could simply create a leaf `vtree_node` for each variable needed, and construct the vtree bottom-up. Observe that the variables themselves are not stored; this is because we organise the nodes so that each variable is *one more* than the ID of the leaf which contains it (IDs start from 0 in the file, whereas variables are numbered from 1 in the SDD manager).

The function `vtree_node_get_file_content()` was written to recursively return the string representing the `vtree_node` in the file format required by `sdd_vtree_read()`. The code for the function, as well as an example, can be found in the appendix (To do).

In practical applications, the number of variables is not known in advance but we still need a way of generating the vtree dynamically. To this aim, we wrote the `create_vtree()` function which is called in the program as soon as the number of variables needed is known, but *before* the SDD manager is created, so that the vtree produced can be used during its initialisation – note that this is only possible since our vtree is not based on actual variables but on a temporary structure which does not involve the manager.

To use `create_vtree()`, we need to pass it the number of variables that the vtree should consist of, as well as the desired *vtree type*. This is an integer corresponding to one of the various vtrees which we implemented: vtree types 1-4 correspond to the standard vtrees defined in 3.1.2, whereas types 4-8 are "new" vtrees which we used for experimentation, thinking they had more potential in the context of model checking. We leave out the details for now and use the evaluation section to describe these various vtrees and the corresponding experimental results.

3.5.2 A New Dynamic Vtree Search Algorithm

In 3.3.5, we introduced the idea of a new dynamic vtree search algorithm, focusing on a re-structuring of the l-subtrees only. In this subsection we describe our implementation of this new algorithm.

First note that in the SDD Package, the default implementation of the minimisation function can be modified using `sdd_manager_set_minimize_function()`, indicating which function should be used instead. The code for our replacement function is as follows:

```
Vtree* vtree_group_minimize(Vtree* vtree, SddManager* manager) {
  // start from the root
  Vtree* current = vtree;
  // repeat until a leaf is reached
  while (!sdd_vtree_is_leaf(sdd_vtree_right(current))) {
    // minimise the left subtree using original algorithm
    sdd_vtree_minimize(sdd_vtree_left(current), manager);
    // and go down one step to the right
    current = sdd_vtree_right(current);
  }
  sdd_vtree_minimize(current);
  return vtree;
}
```

To illustrate this, we refer the reader to the vtree in Figure 3.3 above. If called on this vtree, the `vtree_group_minimize()` function will *only* search for an improvement of the subtrees rooted at nodes 26, 21, 28 and 29 (i.e. the l-subtrees).

3.6 Software Engineering Issues

Here we explain some of the challenges faced during the implementation of the model checker, as well as our approach to testing correctness of the model checker.

3.6.1 Garbage Collection

As described in 3.1.4, the SDD Package has a automatic garbage collection feature based on reference counts, but it leaves the referencing and dereferencing of nodes to the user.

If not done properly, node referencing can lead to issues such as unwanted garbage collection, dead nodes kept in memory, or dead node dereferencing, all of which being very undesirable (but for different reasons).

This forced the constant track-keeping of node reference counts and required more debugging time. Nonetheless, it is important to mention that the manual referencing and dereferencing of nodes is probably more efficient in the long run than if the SDD manager had to do it in the background, which would require more thinking on its part.

3.6.2 Comparing SDDs and BDDs

The goal of this implementation was to build an SDD-based model checker which produced the same results as MCMAS in all circumstances. We needed to make sure that the Boolean functions representating our sets of states and transition relations

were the same at each step of our model checking algorithm. Unfortunately, there is no convenient and precise way of programmatically comparing an SDD with a BDD.

The only *exact* method available is to look at the Boolean function corresponding to each data structure, and compare these. However, due to their syntactic differences, it had to be done using a SAT checker (we used [32]), which is a good solution for small functions but becomes impractical very quickly when the number of variable exceeds about 20 (which happens in all non-trivial cases).

An useful alternative is to construct the SDD with respect to a right-linear vtree and the BDD with the equivalent variable ordering (see 2.4.4 for details), to ensure that the resulting structures are comparable. We then have two options:

- Compare the graphical representation of each data structure, provided by both APIs via a DOT file [33]. An image is often enough to tell if two representations are not equivalent, but in the case where they are, it can be a very long process to manually verify it.

- Compare the size (i.e. the number of nodes) of each structure. Again, most of the time two non-equivalent representations have very different sizes, but representations with comparable sizes are not necessarily equivalent (note that due to the way SDD and BDD nodes are counted, the size of a BDD is not exactly the same as the size of the equivalent SDD so this method is not 100% conclusive either).

This second option is nonetheless the only available solution in the case of very large SDDs and BDDs, which explains the difficulty in debugging larger examples.

Reflecting back on this issue, it would have been much less time-consuming to write a small logic equivalence tool ensuring that the Boolean functions obtained from two equivalent data structures was not only equivalent, but also *syntactically equal*, to reduce the problem to a simple comparison of strings. We did not anticipate enough the amount of work that would be required when debugging the model checker.

3.6.3 Correctness

Due to the reasons mentioned above, we had to use a few simple tricks to make sure that our code was indeed an implementation of the model checking algorithm.

The first technique we used was that of comparing the *output* of our model checker with that of MCMAS, on the set of examples that we had available (see 4.1.2 for a detailed description of these examples). Once we were certain that these were identical (so no obvious bugs existed), we did a more thorough search to ensure that there was nothing wrong with the code.

Although our set of examples was relatively diverse, it certainly did not cover the whole range of expressions, data types, etc. available in ISPL, nor did it use

the full set of CTLK connectives that ISPL supports. Hence, our second approach to verification was to "invent" new formulae to check, using different combinations of connectives from those to be tested. Also, we made a series of modifications to the example ISPL files to check that most (not all, see 3.2.4) of the ISPL syntax was handled by our model checker. This even revealed a bug in the MCMAS source code (which had persisted in our code) causing a segmentation fault when the input ISPL file used more than one bitwise XOR (^). This bug has now been fixed (in both model checkers).

Although these verification techniques are not guaranteed to have caught all bugs, we are confident that they were sufficient for our purposes, and that in all the examples considered, both model checkers had the same outcome.

Chapter 4

Evaluation

4.1 Introduction

4.1.1 Evaluation Strategy

The objective of this evaluation is to experiment with SDDs in the context of model checking multi-agent systems, and explore the various in which situations our model checker performs best.

TODO The plan for quantitative analysis is to do a side-by-side comparison of our model checker and MCMAS on a series of examples, and conclude. However, in order to provide a fair comparison we had to ensure that both structures were performing "at the best of their abilities".

There are suggested methods for the use of BDDs in model checking, in particular concerning initial variable orders [17]. For the specific case of MCMAS we observe that the standard order number 2 (see 3.2.3) *often* yields the best results. On the other hand, SDDs had not yet been explored in the context of model checking, and consequently no heuristics existed.

Through this evaluation we planned to remediate this by investigating various vtree constructions and comparing the resulting model checker with MCMAS.

To start with, we decided to focus on the issue of *static* vtree generation, for the situation where the vtree has to be determined *before* SDDs are constructed, as opposed to continuously modified as the construction process happens – this is called *dynamic* minimisation.

Although it may seem like wasted effort, the importance of static vtree generation is non-negligible, for two reasons: firstly, it helps us understand what aspects of the vtree have the greatest impact on the final SDD; secondly, even though dynamic minimisation is generally the most efficient technique, it can also be a very time-consuming process which some applications might find less practical.

After this first investigation, we planned to look at the performance of SDDs when enabling the dynamic minimisation feature of the SDD Package, and compare

it to BDD dynamic variable reordering as implemented in MCMAS.

We hoped that both of these analyses (static and dynamic) would provide enough information to know whether or not SDDs (as implemented in the SDD Package) are suitable for model checking multi-agent systems.

4.1.2 Example Models

In this section, we describe the "real-life" example models used as benchmarks throughout this chapter. This is to provide the reader with some context before launching into numerous tables of results, and to help understand the analysis we make of these results.

Our set of examples is made up of three main classes of models, presented below. The (very trivial) bit transmission problem (Appendix A) is also sometimes used to demonstrate the *inefficiency* of a particular technique, so we also provide some useful characteristics of this particular model.

The Dining Cryptographers

The original dining cryprographers problem was introduced by D. Chaum in 1988, refering to the following situation: three cryptographers go out for dinner, and at the end of the meal, the waiter informs them that the bill has already been paid by someone, who could either be one of the cryptographers, or the National Security Agency (NSA). While respecting each other's right to remain anonymous, they would like to find out if the NSA has paid. They use the following two-step protocol: in the first step, each cryptographer flips a coin behind a menu, so that only them and the cryptographer to their right can see the outcome; then, they each know the outcome of two out of the three coin flips, and in the second step, the cryptographers that haven't paid publicly announce whether or not these two outcomes are the same. If one cryptographer *has* paid for the meal, they announce the opposite. The NSA has paid for the meal only if there is an even number of "same".

This problem can be generalised to any number n of cryptographers, and modelled by a multi-agent system consisting of one agent for each cryptographer, and an environment holding information about the coin flipping outcomes and the cryptographers' claims.

With the help of an ISPL file generator, we use a *parametrised* version of the dining cryptographers problem, allowing us to easily rescale the model depending on the efficiency of the approach considered. The CTLK formula we are verifying in this example is

$$AG\left[(\text{odd} \wedge \neg \text{c1paid}) \rightarrow (K_{c1}(\text{c2paid} \vee ... \vee \text{c}n\text{paid}) \wedge \neg K_{c1}\text{c2paid} \wedge ... \wedge \neg K_{c1}\text{c}n\text{paid})\right],$$

which encodes the following true property: *"At any point, if we know that there is an odd number of 'same' and the first cryptographer is not the one that paid for*

dinner, then the first cryptographer knows that one of the other cryptographers paid, without knowing which one".

Dining Cryptographers				
n	#agents	#reachable states	#vars (states/actions)	depth
4	5 (4 + env.)	400	57 (24/9)	4
5	6	960	69 (29/11)	4
6	7	2240	81 (34/13)	4
7	8	5120	93 (39/15)	4
8	9	11520	105 (44/17)	4
9	10	25600	117 (49/19)	4

100 Prisoners and a Lightbulb

The *100 prisoners and a lightbulb* problem is a mathematical riddle. The problem, of unknown origin, is stated as follows:

> One hundred prisoners have been newly ushered into prison. The warden tells them that starting tomorrow, each of them will be placed in an isolated cell, unable to communicate amongst each other. Each day, the warden will choose one of the prisoners uniformly at random with replacement, and place him in a central interrogation room containing only a light bulb with a toggle switch. The prisoner will be able to observe the current state of the light bulb. If he wishes, he can toggle the light bulb. He also has the option of announcing that he believes all prisoners have visited the interrogation room at some point in time. If this announcement is true, then all prisoners are set free, but if it is false, all prisoners are executed. The warden leaves, and the prisoners huddle together to discuss their fate. Can they agree on a protocol that will guarantee their freedom?

There are several solutions to the riddle. The one that we will be verifying involves designating a "counter" amongst the prisoners, who will be the only one allowed to announce whether he believes all prisoners have visited the room.

Using another file generator, we can create a parametrised (scalable) version of the problem as a multi-agent system, where the agents are the Prisoners (excluding the counter), the Counter, the Prison (keeping track of which prisoners have visited the room, and whether or not they have been released/executed), and the Environment (containing the current state of the light bulb and the identity of the prisoner being interrogated). The parameter n corresponds to the number of prisoners, including the counter. We verify the following CTL properties:

- *AF* Release: *The prisoners will eventually be released.* (False)
- *EF* Release: *It is possible that the prisoners will eventually be released.* (True)
- *EF* Execute: *It is possible that the prisoners will eventually be executed.* (False)
- *AG*¬Execute: *The prisoners will never be executed.* (True)

(It is a *safe* model, in the sense that the counter will only make an announcement if he is absolutely certain that all prisoners have visited the room.)

Prisoners				
n	#agents	#reachable states	#vars (states/actions)	depth
3	5 (4 + env)	50	33 (13/7)	9
5	7	746	50 (20/10)	17
6	8	2631	57 (23/11)	21
7	9	8980	64 (26/12)	25
8	10	30001	72 (29/14)	29
9	11	98798	81 (33/15)	33
10	12	322107	88 (36/16)	37

The Needham-Schroeder Public-Key Protocol

The Needham-Schroeder Public-Key protocol [24] (NSPK) is a communication protocol based on public-key cryptography, aiming to establish mutual authentification between an initiator A and a responsor B.

As opposed to the previous two classes of models, this one is not parametrised, but we have different *versions*, consisting in one or more *instances* of the protocol. Each instance is an agent, and the environment stores information about each of them.

The CTLK formulae to check differ depending on the version, but all of them involve verifying that different instances of the protocol *agree* by the time they are finished. For example, in the first version, the only formula to check is

$$AG(\text{instance\_2\_end} \rightarrow K_{\text{instance\_2}}(\text{agree\_instance\_2\_and\_instance\_1})).$$

NSPK				
version	#agents	#reachable states	#vars (states/actions)	depth
1	2	618	75 (29/17)	6
2	3	42240	107 (42/23)	9
4	3	5736	108 (42/24)	6
5	4	2.88×10^6	176 (72/32)	11
6			262(106/50)	

The Bit Transmission Problem

The problem and its model are fully described in Appendix A. Below are some stats:

Bit Transmission Problem			
#agents	#reachable states	#vars (states/actions)	depth
3	18	17 (6/5)	4

Range of Examples

The dining cryptographers models are an example of a model where one of the agents (namely, the environment) has a very large evolution function, leading to an even larger local transition relation. The prisoners models are an example of a very *deep* model, making it longer to obtain a fixed point in the reachable state space computation, and harder to check some of the formulae. Finally, the NSPK are extremely large examples of models, particularly versions 5 and 6.

These three "classes" of models are therefore relatively diverse, very different from each other, and will help us analyse more accurately the results obtained in our experiments.

4.1.3 Experimental Framework

Throughout this chapter, we report a number of tables containing experimental results. Here, we attempt to describe as accurately as possible the framework in which these experiments took place, as well as the format in which we present the results.

All benchmarks were run on a Linux Ubuntu 13.04 64-bit machine, with 16GB memory and a Intel Core i7-4770 processor at 3.40GHz. Results of all experiments are split into two tables: computation time and memory usage, where time measurements are in seconds, memory usage in megabytes. All computations were performed three times and computation times were averaged; the memory used is always the same on a particular example, as both MCMAS and our model checker are deterministic. In each table we indicate in bold the situations where SDDs outperformed BDDs.

We indicate by a dash '-' the cases where the system either ran out of memory, or reached a timeout of an hour. It is interesting to note that in the case of static vtree generation (4.2 & 4.3), all dashes represent OOMs, whereas in our dynamic minimisation experiments (4.4), a dash always corresponds to a timeout.

For each situation explored, we selected a set of models from the ones described in 4.1.2, including several of the same type with different parameters. This selection was done in such a way that the range of results would be as broad as possible, while remaining within the timeout/memory limit.

Finally, in addition to the results of the experiments conducted on SDDs, most tables contain the best BDD results obtained on the same models, in order to make the comparison easier.

4.2 Static Comparisons with Standard Vtrees

We started our investigation by experimenting with the standard vtrees, namely right-linear, left-linear, balanced, and vertical. Recall that this first investigation was focused exclusively on static vtree generation, so we disabled *both* the dynamic SDD minimisation feature in our model checker, and the CUDD dynamic reordering feature in MCMAS. Due to the SDD Package design, this forced the garbage collector to be off as well in our model checker.

4.2.1 Right-Linear Vtrees

Our first experiments involved right-linear vtrees. Recall that BDDs are structurally identical to SDDs build using a right-linear vtree, and the same variable ordering (2.4.4). Experimenting with this particular type of vtrees was therefore of significant importance, as it would allow us to compare both model checkers in the case where they were handling data structures of the exact same size.

Tables 4.1 and 4.2 correspond, respectively, to the time and memory comparisons of our model checker with MCMAS, when SDDs were built with right-linear vtrees. We include results for each of the four standard orderings with which the vtrees were constructed.

During this first phase of the investigation, it became quickly apparent that SDDs were *not* better than BDDs in this particular situation, outperforming them top in only three cases, and being slower in most cases, sometimes considerably (up to two orders of magnitude). The important difference in memory usage can be explained by the garbage collector being off in the SDD Package.

We observe that the time difference is more important in the prisoners and NSPK examples than for the dining cryptographers. A particularity of these two examples is that they involve larger data structures: the prisoners models both have a large number of reachable states, and in the NSPK models, each agent is allocated

	Ordering 1		Ordering 2		Ordering 3		Ordering 4	
	BDDs	SDDs	BDDs	SDDs	BDDs	SDDs	BDDs	SDDs
cryptos7	2.06	12.35	2.64	6.39	3.33	12.77	3.83	15.186
cryptos8	11.85	56.02	17.24	30.42	20.41	58.70	23.85	69.54
cryptos9	853.17	**221.16**	98.95	145.313	617.32	**266.84**	911.70	**305.17**
pris9	50.14	-	11.47	252.21	12.52	369.83	17.31	-
pris10	236.15	-	35.11	-	42.12	-	52.13	-
nspk1	0.06	0.79	0.04	0.59	0.09	1.21	0.06	0.71
nspk2	11.25	154.46	2.71	76.32	7.36	120.42	4.55	84.15
nspk4	1.29	27.29	0.32	9.79	1.75	23.74	0.95	14.68
nspk5	-	-	158.62	-	-	-	-	-
nspk6	-	-	-	-	-	-	-	-

Table 4.1: Computation Time: SDDs were built with right-linear vtrees, and standard orderings

	Ordering 1		Ordering 2		Ordering 3		Ordering 4	
	BDDs	SDDs	BDDs	SDDs	BDDs	SDDs	BDDs	SDDs
cryptos7	198	782	160	306	248	656	272	791
cryptos8	758	3120	647	1308	1024	2630	1120	3166
cryptos9	3001	11556	2850	5566	4246	10134	4690	11927
pris9	594	-	313	7658	3502	12382	4349	-
pris10	1955	-	847	-	968	-	1218	-
nspk1	16	34	13	24	25	50	17	34
nspk2	324	2065	135	1012	493	1520	275	1104
nspk4	92	624	45	223	188	693	81	344
nspk5	-	-	3841	-	-	-	-	-
nspk6	-	-	-	-	-	-	-	-

Table 4.2: Memory Usage: SDDs were built with right-linear vtrees, and standard orderings

around 40 variables, leading to very large transition relation SDDs. These results therefore seem to imply that the SDD manager is not as efficient as CUDD when it comes to handling large data structures. We also mentioned that it was likely for the `condition` algorithm to be slightly more efficient on BDDs than on SDDs.

These hypotheses are contradicted by the three situations where SDDs are faster than BDDs. We suspect that this is due a memory management issue in CUDD: it is possible that these particular models filled up the cache and slowed down the rest of the computation. This is difficult to check, as CUDD handles this internally. Moreover, pushing the comparison further would be impossible, as SDDs run out of memory at the next level.

Although these particular results are disappointing, they could have been expected: although the data structures are equivalent, the SDD algorithms have not been particularly optimised for this particular type of vtrees, as they must be prepared to handle different types of data structures. On the other hand, CUDD was developped specifically for BDDs, and has the advantage of being much older and benefiting from the extensive BDD research done in the past few decades.

To conclude this first subsection, we make the following remark: in this situation the model checkers are dealing with the *exact* same structures, so these numbers *should* be the same for BDDs and SDDs. But they aren't, for the reasons explained above and in 3.1.4. However, should the SDD Package ever become as efficient as CUDD, any improvement on the SDD numbers in Tables 4.1 and 4.2 would be an improvement on BDDs. This is hypothetical, but it motivated the rest of our experiments, and in particular, the search for a better vtree.

Note on BDD data: from now on, all the BDD data found in the comparison tables corresponds to computations using standard ordering number 2, the most efficient for BDDs in the vast majority of cases (3.2.3).

4.2.2 Other Standard Vtrees

We moved on from right-linear vtrees, hoping that one of the other standard vtrees would lead to more efficient computations. This was not the case.

Left-linear and vertical vtrees proved *extremely* inefficient. As shown in Tables 4.3 and 4.4, even on the small examples chosen they were significantly slower than BDDs. Furthermore, although the variable ordering chosen for these vtrees seems to have a big impact on the computation (much bigger in fact than on BDDs), even with the better ones there is no competition.

Before pushing the analysis further, we look at balanced vtrees; these are slightly more efficient, but still lead to much higher computation times than BDDs, or even right-linear vtrees. This results appear very clearly in Tables 4.5 and 4.6.

	BDDs	Left-Linear				Vertical			
		O. 1	O. 2	O. 3	O. 4	O. 1	O. 2	O. 3	O. 4
btp	0.02	44.3	8.36	6.42	6.56	0.61	0.13	0.09	0.10
cryptos4	0.02	-	-	-	-	-	21.72	57.67	47.15
pris3	0.01	-	-	-	-	1267.91	3.14	4.56	4.34
pris5	0.04	-	-	-	-	-	-	-	-
nspk1	0.03	-	-						

Table 4.3: Time Comparisons with left-linear vtrees.

	BDDs	Left-Linear				Vertical			
		O. 1	O. 2	O. 3	O. 4	O. 1	O. 2	O. 3	O. 4
btp	9	559	176	122	121	15	4	4	4
cryptos4	11	-	-	-	-	-	267	522	407
pris3	9	-	-	-	-	7272	48	73	55
pris5	13	-	-	-	-	-	-	-	-
nspk1	12	-	-						

Table 4.4: Memory with left-linear vtrees.

4.2.3 Towards a Better Vtree: Analysis & Observations

It was very clearly apparent in our experiments that the standard vtrees were *not* appropriate for model checking. Left-linear and vertical vtrees are *extremely* slow, balanced vtrees are *very* slow, and right-linear vtrees are barely usable, and still much slower than BDDs, when they should theoretically be just as fast. In this subsection, we attempt to understand why this might be the case.

Recall our "new perspective" on vtrees, defined in 3.3.3, which involved identifying a vtree by studying its *l-subtrees*. In particular, we saw that left-linear and vertical vtrees only have one big l-subtree, whereas balanced vtrees have a few more, and right-linear vtrees have almost as many l-subtrees as variables.

This seems to suggest that, as previously imagined, there is an important link between a vtree's l-subtrees and its performance in model checking. Although the experiments conducted so far are not enough to tell us much more about the nature of this link, we will base the rest of our evaluation on the assumption that the l-subtree structure of a vtree is one of the most important factors of its performance in model checking (more important, for instance, than the variable order it induces).

	BDDs	Balanced			
		O. 1	O. 2	O.3	O.4
cryptos5	0.06	0.92	23.56	13.78	8.38
cryptos6	0.44	5.78	254.3	109.28	62.26
cryptos7	2.06	40.76	-	2514.66	714.31
pris5	0.04	43.01	2.46	2.23	4.46
pris6	0.15	-	16.55	11.37	50.20
pris9	50.14	-	-	3182.7	-
nspk1	0.06	7.73	1.11	19.01	4.34
nspk2	2.71	-	1063.2	-	-
nspk4	0.32	-	18.33	88.12	-

Table 4.5: Balanced Vtrees: Computation Time

	BDDs	Balanced			
		O. 1	O. 2	O.3	O.4
cryptos5	18	29	610	329	229
cryptos6	45	136	4314	1687	1402
cryptos7	198	612	-	8900	8134
pris5	14	1562	45	33	59
pris6	31	-	178	120	215
pris9	594	-	-	8827	-
nspk1	16	206	16	652	63
nspk2	324	-	3970	-	-
nspk4	92	-	189	960	-

Table 4.6: Balanced Vtrees: Memory Usage

An "Ideal" Example

In order to get a better idea of what an efficient vtree *could* look like, we run our model checker on a few examples, after temporarily *enabling* the dynamic minimisation feature. We then inspect the vtrees found by the search algorithm, one of which is shown in Figure 4.1. We make some observations:

- The vtrees obtained after dynamic minimisation all seem to be *very* similar in structure, whatever the initial vtree or the example considered. The vtree in Figure 4.1 is a fair representative in terms of the size and structure of the l-subtrees.
- The dynamic minimisation algorithm is "symmetric" in the sense that it does *not* force the vtree in one particular direction. It considers 24 vtrees (of which 12 are induced from the l-vtree by applying one of the usual vtree operations, and 12 arise from the r-vtree) and picks the best one. The fact that we get this vtree structure in all cases is evidence that it is in fact the *ideal* configuration.
- One of the l-subtrees in the middle is very large – it contains 30 variables, over a third of the total. This shows that smaller primes are *not* necessary for an efficient computation, and that larger sets of variables may in fact improve it.
- It tends to happen (and it is definitely the case in Figure 4.1) that the first few l-subtrees are slightly smaller than the ones below. A reasonable explanation for this is that the first few l-subtrees correspond to the primes in the first few levels of the corresponding SDD (starting from the root), i.e those which need to be evaluated most often.

It is now even more evident that the l-subtree decomposition is a reasonable basis for vtree construction. We will take inspiration from these "ideal" vtrees in order to find a more efficient alternative to the standard vtrees.

4.3 Static Comparisons with Alternative Vtrees

In this section we explore and evaluate the various vtree constructions permitted by the l-subtree decomposition described in 3.3.3, and induced by the partition of the set of variables corresponding to the agents' variable allocation. We are still focusing on static vtree generation, so the dynamic minimisation features of both CUDD and the SDD Package are turned off.

This section is split into multiple subsections, each corresponding to a new investigation (but using the same construction process). At the beginning of each subsection, we describe the goal of the corresponding investigation and the particular vtree construction considered. We then present some experimental results and discuss any potential conclusions, which led us to the next investigation.

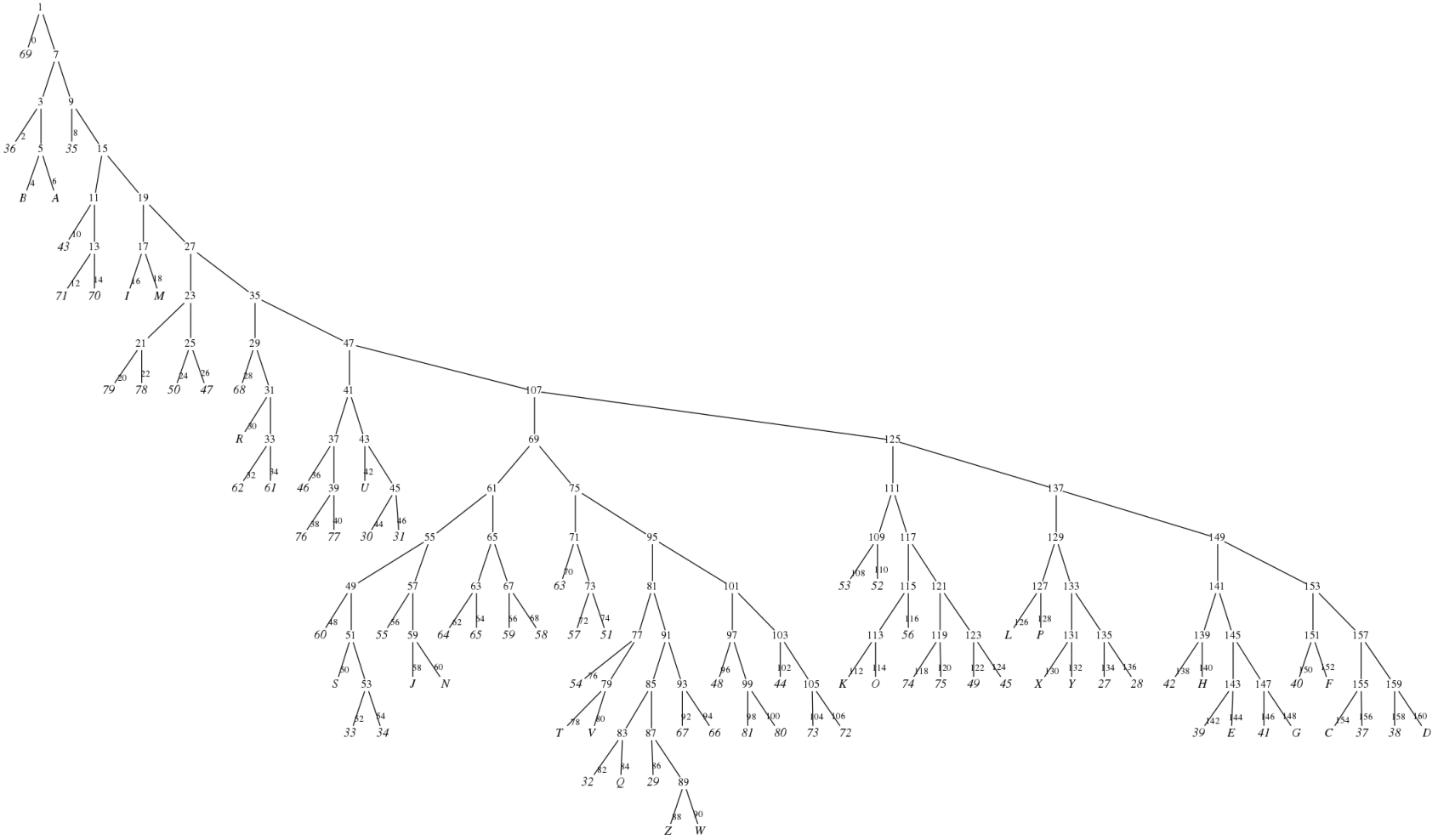


Figure 4.1: The manager's vtree at the end of a run of our model checker on crypto6, with an initial vtree of type balanced.

Before we start, let us recall the setup: we are in the middle of verifying an interpreted system of agents, variable allocation *has* taken place, and to each agent have been assigned three (distinct) sets of variables: state variables, primed state variables, and action variables. The union of the sets of each agent is the full set of variables used to encode the model. In order to construct a vtree for the model checker, we want to create a partition of this set which will be used to create the l-subtrees of the vtree.

4.3.1 A First Attempt Using Agent Variables

The first vtree construction we consider is the most natural, and was already suggested as a starting point in 3.3.4: we take the partition consisting of the variables allocated to each agent. That is, each agent corresponds to an l-subtree in the final vtree (strictly speaking, the last agent corresponds to the last *few* l-subtrees of the vtree constructed).

More specifically, for each agent we choose to construct a balanced subtree as according to the observations made in 4.2.3, it is the closest to the ideal vtree. We need to point out however, that constructing a perfectly balanced vtree for each agent is not often possible, as for this we would need the number of variables of each agent to be a power of 2. We are particulatly interested in the following:

1. The efficiency of this particular l-subtree decomposition;
2. The effect of changing the l-subtree order of the vtree;
3. The effect of changing the variable ordering induced by each l-subtree.

To evaluate these three parameters, we started our investigation with the most basic configuration: the l-subtrees were ordered using the *agent order* (i.e. the way agent definitions are ordered in the ISPL file), and the variable ordering within each agent subtree was simply [state variables, primed state variables, action variables].

Then, we looked at the effect of changing the l-subtree order, by trying successively the *inverse* of the agent order, and the *ascending* order (in terms of the number of variables needed for each agent).

Finally, we evaluated the impact of the variable order of each balanced subtree, and tried various orderings, in particular the equivalent of the MCMAS standard ordering 2, namely

$$[(\text{states and primed states interleaved}), \text{actions}].$$

Tables 4.7 and 4.8 contain our experimental results. It appears very clearly that the second variable ordering (MCMAS number 2) is *much* more efficient than the original one, in all cases. This is somewhat surprising, since when we compared balanced subtrees in 4.2.2, we concluded that the impact of the variable ordering differed a lot depending on the model. Yet, when these balanced subtrees are used

	BDDs	$[s, s', a]$			$[(s_1, s'_1, ..., s_n, s'_n), a]$		
		Agent	Inverse	Ascending	Agent	Inverse	Ascending
cryptos4	0.02	2.41	1.31	1.31	0.24	0.15	0.15
cryptos5	0.06	31.32	32.08	32.08	0.68	0.31	0.31
cryptos6	0.44	993.71	-	-	4.05	1.01	1.01
cryptos7	2.64	-	-	-	34.59	5.84	5.04
cryptos8	17.24	-	-	-	-	36.94	32.91
cryptos9	98.95	-	-	-	-	271.16	248.18
pris6	0.15	7.43	3.94	2.52	2.02	2.16	1.03
pris7	0.57	48.01	21.36	10.22	8.31	8.63	3.61
pris8	2.08	303.73	96.39	48.76	24.64	23.17	9.43
pris9	11.47		3259.27	256.0	120.17	109.24	38.75
pris10	35.11		-	-	445.36	347.23	128.1
nspk1	0.03	1.85	2.01	2.12	0.71	0.81	0.99
nspk2	2.71	542.32	-	-	133.53	163.57	124.32
nspk4	0.32	68.95	54.69	55.65	4.62	10.47	8.47

Table 4.7: Vtree Option 5 – Time

	BDDs	$[s, s', a]$			$[(s_1, s'_1, ..., s_n, s'_n), a]$		
		Agent	Inverse	Ascending	Agent	Inverse	Ascending
cryptos4	11	59	44	44	6	3	**3**
cryptos5	18	673	574	574	31	7	**7**
cryptos6	45	8716	-	-	196	33	**33**
cryptos7	160	-	-	-	1367	170	165
cryptos8	647	-	-	-	-	997	974
cryptos9	2850	-	-	-	-	5270	5066
pris6	31	76	45	52	42	31	**29**
pris7	48	316	141	158	127	88	83
pris8	102	1558	483	744	387	116	233
pris9	313		9000	3052	1418	700	724
pris10	847		-	-	5255	2059	2435
nspk1	16	27	29	25	11	13	**13**
nspk2	324	2949	-	-	1301	1374	1396
nspk4	92	685	884	901	63	81	**85**

Table 4.8: Vtree Option 5 – Memory

as l-subtrees of a larger vtree, the results are more constistent, and there is a clear winner.

Concerning the impact of the l-subtree order in the vtree, the conclusion is less obvious from the tables alone, and again, it seems to depend on the model. We inspect the results a little more closely. In a majority of cases, it appears that leaving the bigger agent subtrees at the bottom of the vtree (using the ascending order) reduces computation times. But this is not always the case.

For the dining cryptographers in particular, when the SDD sizes exceed a certain number, it is more efficient to leave the Environment subtree (the bigger one by far, and the first in the agent order) at the top of the vtree. One possible explanation is that in this model, the transition relation for the Environment is *considerably* larger than that of individual Cryptographers, therefore the major part of the computation time is spent in the reachable state-space computation, applying this particular relation (the others are negligible). The time gained by facilitating this unique operation therefore makes a more significant difference than that of facilitating all the others.

We suspect that for the NSPK examples, where results are also a little inconsistent, the same problem occurs. However, this is harder to analyse, considering agents in these examples have comparable numbers of variables.

It is also interesting to compare the time and memory usage in different runs of the same example, but with different vtrees. A few "irregularities" can be found there too, which confirms the fact that the size of the data structures is not the only thing affected by the vtree, but the efficiency of operations is too. This was much less appearant in the case of right-linear vtrees (i.e. changing the variable ordering of a right-linear vtree keeps the ratio constant, whereas changing the structure of the vtree makes it vary).

We can conclude that when l-subtrees correspond to agent variable sets,

- The most efficient variable order with this is the equivalent of the standard order number 2 in MCMAS;
- Ordering the subtrees with respect to their size is unpredictable.

To prevent this uncertainty, we had the idea of setting an *upper bound* on the size of the l-subtrees in the vtree that we construct. This is the next investigation.

4.3.2 An Upper Bound on the Size of Subtrees

Theoretically, setting a maximum number of variables per l-subtree should prevent the lengthy operations sometimes caused by an irregularity in the number of variables allocated to various agents, as described above. Whether or not this will reduce the *total* computation time, or the size of the resulting SDDs, is still to determine.

In this subsection, we start with the same partition of the set of variable as in the previous one (i.e. the partition arising from agents), however we *bound* the

size of each set in the partition by a given integer N. This means that if an agent has been allocated more than N variables, it will be "represented" by two or more l-subtrees in the final vtree. With these experiments, we try to explore the following questions:

1. What is an appropriate upper bound N? Should it depend on the total number of variables?

2. When l-subtrees are limited to a size of N variables, does their order still matter?

3. If we need to split an agent's variable set into multiple subsets, how should we arrange the variables?

4. Does any of the above lead to an improvement on the vtree defined in the previous subsection?

To investigate these questions, and draw conclusions as accurately as possible, we make some experiments.

We start by comparing different values of the upper bound N, which we define as a function of the total number of variables, denoted n. The main reason for using a function rather than a constant is that for larger-scale models, where agents have a large number of variables, a small upper bound is likely to slow down the process rather than accelerate it – we do not want to lose the value that l-subtrees are here to add in the first place (that is, to lower the number of *decisions* to make). That said, we choose functions which "grow slowly", so that even for *very* large-scale models, the upper bound is still reasonably low.

As shown in Tables 4.9 and 4.10, we used three different functions for this experiment: $N = log_2(n), N = \sqrt{n}$ and $N = 2log_2(n)$, where for each example, we round up N to the nearest integer. These functions almost behave like constants on the set of examples that we look at. As an indication, we give the actual values of N for each example, in each case, in the table below:

Example	$log_2(n)$	$\sqrt{n}$	$2log_2(n)$
cryptos7	7	10	14
cryptos8	7	11	14
cryptos9	7	11	14
pris9	7	9	13
pris10	7	10	13
nspk1	7	9	13
nspk2	7	11	14
nspk4	7	11	14
nspk5	8	14	15

	BDDs	Values of N		
		$log_2(n)$	$\sqrt{n}$	$2log_2(n)$
cryptos7	2.64	1.51	1.21	1.02
cryptos8	17.24	5.70	4.78	4.90
cryptos9	98.95	30.46	27.51	63.06
pris9	11.47	42.17	32.86	27.98
pris10	35.11	157.01	112.781	102.0
nspk1	0.03	0.55	0.48	0.65
nspk2	2.71	27.88	20.15	20.05
nspk4	0.32	3.94	5.12	4.16
nspk5	158.62	-	859.42	782.07

Table 4.9: Time Comparisons With Different Values of the Upper Bound N. In the table, n denotes the total number of variables in the model.

	BDDs	Values of N		
		$log_2(n)$	$\sqrt{n}$	$2log_2(n)$
cryptos7	160	65	56	51
cryptos8	647	216	188	154
cryptos9	2850	1117	886	1350
pris9	313	1086	718	698
pris10	847	3464	2576	2262
nspk1	13	12	12	12
nspk2	135	370	273	257
nspk4	45	89	78	72
nspk5	3841	-	6378	5876

Table 4.10: Memory: Different Values of the Upper Bound N.

It is clear from the results that setting an upper bound *does* reduce SDD computation times significantly, to the point where the SDD computations are around *three times faster* than the BDD computations in the case of the dining cryptographers, and at a comparable level (the difference being less than an order of magnitude) in all other cases. Observe that this is the first time we have achieved faster computations than with right-linear vtrees. It is also the first time we have managed to verify NSPK, version 5 with SDDs without running out of memory.

On the other hand, the upper bound itself doesn't seem to have such a big impact on the computation: none of the ones we tried seems to come out as a more efficient solution than the others in the general case. Our conclusion from this is that it is sufficient to "smoothen" the vtree by splitting up the very large l-subtrees, but it does not matter too much how small the resulting l-subtrees are.

These results confirm our hypothesis that large l-subtrees are "blocking" the computation at various steps of the execution, and that they were the main factor of the slow performances obtained in our previous experiments, particularly with the left-linear and vertical vtrees.

Recall now that in 4.3.1 we experimented with different *orderings* of the vtree's l-subtrees. We saw that these had a relatively important effect on the efficiency of the computation, and we concluded that the *irregularity* in the number of variables allocated to each agent was the main cause of this effect, as changing the order would mean shifting very large l-subtrees up and down the vtree (leading to an important change in the SDDs themselves).

Now that we have an upper bound on the l-subtree size, we would like to find out whether changing this ordering still has such a big impact. More generally, we would like to know the impact of the way we re-arrange variables when splitting an l-subtree due to its size exceeding the upper bound N. In the following experiment, we set the upper bound to be $N = \sqrt{n}$, and we compare the following three vtree constructions:

1. The l-subtrees are still balanced and in the agent order, and if one of them has size $> N$, we split it into two (or more) l-subtree and put the second one (and the ones after, if any) immediately after in the final vtree.

2. The l-subtrees are still constructed and split in the same way, but in the final vtree we order them in ascending order of size.

3. Instead of starting with one l-subtree per agent, we start with two: one containing state and primed state variables, and another containing action variables. We then split any of them whose size exceeds N, and leave them in the same order as agents to construct the vtree.

The first of these three constructions is essentially the one used in the last experiment, which we include here for comparison purposes. The second one will help

us explore whether the l-subtree order matters, and the third will tell us whether re-arranging the variables in a more meaningful way *might* lead to an improvement of the computation time (more meaningful, because a split is less likely to occur at a random place in the variable list, considering we force it to happen between action and state variables).

The results are in Tables 4.11 and 4.12. We can see that none of the new configurations (2. and 3. in the tables) are particularly striking improvements on the one we had before (1. in the table).

Perhaps surprisingly, the "more meaningful" third construction actually seems to make things worse in a majority of cases. We also tried (although without showing the results here) similar constructions where all action l-subtrees are kept at the top, or the bottom of the vtree, rather than next to their corresponding state l-subtrees, as in 3. These were even more inefficient (on our set of models). The conclusion we draw from this is that when considering the bounded vtree construction presented in this section, any reordering set to occur *after* splitting is not particularly efficient; although it may not necessarily lead to worse computation times, in all the cases we tried it never improved them. This could perhaps be explained by the fact that action variables are only used in the SDDs for transition relations, which also use state and primed state variables; not separating them completely might therefore be a better idea.

Overall, it seems that setting an upper bound on the l-subtree size does lead to a much more efficient performance, and this was very clear in all of our experiments. However, reordering l-subtrees after splitting doesn't seem to give particularly interesting results, and is probably not worth it, as it presents a risk of separating variables which would have been more efficient closer together.

Before moving on, we want to insist on the fact that the use of *balanced* l-subtrees seems essential to the efficiency of this vtree construction. Early on in the project, we experimented with right-linear, and left-linear vtrees as l-subtrees; the results were *not* satisfying, and we chose not to include them in this report, in order to keep the focus on the balanced approach.

4.3.3 A Different Partition of the Set of Variables

The objective of this subsection is to explore different *partitions* of the set of variables as bases for the l-subtree construction, diverging from the purely "agent-based" approach taken so far.

Before we propose a replacement partition, we want to make the assumption that we are better off keeping the action variables of an agent close to each other in the vtree, preferably contiguous. This is based on the fact that it is often the case for vtrees produced by the dynamic minimisation algorithm, even though these were completely re-structured; it is for instance true with most agents in Figure 4.1. (Of course, it may well be possible to generate good vtrees without respecting this

	BDDs	Vtree Construction		
		1.	2.	3.
cryptos7	2.64	1.21	1.33	2.26
cryptos8	17.24	4.78	4.09	8.36
cryptos9	98.95	27.51	20.343	42.60
pris9	11.47	32.86	41.14	86.98
pris10	35.11	112.78	105.16	-
nspk1	0.03	0.48	0.46	0.31
nspk2	2.71	20.15	36.675	20.63
nspk4	0.32	5.12	4.94	3.11
nspk5	158.62	859.2	-	1062.52

Table 4.11: Time

	BDDs	Vtree Construction		
		1.	2.	3.
cryptos7	160	56	47	95
cryptos8	647	188	175	350
cryptos9	2850	886	732	1751
pris9	313	1086	844	2796
pris10	847	3464	1919	-
nspk1	13	12	13	9
nspk2	135	370	482	291
nspk4	45	78	120	69
nspk5	3841	6378	-	7849

Table 4.12: Memory

	BDDs		SDDs - New Partition	
	Time	Memory	Time	Memory
cryptos7	2.64	160	1.41	75
cryptos8	17.24	647	5.05	264
cryptos9	98.95	2825	20.36	969
pris9	11.47	313	59.88	1516
pris10	35.11	847	197.95	4341
nspk1	0.03	13	0.57	17
nspk2	2.71	135	83.62	932
nspk4	0.32	45	11.09	219
nspk5	158.62	3841	-	-

Table 4.13: Time and Memory: new partition

assumption. It seems however that this is one principle that the dynamic search agrees with, that we can easily follow, and with which we have obtained good results so far).

So, what are our options? Recall that we still need to find a way of generating the vtree which "makes sense", i.e. which is dependent on the abstract structure of a multi-agent system, rather than on a specific model or example. We propose the following vtree construction:

- We create balanced l-subtrees which contain 2 variables from each agent, a state variable and its primed counterpart;
- We continue until all agents have run out of variables (i.e l-subtrees get smaller as the process goes on);
- For action variables, we create an l-subtree containing *all* the action variables which we append at the bottom of the vtree.

The results, reported in Table 4.13, are very similar to those obtained with the bounded construction in the previous section.

TODO very interesting

4.4 Dynamic Variable Reordering and Minimisation

Dynamic minimisation is very important. In the case of BDDs, it consists in reordering the variables; for SDDs it is about searching for a better vtree – all of this *while* the program is running. Turning on the dynamic variable reordering feature in MCMAS allows for *much larger* models to be verified.

In this section, our objectives are the following:

1. Evaluate the efficiency of the dynamic minimisation feature in the SDD Package, exploring various *initial* vtrees, identifying some of the better ones in the context of model checking;

2. Compare the performance of our own implementation of a dynamic minimisation algorithm (see 3.5.2) against the one included in the SDD Package;

3. Ultimately, compare the performance of our model checker with that of MCMAS when both dynamic minimisation features are enabled.

In addition, we will explore another aspect of dynamic minimisation, namely the *configuration* of the SDD manager, including the thresholds, time and memory limits to be applied in the dynamic vtree search algorithm.

4.4.1 The Default Dynamic Minimisation Function

In this first subsection, we experiment with the dynamic minimisation feature using its default behaviour, while changing the *initial* vtrees (i.e. the vtree the manager was constructed with). Tables 4.14 and 4.15 contain the results obtained with seven different initial options:

- Options 1 to 4: the four standard vtrees, in the order used previously, namely right-linear, balanced, left-linear, vertical. All of these vtrees were constructed with standard ordering 2.

- Option 5: a single balanced l-subtree for each agent (as in 4.3.1), in ascending order of size

- Option 6: the "bounded version" of option 5 (as in **??**)

- Option 7: a single *right-linear* vtree for each agent (non-bounded). This has not been tried before.

Our first observation is the very low memory usage of the model checker in this situation. As previously mentioned, the dynamic minimisation feature triggers the garbage collector, which explains those results. As opposed to all the previous experiments, the memory usage and execution time of the program are now completely unrelated, and one can no longer be "guessed" from the other. From now on, considering the vast majority of our memory results are very acceptable, we will mostly be commenting on the computation times.

Our second observation is that no initial vtree stands out. By this, we certainly do not mean that all of them lead to the same computation time, but rather that it seems to vary a lot depending on the example. Furthermore, we notice a very important irregularity even within the same example class. For the dining cryptographers, an initial right-linear vtree (option 1) seems to make the $n = 12$ version *slower* than the $n = 15$, which is not what is expected.

These inconsistencies are interesting, because they demonstrate, once again, the impact of the vtree and the difficulty in finding a good one. The search algorithm must have been "luckier" in the case $n = 15$ than it was for $n = 12$, or perhaps the "ideal" vtree was "closer" to a right-linear vtree in the first case, so the algorithm got to it faster. (Pardon the extensive use of quotation marks, but omitting them would imply the existence of a such notions, which are probably very subjective.)

This does however raise the question of the *distance* between two vtrees, which seems fundamental in the search for an initial vtree for this particular algorithm which navigates the whole space of vtrees. We will not investigate this further in this project, although it seems that it would be an interesting perspective, and that the *geometry* of the space of vtrees would surely give us useful information on how to navigate it more efficiently.

The final comment we make is that although no vtrees stand out as the best, left-linear vtrees are clearly some of the worst and should probably not be used as initial vtrees. One possible explanation for this is that the number of vtree operations required to obtain an acceptable vtree (similar to the one in Figure 4.1, which is what the algorithm seems to be reaching for) is higher than for other vtrees.

These results are *slower* than those obtained with MCMAS, for all examples. From now on, we only compare different vtrees to each other. We discuss the comparison between both model checkers in 4.4.4.

4.4.2 Changing the Configuration

In this section, we attempt to reduce computation times by exploring alternative *configurations* of the SDD manager. Mainly, this involves modifying thresholds, time and memory limits for vtree operations and minimisation.

Recall that at each step of the dynamic vtree search, the manager attempts to compute 24 alternative vtrees (see 3.1.3 for details), and selects the best one. Out of those 24 vtrees, some might lead to a important increase in the size of the SDD nodes, and checking this might potentially take a long time. For this reason, the SDD Package sets limits on the size increase and time of each vtree operation, to prevent the overhead from being too significant. These size and time limits can be modified by the users, and they will be the focus of our investigation.

Before we start, we inspect the SDD "statistics" provided by the manager at the end of the execution of the program, which indicate the number of vtree operations which have been prevented by the manager. In the vast majority of cases, no operations have been rejected due to them being over the time limit. However, a large number of them (often over 10^5, even for small examples) seem to be turned down because of an important size increase. It therefore seems more important that we explore this aspect of the configuration.

The size limit is supplied as an "increase percentage", so a size limit of 1.1 will prevent the size of an SDD node from growing more than 10%. This is the default

	1	2	3	4	5	6	7
cryptos9	6.62	4.60	12.41	12.96	2.42	1.74	3.17
cryptos12	52.74	3.22	26.2	65.18	6.63	15.21	15.54
cryptos15	19.93	66.07	257.09	-	12.58	21.15	22.35
pris10	22.4	77.33	465.14	48.35	90.8	107.9	32.91
pris11	218.45	115.85	283.67	186.15	103.97	268.32	157.3
pris12	219.98	585.11	983.0	162.6	344.76	290.53	151.26
nspk2	266.8	50.23	181.95	39.08	105.57	181.31	117.79
nspk4	30.31	55.85	75.43	72.51	39.7	70.08	34.56
nspk5	-	-	-	-	-	-	-

Table 4.14: Time

	1	2	3	4	5	6	7
cryptos9	10	12	26	22	4	4	4
cryptos12	87	3	38	36	6	12	8
cryptos15	32	1175	735	-	15	14	15
pris10	8	17	23	15	12	44	10
pris11	34	619	64	25	106	70	15
pris12	31	45	89	40	46	30	23
nspk2	289	62	114	62	60	94	65
nspk4	31	36	39	72	18	32	50
nspk5	-	-	-	-	-	-	-

Table 4.15: memory

	1.01	1.1	1.4
cryptos9	8.30	2.42	3.70
cryptos12	25.43	6.63	11.55
cryptos15	114.169	12.58	19.45
pris10	211.11	90.8	165.84
pris11	335.65	103.97	510.32
pris12	268.24	344.76	613.84
nspk2	99.17	105.57	159.77
nspk4	17.34	39.7	187.65
nspk5	-	-	-

Table 4.16: Time

	1.01	1.1	1.4
cryptos9	68	4	3
cryptos12	68	6	15
cryptos15	399	15	9
pris10	710	12	26
pris11	252	106	29
pris12	132	46	25
nspk2	122	60	50
nspk4	57	18	49
nspk5	-	-	-

Table 4.17: memory

option, and in our experiments we compare this with runs of the model checker with limits of 1.01 and 1.4. These results are presented in Tables 4.16 and 4.17.

The default option is clearly the best one out of those three. It seems that the lower limit prevents so many vtree operations from happening, that the algorithm fails to find an efficient enough vtree. In particular, memory results are very high compared to what we usually obtain when the garbage collector enough. On the other hand, a higher limit seems to waste a lot of time applying numerous vtree operations which, looking at the results, are not worth the time spent.

4.4.3 The New Dynamic Minimisation Function

In order to lower the number of vtrees considered during dynamic minimisation, we had the idea of limiting the vtree operations to the l-subtrees which, as our previous experiments demonstrated, are a very important factor in the overall performance of the model checker. The original motivation was that it would limit the search to vtrees which we *know* are efficient, and thus would avoid wasting time searching for alternative vtrees.

Unfortunately, this part of the project did not go as well as we had hoped, and the performance of our implementation turned out to be extremely bad. The results our not included here as (we believe) this issue requires more work, both on the theoretical and on the implementation side.

The following are likely reasons why our algorithm and implementation failed to improve computations:

- The implementation was not properly synchronised with the default minimisation function of the SDD Package, which is used for minimising l-subtrees. This resulted in our function sometimes being called for minimise l-subtrees themselves, which was not the point. More work would need to be done in order to properly *integrate* our new algorithm into the SDD Package.

- In the CUDD techniques that we were attempting to mimic, groups of variables are created in order to be kept contiguous during the reordering process. Not only does the process then reorder variables within groups (which our algorithm does too), the process may also reorder groups themselves. We did not implement this for l-subtrees as it would have been impossible to use the default minimisation function and a new reordering function would have been needed.

Nonetheless, this idea of grouping variables has proved critical for the efficiency of some BDD reordering techniques, and we believe that there is much room for improvement in this area of SDD minimisation.

4.4.4 Comparison with the MCMAS dynamic minimisation feature

In this final subsection, we compare the results obtained for SDD dynamic minimisation in our model checker with the BDD variable reordering functionality implemented in MCMAS. Table 4.18 contains both the time and memory results of our comparison. The BDD results were obtained by running MCMAS using the standard ordering 2 as the initial variable order. The SDD results do not correspond to one experiment in particular, but instead were taken as the *shortest* of all the computation results obtained previously. Memory and time results do however correspond to the same run of the model checker.

	Time		Memory	
	BDDs	SDDs (min.)	BDDs	SDDs (min.)
cryptos9	0.33	1.74	12	4
cryptos12	0.90	3.22	13	3
cryptos15	7.19	12.58	32	15
pris10	0.61	22.4	32	8
pris11	0.94	103.97	44	106
pris12	1.61	151.26	47	23
nspk2	12.53	39.08	43	62
nspk4	5.68	17.34	20	57
nspk5	-	-	-	-

Table 4.18: Dynamic Comparison with MCMAS (Time and Memory)

Although this is not really a fair comparison, it is enough for us to realise that BDDs *massively* outperform SDDs, when both minimisation features are activated. This could have been expected, as there are two major reasons for this:

- Vtree search is a much more time-consuming process, as seen earlier in this report. It is very likely that the structures obtained after a good vtree search are more compact than the corresponding BDDs, however the difference in size does not make up for the time required for the search – at least in our examples.
- In CUDD are implemented 14 different variable reordering techniques [30]. Some of these were designed specifically for the development of the library, and the rest were taken from the large amount of existing research on BDD minimisation. In comparison, only one SDD minimisation algorithm has been proposed, and it was not designed for model checking but for knowledge compilation, which requires far less operations on SDDs.

4.5 Existential Quantification

In these final experiments, we evaluate our new implementation of the existential quantification of a set of variables, as detailed in 3.4.2. Recall that the default `exists` algorithm in CUDD is recursive, whereas no such implementation exists in the SDD Package, leaving us with the basic method of iterating over the set of variable, which seems like a slower method. For this reason, we implemented the equivalent of CUDD's recursive algorithm in our model checker, hoping to reduce execution times.

Table 4.19 contains the results of our comparisons of both algorithms, with BDDs, and SDDs with both right-linear and "bounded agent-based" vtrees.

	BDDs		SDDs			
			right-linear		bounded agents	
	recurs.	iter.	recurs.	iter.	recurs.	iter.
cryptos8	17.24	18.53	30.51	30.42	10.49	5.70
cryptos9	98.95	109.29	149.08	145.313	53.59	30.46
pris8	2.03	4.05	69.73	64.43	43.67	29.696
pris9	11.47	18.42	-	252.21	-	42.17
nspk2	2.71	8.97	-	76.32	-	27.88
nspk4	0.32	0.91	-	9.79	29.28	3.94

Table 4.19: Comparisons of various exists functions

In the case of BDDs, the recursive algorithm performs slightly better, as we expected. However, the SDD results are rather suprising at first sight, as it is the iterative algorithm which is more efficient in all cases.

We observe that the time difference between the two algorithms is much more marked with the bounded vtrees than with the right-linear, which *confirms* that the unsuitability of this algorithm for SDDs is due to the fact that a "top variable" makes sense in a right-linear vtree but not in a bounded SDD. That is, the order in which variables are existentially quantified makes an important difference in a BDD or a right-linear SDD, but not in other types of SDDs, for which a recursive algorithm is therefore just additional overhead.

In the case of right-linear vtrees, a likely reason for the longer recursive computations is the fact that the `condition` algorithm on SDDs was not optimised for this particular type of vtrees, and therefore probably isn't as efficient as it should be for this type of data structures. This, along with the overhead added by the recursive calls, makes the iterative method preferable even in this case.

4.6 Summary

STATIC - standard not efficient - by agent, better but not so good - bound it to get rid of irregularities, it works well. summarize (1) the order and (2) the bound - try a new partition but didn't find anything as efficient, this is logical because it's not as natural

DYNAMIC - hard to find heuristics because it varies so much - 1.1 size is the best - New grouped implementation does not work well, if we had more time we w -

4.7 Qualitative Evaluation

4.7.1 Pertinence of the Approach

Our theoretical approach identified the key challenges in model checking with SDDs, namely:

- Finding an efficient method for existential quantification of a set of variables on an SDD;
- Identifying appropriate, statically generated vtrees;
- Developing a powerful dynamic vtree search algorithm.

4.7.2 Effectiveness and Elegance of the Implementation

In this subsection, we discuss two aspects of the code: its effectiveness and its elegance. To summarise, our implementation is believed to be effective, but not quite so elegant.

We argue this first point by saying that most of the vtree options used throughout this project were implemented as command line options, following the design of MCMAS. For example,

```
./mcmas_sdd -vtree 1 -o 2 example.ispl
```

will verify the model encoded in the file `example.ispl`, by using a static right-linear vtree (option 1) with standard ordering 2 (the same as MCMAS). Similarly,

```
./mcmas_sdd -vtree 5 -d example.ispl
```

will enable the dynamic minimisation feature (the `-d` option), with an initial "agent-based" vtree (option 5). Running `./mcmas_sdd` without arguments will print out a help screen describing all possible options.

On the other hand, the code base itself is not the most elegant or accessible. Specifically, we believe that our implementation doesn't make enough use of the object-oriented features of C++, in particular during the vtree generation process, which has some duplicate code that could have been avoided, had we created a `vtree_creator` class, and similarly a `var_order_creator` class. Also, the code that we added is well commented, but requires an understanding of the original MCMAS implementation, which itself is not the most documented code base.

4.7.3 Depth of Experiments

During the evaluation period of our project, we chose to explore a broad range of options, in order to give the most accurate judgement of the suitability of SDDs for model checking, and attempt to find alternative techniques to the less efficient ones.

Unfortunately, this prevented us from going into very much depth in each of our experiments, and we were often limited to only trying a small number of different options, or spend too much time investigating the causes of various performance issues. A particular area that we somewhat overlooked is the way memory is managed in the CUDD library and the SDD Package. This is really important, as the *cache* is an essential element to the efficiency of using these data structures in practice. In CUDD, this is relatively well documented, but we did not have access to the SDD Package source code and were forced to use it as a "black box".

Nevertheless, this initial investigation provides a good basis for identifying potential ways of improving the performance of an SDD-based model checker.

4.7.4 Limitations

One limitation of the project is the small variety of examples on which our model checker was tested. Although our three classes of examples were sufficient to identify important flaws and more efficient techniques, a more thorough investigation could be conducted using even more diverse models.

Chapter 5

Conclusion and future work

So far, it seems relatively unlikely that SDDs will replace OBDDs in model checking. However, our project clearly layed out

REVIEW - very interesting project because of its potential - basis for further improvement on SDDs, it is possible! - project is useful for any other use of SDD, perhaps knowledge compilation (although agent-based approaches won't work)

FUTURE WORK: - ADDs - new grouped implementation, being able to specify actual groups - Studying geometry of the space of vtres -

5.1 Review

5.2 Future work

Missing features will be counterexample/witness generation, and checking for deadlock or model overflow. Also being able to check ATL formulas. At some point I would like to have a go at implementing an ADD equivalent for SDDs. -¿ Call `sdd_apply()` on a reduced vtree

Bibliography

[1] E. M. Clarke, E. A. Emerson: *Design and synthesis of synchronization skeletons using branching time temporal logic*, 1981.

[2] E. M. Clarke: *The Birth Of Model Checking*, 2008.

[3] J. P. Queille, J. Sifakis: *Specification and Verification of Concurrent Systems in CESAR*, 1981.

[4] K. L. McMillan, *Symbolic Model Checking: an Approach to the State Explosion Problem*, PhD Thesis, 1992.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang: *Symbolic Model Checking: 10^{20} States and Beyond*, 1990.

[6] A. Lomuscio, F. Raimondi: *Verification of multiagent systems via ordered binary decision diagrams: an algorithm and its implementation*, 2004.

[7] A. Darwiche: *SDD: A New Canonical Representation of Propositional Knowledge Bases*, 2011.

[8] A. Darwiche, A. Choi, Y. Xue: *Basing Decisions on Sentences*, 2012.

[9] A. Darwiche, A. Choi: *Dynamic Minimization of Sentential Decision Diagrams*, 2013.

[10] P. Roux, R. I. Siminiceanu: *Model Checking with Edge Valued Decision Diagrams*, 2010.

[11] A. Darwiche, P. Marquis: *A Knowledge Compilation Map*, 2002.

[12] S. Subbarayan, L. Bordeaux, Y. Hamadi: *Knowledge Compilation Properties of Tree-of-BDDs*, 2007.

[13] K. L. McMillan: *Hierarchical representations of discrete functions, with application to model checking, 1994.*

[14] M. Huth, M. Ryan, *Logic in Computer Science*, Cambridge University Press, 2004.

[15] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*, Addison-Wesley Professional, 2005.

[16] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, F. Somenzi: *Algebraic decision diagrams and their applications*, 1993.

[17] M. Rice, S. Kulhari: *A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction*, 2008.

[18] H. Fujii, G. Ootomo, C. Hori: *Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams*, 1993.

[19] R. Alur, T. A. Henzinger, O. Kupferman: *Alternating-Time Temporal Logic*, 2002.

[20] R. E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*, 1986.

[21] E. A. Emerson, J. Y. Halpern: *"Sometimes" and "not never" revisited: on branching versus linear time temporal logic*, 1986.

[22] R. Fagin, J. Y. Halpern, Y. Moses, M. Y. Vardi: *Reasoning about Knowledge*, 1995.

[23] O. Coudert, J-C. Madre: *A Unified Framework For the Formal Verification of Sequential Circuits*, 1990.

[24] R. Needham, M. Schroeder: *Using Encryption for Authentication in Large Networks of Computers*, 1978.

[25] K. L. McMillan: *Applying SAT Methods in Unbounded Symbolic Model Checking*, 2002.

[26] M. Sheeran, S. Singh, G. Stalmarck: *Checking Safety Properties Using Induction and a SAT-Solver*, 2000.

[27] A. Biere, A. Cimatti, E. Clarke, Y. Zhu: *Symbolic Model Checking without BDDs*, 1999.

[28] M. Chechik, A. Gurfinkel, B. Devereux, A. Lai, S. Easterbrook: *Data Structures for Symbolic Multi-Valued Model-Checking*, 2006.

[29] MCMAS webpage, http://vas.doc.ic.ac.uk/software/mcmas/

[30] CUDD Package webpage, http://vlsi.colorado.edu/ fabio/CUDD/

[31] SDD Package webpage, http://reasoning.cs.ucla.edu/sdd/

[32] David A. Wheeler's MiniSAT solver, http://www.dwheeler.com/essays/minisat-user-guide.html

[33] Graphviz DOT, http://www.graphviz.org/Documentation.php

[34] A. Lomuscio, Software Engineering: Software Verifications, lecture notes (as taught in 2012-13)

[35] I. Hodkinson, Modal and Temporal logic, lecture notes (as taught in 2013-14)

Appendix A

The Bit Transmission Problem

A.1 The Problem

The Bit Transmission Problem is the following situation: a sender (S) wants to send a one-bit message to a receiver (R) through a communication channel. However, the channel is faulty, and might drop the bit (though it will never flip it).

As soon as R receives the bit, it starts sending an acknowledgement message to S, which the channel might also drop, and continues sending it forever. S keeps sending the message until it receives an acknowledgement from R.

A.2 The Model

Observe that the BTP is a multi-agent systems, consisting of two agents (the sender and the receiver) acting within an environment (the channel). We can therefore model the BTP as an interpreted system.

The agent S has local states $L_S = \{(b0, noack), (b1, noack), (b0, recack), (b1, recack)\}$, where $b0, b1$ are variables representing the message that S wants to send (i.e. either $b0$ or $b1$), *noack* is true iff S hasn't received the acknowledgement from R, and *recack* is true as soon as it has received it. S has set of actions $\mathrm{Act}_S = \{sendb0, sendb1, nothing\}$, and protocol function

$$\begin{aligned} P_S : L_S &\longrightarrow Act_S \\ (b0, noack) &\longmapsto sendb0 \\ (b1, noack) &\longmapsto sendb1 \\ (b0, recack) &\longmapsto nothing \\ (b1, recack) &\longmapsto nothing. \end{aligned}$$

Similarly, the agent R has local states $L_R = \{r0, r1, empty\}$, where the initial state of R is *empty*, and $r0, r1$ represent the bit that R received, once it has received it. S has set of actions $\mathrm{Act}_R = \{sendack, nothing\}$, and protocol function

$$\begin{aligned} P_R : L_R &\longrightarrow Act_R \\ \text{empty} &\longmapsto nothing \\ r0 &\longmapsto sendack \\ r1 &\longmapsto sendack. \end{aligned}$$

Finally, the Environment represents the faulty channel, which can either transmit messages both ways, one way, or not at all. Hence

$$L_E = \{(\rightarrow, \leftarrow), (\rightarrow, \not\leftarrow), (\not\rightarrow, \leftarrow), (\not\rightarrow, \not\leftarrow)\}.$$

The only actions of the channel are to transmit the messages, when it does. We use the same notation as for its states, and define

$$\mathrm{Act}_E = \{(\rightarrow, \leftarrow), (\rightarrow, \not\leftarrow), (\not\rightarrow, \leftarrow), (\not\rightarrow, \not\leftarrow)\},$$

where for example $(\rightarrow, \not\leftarrow)$ represents the action of sending a message from S to R but not the other way around.

The channel has an undeterministic behaviour, so its protocol $P_E : L_E \longrightarrow 2^{Act_E}$ is defined to return any possible action.

A.3 ISPL Specification

TODO write something about this

```
Agent Environment
  Vars:
    state : {S,R,SR,none};
  end Vars
  Actions = {S,SR,R,none};
  Protocol:
    state=S: {S,SR,R,none};
    state=R: {S,SR,R,none};
    state=SR: {S,SR,R,none};
    state=none: {S,SR,R,none};
  end Protocol
  Evolution:
    state=S if (Action=S);
    state=R if (Action=R);
    state=SR if (Action=SR);
    state=none if (Action=none);
  end Evolution
end Agent
```

Figure A.1: The ISPL code for the Environment.

```
Agent Sender
  Vars:
    bit : { b0, b1}; -- The bit can be either zero or one
    ack : boolean; -- This is true when the ack has been received
  end Vars
  Actions = { sb0,sb1,nothing };
  Protocol:
    bit=b0 and ack=false : {sb0};
    bit=b1 and ack=false : {sb1};
    ack=true : {nothing};
  end Protocol
  Evolution:
    (ack=true) if (ack=false) and
       ( ( (Receiver.Action=sendack) and (Environment.Action=SR) )
        or
       ( (Receiver.Action=sendack) and (Environment.Action=R) )
       );
  end Evolution
end Agent

Agent Receiver
  Vars:
    state : { empty, r0, r1 };
  end Vars
  Actions = {nothing,sendack};
  Protocol:
    state=empty : {nothing};
    (state=r0 or state=r1): {sendack};
  end Protocol
  Evolution:
    state=r0 if ( ( (Sender.Action=sb0) and (state=empty) and
                  (Environment.Action=SR) ) or
                ( (Sender.Action=sb0) and (state=empty) and
                  (Environment.Action=S) ) );
    state=r1 if ( ( (Sender.Action=sb1) and (state=empty) and
                  (Environment.Action=SR) ) or
                ( (Sender.Action=sb1) and (state=empty) and
                  (Environment.Action=S) ) );
  end Evolution
end Agent
```

Figure A.2: The ISPL code for the two agents of the system, Sender and Receiver.

```
Evaluation
  recbit if ( (Receiver.state=r0) or (Receiver.state=r1) );
  recack if ( ( Sender.ack = true ) );
  bit0 if ( (Sender.bit=b0));
  bit1 if ( (Sender.bit=b1) );
  envworks if ( Environment.state=SR );
end Evaluation

InitStates
  ( (Sender.bit=b0) or (Sender.bit=b1) ) and
  ( Receiver.state=empty ) and ( Sender.ack=false) and
  ( Environment.state=none );
end InitStates

Fairness
  envworks;
end Fairness

Formulae
  AF(K(Sender,K(Receiver,bit0) or K(Receiver,bit1)));
  AG(recack -> K(Sender,(K(Receiver,bit0) or K(Receiver,bit1))));
end Formulae
```

Figure A.3: The rest of the ISPL code of the BTP, including the initial states, the fairness condition, the formulae to be checked, and the evaluation function for atoms of the model.

Appendix B

Implementation Details

Code:

- the SDD params structure
- the compute reach function
- the getvarorder and createvtree functions