Contents

Abstract

1 Introduction

1.1 The Problem

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. [finir le paragraphe avec des exemples]

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 location.

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. These include automated theorem proving, TODO, 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).

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 much larger 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, it is

1.2 The Idea

In this project we propose an alternative to OBDDs for symbolic model checking: another representation of Boolean functions

1.3 The Deliverables

- Autonomous systems, what they are, why they are useful, and why they are vulnerable but we need to make sure they are reliable.
- Verifying autonomous systems: what it entails, how we go about it (methods) and what are the challenges
- The state explosion problem
- Our idea and contributions
- structure of the report

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(deadlocked)

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

- GF(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:

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

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 [17] 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 [18] 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.

Suppose that we want to check that a multi-agent system Σ satisfies a propery 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 $I \subseteq [\![\varphi]\!]$. The following algorithm returns $[\![\varphi]\!]$ for any CTLK formula φ.

```
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 = K_iφ1: return SAT_K(i, φ1)
if φ = Eφ1: return SAT_E(φ1)
if φ = Cφ1: return SAT_C(φ1)
end
```

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 $g_0, g_1, ...$).

```
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(φ1 U φ2)]]
  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 [[Ki φ]]
  X := SAT(¬φ)
  Y := {g0 ∈ G | ∃g1 ∈ X with g0 ~i g1}
  return G \ Y
end

function SAT_E(φ)
// returns [[E φ]]
  X := SAT(¬φ)
  Y := {g0 ∈ G | ∃g1 ∈ X with g0 ~i g1 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 := {g0 ∈ G | ∃g1 ∈ X with g0 ~i g1 for all i = 1, ..., n}
  end
  return G \ Y
end
```

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

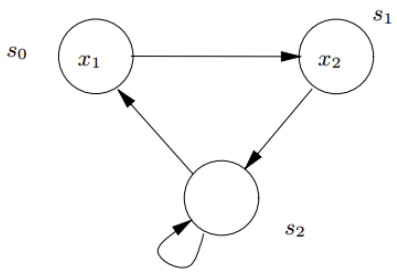

Figure 1: A model

obtained by computing the fixed point of the transition relation function, as shown in the algorithm below:

```
function compute_reach()
// returns the reachable state space G
  X := ∅
  Y := I
  repeat until X = Y
    X := Y
    Y := Y ∪ {g1 ∈ G | g0 → g1 for some g0 ∈ X}
  end
  return Y
end
```

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

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 [11] to illustrate the encoding process. Consider the model in Figure 1, 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:

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

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,

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

and

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

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.

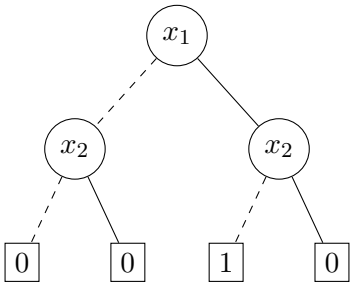

Figure 2: BDT for $f(x_1, x_2) = x_1 \wedge \neg x_2$ under the ordering $[x_1, x_2]$

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 $\{0,1\}^n \to \{0,1\}$ 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 of the variable the node is labelled with). An example is shown in Figure 2. 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. Fortunately, a BDT can be reduced to an *ordered binary decision diagram* (OBDD), a much more compact data structure.

The Reduction Algorithm

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

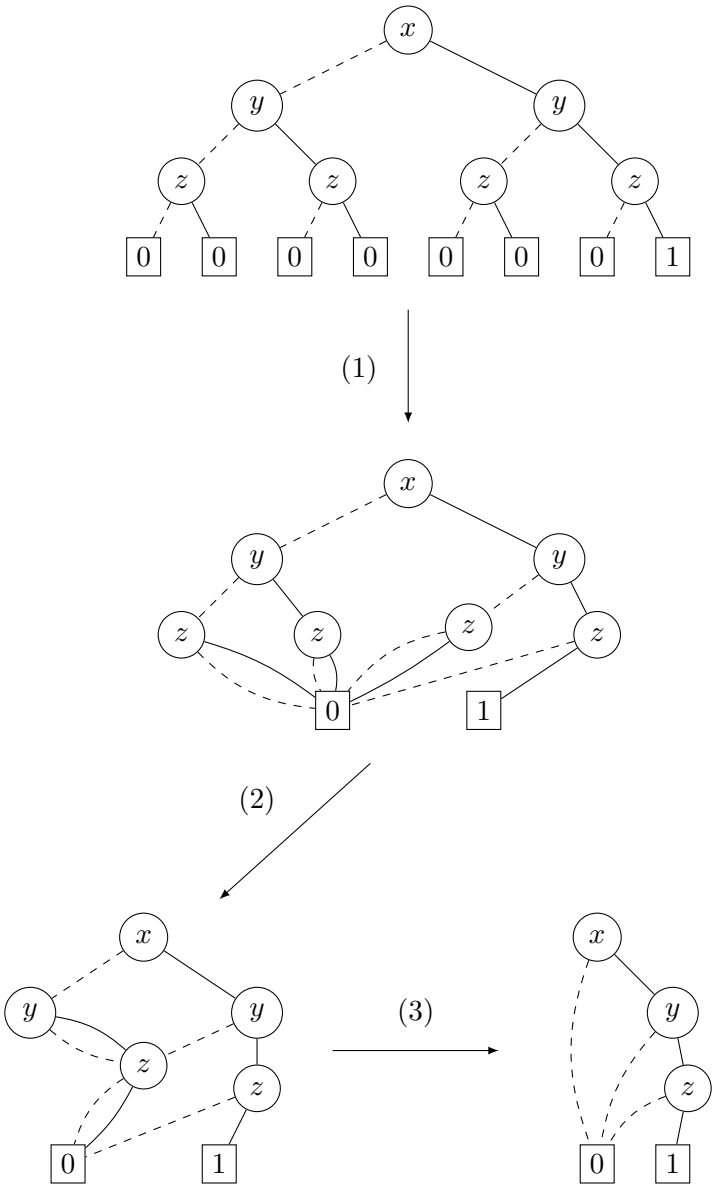

Figure 3: Reduction to the OBDD for function $f(x, y, z) = x \wedge y \wedge z$ with variable order $[x, y, z]$. In this case, applying Steps (2) and (3) once is sufficient as the resulting diagram is reduced.

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.

The following key result makes the use of OBDDs viable in practice:

Theorem 2.1 *[16] If f is a Boolean function over the variables $x_1, ..., x_n$, then the OBDD representing f is unique, up to the order of $x_1, ..., x_n$ chosen.*

The immediate consequence of Theorem 2.1 is that one can easily compare two Boolean functions by comparing their respective OBDDs (provided both OBDDs have the same variable order).

Another important observation to make is that OBDDs 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.

OBDDs 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: throughout this report we often drop the 'O' and refer to OBDDs 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 [8], a comparative analysis of most

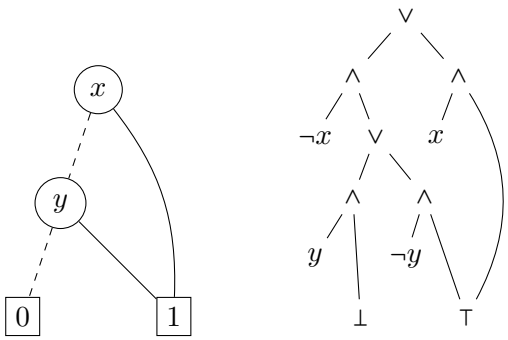

Figure 4: An OBDD (left) and the corresponding NNF sentence (right). Although the former seems much more compact, the difference in size is only linear and both representations are essentially the same.

existing DAG-based target compilation languages in terms of their succintness and the polytime operations they support (TO DO check que ya vmt les operations). 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 4 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.

After considering a number of different representations including tree-of-BDDs [9], BDD-trees [10], and several other subsets of NNF described in [8], 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 [5] and [7] 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 [5], 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.

Notation: we also write $f(X,Y)$ if $f(Z)$ and 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 2.2 *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.

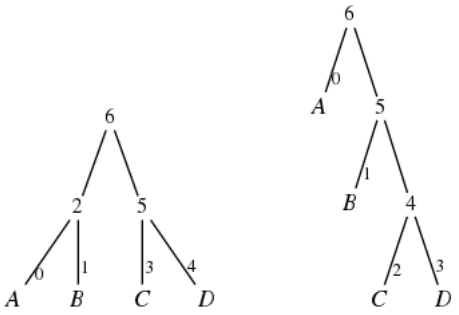

Figure 5: Two vtrees for $X = \{A, B, C, D\}$.

As we see later on, an important consequence of Theorem 2.2 is the polynomial time operations available on SDDs. Canonicity of SDDs is due to the following result:

Theorem 2.3 *A function $f(X, Y)$ has exaclty one compressed X-partition.*

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.

Note that a vtree on a set X is stronger than a total order of the variables in X. Figure 5 shows two distinct vtrees which induce the same variable order.

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 5, 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 6. On the right is the SDD for f constructed with respect to the vtree on the right in Figure 5.

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.

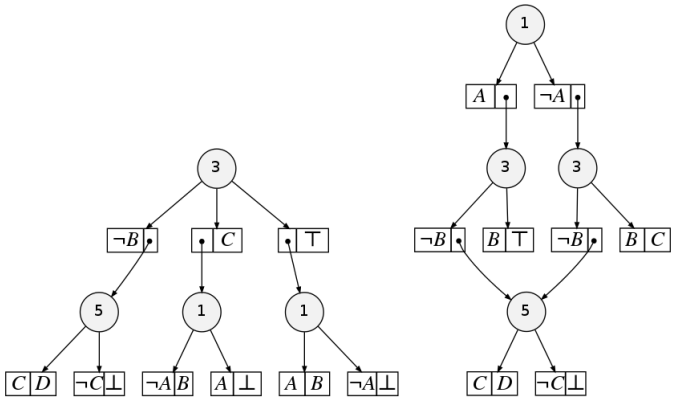

Figure 6: SDDs for $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ corresponding to the vtrees in Figure 5. Notice that identical SDD nodes have been merged.

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 *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 2.6 below shows that they are in fact sufficient for the representation to be canonical. To prove it we first need another lemma.

Lemma 2.5 *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 2.6 *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). Then by Lemma 2.5, 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 2.3, 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.

```
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
```

If α and β are compressed, then this algorithm returns a compressed SDD for $\alpha \circ \beta$. Theorem 2.2 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.

The `apply` algorithm to compose SDDs α and β takes time $O(|\alpha||\beta|)$, which implies that conjunction, disjunction and negation of SDDs can all be done in polynomial time.

To do: Need to write something about the `condition()` algorithm which comes up a lot in model checking.

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 one-to-one correspondence between BDDs and SDDs respecting right-linear vtrees. Figure 7 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

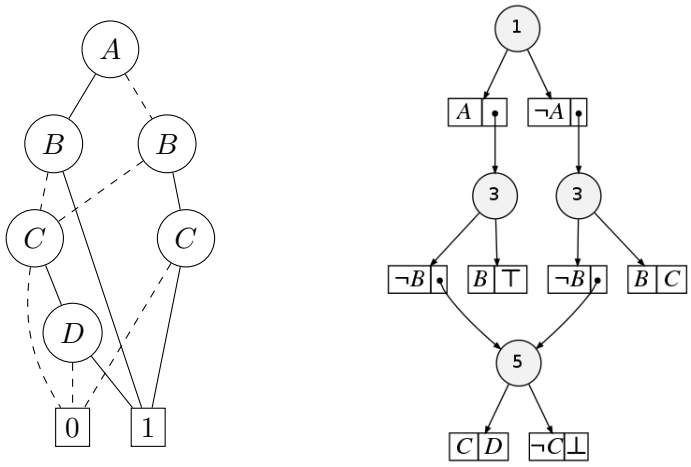

Figure 7: The BDD for $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ with variable order $[A, B, C, D]$ (left), and the SDD respecting the right-linear vtree inducing the same order (right). Note that this is the vtree depicted on the right of Figure 5.

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!).

3 A First Model Checker Based on SDDs

This chapter is devoted to the implementation of the model checker itself. We start by giving an overview of the existing code base and libraries upon which it relies, before presenting the important design decisions and algorithms, and finally discussing the challenges that occured during the implementation of this new model checker.

3.1 Preliminary I: The SDD Package

3.1.1 Description

The SDD Package 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.

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 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 8.

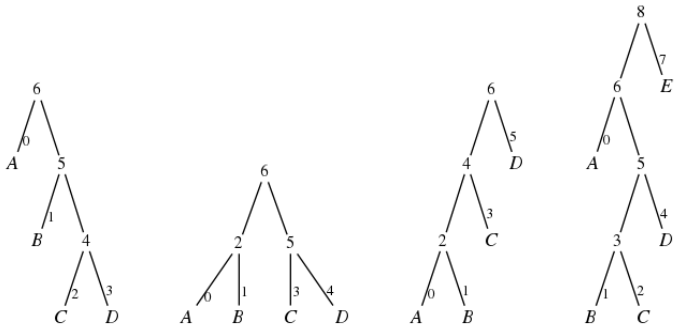

Figure 8: The four standard vtrees, in order: right-linear, balanced, left-linear, and vertical.

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 [7] for more efficient dynamic minimisation of SDDs. This algorithm relies on three standard binary tree operations: left-rotation, right-rotation, and swap; these operations are sufficient for navigating the space of all vtrees [12]. The following example shows in what way these operations affect vtrees:

Example take a vtree with 4 nodes

We now give an informal description of the algorithm in [**?**] (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. It then considers two vtree fragments of v, the *l-vtree* and the *r-vtree*. As shown on Figure **??**, each of these is a binary tree with 3 leaves (which might not be leaves in v). 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 best 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 `exist()` function which relaxes the constraint on a variable x in a function f, returning $f|_x \vee f|_{\neg x}$. To do write more about exist

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). 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 an interpreted system encoded in ISPL: the *Bit Transmission Protocol*, available in the appendix. ISPL is a fairly straightforward language, we therefore will not be spending any more time describing it here, but should anything remain unclear, the reader is refered the MCMAS User Manual [**?**].

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 vs. SDDs comparisons in the next chapters so it is important that the reader 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 [15], 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, [13]) 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.

3.3.2 The Importance of the Vtree

We have seen that the size of an SDD can be significantly affected by the vtree with respect to which is is constructed. However, it turns out that the vtree also 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`).

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.

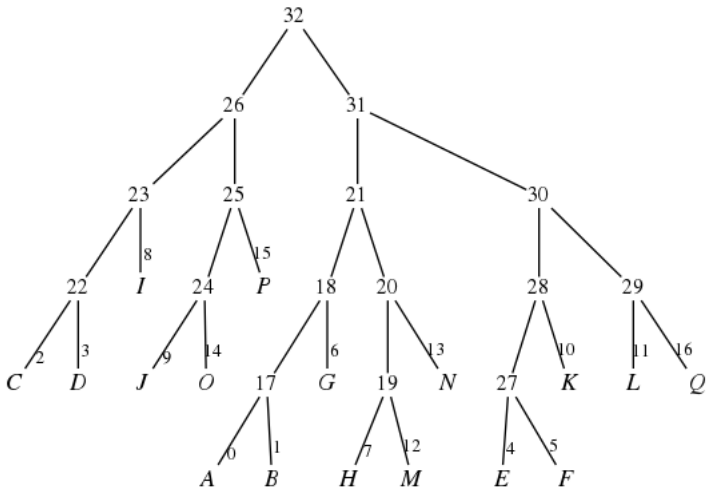

Figure 9: The l-subtrees of this vtree are the subtrees rooted at nodes 26, 21, 28, and 29.

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 9. 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 meaningful, easy to generate 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 evaluated in a later chapter.

3.3.5 Dynamic Minimisation in Multi-Agent Systems

Dynamic minimisation is an essential element to the use of symbolic model checking in practice. When state spaces grow larger,

In 3.1.3 we have described the dynamic vtree search algorithm implemented in the SDD Package.

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`.

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 [19]:

```
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
```

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:

```
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 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

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

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.

In an SDD however, this is not so obvious:

to do: talk about `sdd_condition()` and the performance of our implementation.

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 explained in the evaluation section of this report, these 16 vtrees quickly proved insufficient for our purposes, and being able to create our own "non-standard" vtrees became crucial for the progress 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. To create our own customised vtrees, we had the two following 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 found to have more potential in the context of model checking. We leave out the details for now and use the evaluation section to describe the process leading to their discovery.

3.5.2 A New Dynamic Vtree Search Algorithm

In 3.1.3, we described the dynamic minimisation algorithm implemented in the SDD Package. This algorithm is focused on a complete restructuring of the manager's vtree in order to minimise the size of its SDD nodes. Early experiments with this algorithm *sometimes* revealed large overheads due to the difficulty of searching the huge space of vtrees. We propose a new approach to vtree search adapted to multi-agent systems in those cases where the original approach is not efficient. In this section we focus on the implementation of this new approach, while a detailed description, and a practical comparison of both techniques is detailed in the evaluation section (**??**).

The new approach to vtree search is analogous to some dynamic variable reordering techniques for BDDs, which involve defining *groups* of variables, set to remain contiguous during the reordering process. Using those techniques, reordering can then be applied within each group, and groups themselves can be reordered.

We propose a comparable algorithm for vtree search where the variables for individual agents are grouped together into sub-vtrees, and the minimisation function is called on each of these separately. 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 implementation 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 doum one step to the right
    current = sdd_vtree_right(current);
  }
  sdd_vtree_minimize(current);
  return vtree;
}
```

This algorithm is appropriate for vtrees of a particular shape, which will constitute the focus our study in most of the evaluation section of this report. Without revealing too much information at this point, we simply say that these vtrees are built so that variables are separated into smaller groups, each group forming a left "branch" in the vtree. Figure **??** illustrate this with an example, showing the effect of `vtree_group_minimize()` on a particular vtree.

3.6 Software Engineering Issues and Challenges

Here we explain some of the challenges faced during the implementation 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 [23]), 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 [24]. 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.

4 Evaluation (?)

4.1 Introduction

4.1.1 Evaluation Strategy

The ultimate objective of this project was either to establish, or to refute SDDs (in their current state) as serious competitors to BDDs in model checking. The obvious plan for quantitative analysis was 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 [14]. 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

- Bit transmission problem
- Dining Cryptographers
- Prisoners' dilemma
- NSPK - http://en.wikipedia.org/wiki/Needham-Schroeder_protocol

also: choice of examples, why parametrised, describe the state spaces of each

4.1.3 Experimental Protocol

Throughout this chapter, we report a large number of tables containing experimental results. In this section 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 ...

Most results are split into two tables: time and memory; time measurements are in seconds, memory usage in megabytes.

In each table we indicate in bold the situations where SDDs outperform BDDs.

in most cases we set parameters to be the most appropriate

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. As automatic garbage collection was disabled in

We started with right-linear vtrees.

4.2.1 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 are handling data structures of the exact same size.

Tables 1 and 2 correspond, respectively, to the time and memory comparisons of our model checker with MCMAS, when SDDs are built using right-linear vtrees inducing the standard orderings 1, 2, 3 and 4.

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

We observed one thing in particular. Despite the SDD/BDD ratio being relatively consistent within each class of examples, it varies a lot more across the different classes. Going back to the characteristics of these classes, we realise that this ratio increases with the size of the model

To conclude this first set of experiments, we make the following remark: in this situation the model checkers are dealing with the *exact* same structures, so these numbers

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

	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 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: Memory with left-linear vtrees.

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 1 and 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 3 and 4, even on the small examples chosen they were significantly slower than BDDs. Furthermore, although the variable ordering chosen for the vtrees seems to have a big impact on the computation (much bigger in fact than on BDDs), even for 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 computations times than BDDs, or even right-linear vtrees. This results appear very clearly in Tables 5 and 6.

	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	11.25	-	1063.2	-	-
nspk4	1.29	-	18.33	88.12	-

Table 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 6: Balanced Vtrees: Memory Usage

4.2.3 Towards a Better Vtree: Observations

Upon experimenting with all the 16 standard vtrees, it was clear that if SDDs were ever to be used in model checking, there would need to be a way of generating better vtrees. In this section we provide

Analysis of an Example

In order to get a better idea of the type of vtree which would make the computation more efficient, we ran the model checker on a set of examples using the dynamic minimisation feature, and we looked at the final vtree that the search algorithm produced (i.e. the manager's vtree immediately before the process terminated). The result was unambiguous: in *all* cases, independently of the initial vtree, the final vtree had the same shape, a sort of right-linear vtree where the left branches contain a set of variables rather than single variable. For this vague description to make more sense, we illustrate it here with an example, shown in Figure 10. We make a few remarks about the shape of this vtree (without examiniming the variable order yet):

- The vtree in Figure 10 is *typical*, and an absolutely fair representative of the vtrees obtained after dynamic minimisation (whatever the example, and whatever the initial vtree).
- Recall that 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 shape in all cases is evidence that it is in fact the *ideal* configuration.
- One of the left-branches in the middle is very large – it contains 30 variables, over a third of the total. This shows that smaller subs are not *necessary* for an efficient computation, and that larger sets of variables may in fact present an improvement.
- It tends to happen (and it is definitely the case in Figure 10) that the first few left-branches are slightly smaller than the ones below. A reasonable explanation for this is that the first few left-branches correspond to the subs in the first few levels of the corresponding SDD (starting from the root), i.e the subs which need to be evaluated most often.

Vtrees of this shape were the basis of the next set of experiments we conducted in static generation; these are detailed in Section 4.3.

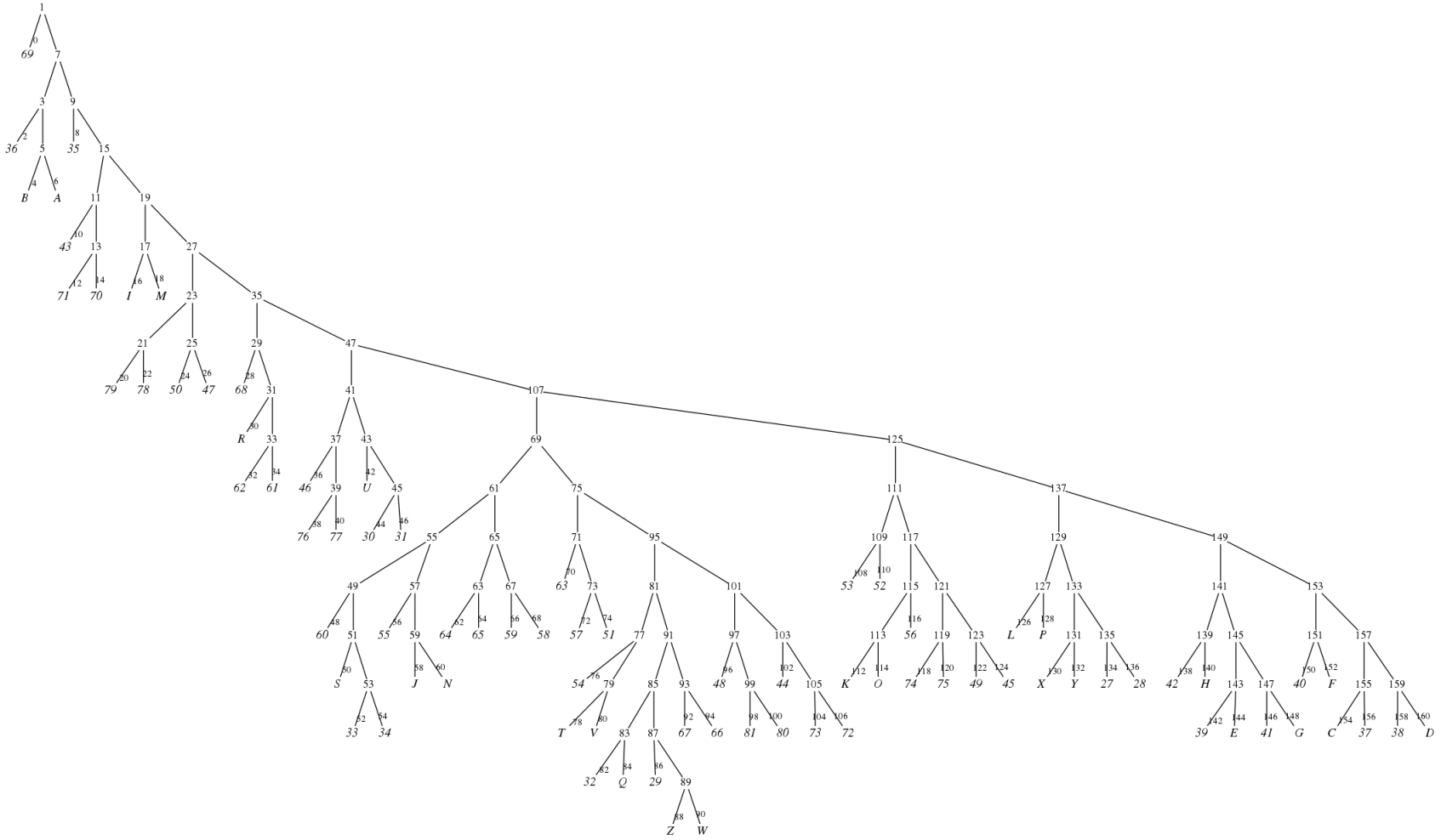

Figure 10: The manager's vtree at the end of a run of our model checker on crypto6, with an initial vtree of type balanced.

	BDDs	Vtree Option 5		
		Agent	Inverse	Ascending
cryptos4	0.02	2.41	1.31	1.31
cryptos5	0.06	31.32	32.08	32.08
cryptos6	0.44	993.71	-	-
pris6	0.15	7.43	3.94	2.52
pris7	0.57	48.01	21.36	10.22
pris8	2.08	303.73	96.39	48.76
nspk1	0.03	1.85	2.01	2.12
nspk2	11.25	542.32	-	-
nspk4	1.29	68.95	54.69	55.65

Table 7: Vtree Option 5 – Time

	BDDs	Vtree Option 5		
		Agent	Inverse	Ascending
cryptos4	11	59	44	44
cryptos5	18	673	574	574
cryptos6	45	8716	8241	8241
pris6	31	76	45	52
pris7	48	316	141	158
pris8	102	1558	483	744
nspk1	16	27	29	25
nspk2	324	2949	-	-
nspk4	92	685	884	901

Table 8: Vtree Option 5 – Memory

4.3 Static Comparisons with Alternative Vtrees

4.3.1 A First Attempt

4.3.2 An Upper Bound on Subtree Size

important to explain why this improves computation times

4.3.3 Various Vtree Characteristics and Their Impact

- changing the way variables are allocated to subtrees
- changing variable order *within* subtrees and subtree type
- changing subtree order, and possibly adapting compute_reach algorithm
-

We aim to find an initial vtree leading to faster computations. We notice that whatever the initial vtree, dynamic reduction algorithms always result in a particular type of vtree, which we call *pseudo-right-linear*. All our experiments are with pseudo-right-linear vtrees.

- vtree experiment 1 (option 5): one balanced subtree per agent.

  This does *not* work well.

- vtree experiment 2 (option 6): We set a maximum size for agent subtrees. An upper bound of $\log_2(n^2)$ (where n is the number of vars) has proved relatively efficient. If an agent has more variables then we create more subtrees for it. Variables of a subtree come from the same agent. Subtrees are balanced, and we experiment with different orderings for them. We observe that the best results are obtained when two principles are followed: action variables are close to each other in the subtree (i.e they alone form a balanced vtree of size `action_count`) and states variables are paired with their primed counterpart. (TODO: confirm this!)

- vtree experiment 3 (option 7): Each subtree contains one variable for each agent, as well as its corresponding primed variable. Action variables for each agent form their own subtree. We order state subtrees 'largest to smallest' and put action subtrees at the bottom. This seems to be the best ordering but TODO need to try: intercaler actions/states.

- vtree experiment 4 (option 8): Think of something!

4.4 Using Dynamic Reordering and Minimisation

4.4.1 Initial Observations

Raw data with all examples and stuff

4.4.2 Experiments

-
-

4.4.3 Grouping Variables

new vtree search method

4.5 Summary

review of best techniques and heuristics

4.6 Qualitative Evaluation

Do SDDs compare with BDDs?

5 Conclusion and future work

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

References

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

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

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

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

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

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

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

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

[13] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

A The Bit Transmission Problem

A.1 The Problem

A.2 The Model

A.3 ISPL Specification

B Implementation Details

Code:

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