# Contents

## Abstract

# 1 Introduction

## 1.1 The Problem

We are becoming increasingly dependent on computer systems. Not only are they an essential part of our lives, ...vital!

## 1.2 The Idea

## 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 $\Sigma$ consisting of $n$ agents $1, ..., n$ and an environment $e$.

**Definition** An *Interpreted System IS* for $\Sigma$ 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 $\Sigma$ 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 $g'$.

- $\pi : PV \rightarrow G$ 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:

$$\varphi \coloneqq p \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid G\varphi \mid \varphi U \varphi$$

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

- $\varphi$ holds at the ne**X**t time instant
- $\varphi$ holds forever (**G**lobally)
- $\varphi$ holds **U**ntil $\psi$ holds

We define the unary operator $F$ to be the dual of $G$, i.e $F\varphi \coloneqq \neg G \neg \varphi$ for any LTL formula $\varphi$. $F\varphi$ represents the idea that $\varphi$ 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 $\rho^i$ the suffix of $\rho$ starting at $i$ (note that $\rho^i$ is itself a path since $\rho$ 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 $\varphi_P$ and $S$ as a model $M_S$, then we need to be able to check whether $\varphi_P$ 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 $\varphi$ and $\psi$, a model $M$ and a state $s_0 \in W$, we say that

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

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:

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

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, $\varphi$ is true at the next state, $\varphi$ holds forever in the future, and $\varphi$ holds until $\psi$ 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, $\varphi$ 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 $\varphi$ and $\psi$, 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:

$$\begin{array}{rcl}
(M, s_0) \models p & \Leftrightarrow & s_0 \in \pi(p) \\
(M, s_0) \models \neg\varphi & \Leftrightarrow & (M, s_0) \not\models \neg\varphi \\
(M, s_0) \models \varphi \wedge \psi & \Leftrightarrow & (M, s_0) \models \varphi \text{ and } (M, s_0) \models \psi \\
(M, s_0) \models EX\varphi & \Leftrightarrow & \exists \text{ a path } s_0, s_1, s_2, ... \text{ such that } (M, s_1) \models \varphi \\
(M, s_0) \models EG\varphi & \Leftrightarrow & \exists \text{ a path } s_0, s_1, s_2, ... \text{ such that } (M, s_i) \models \varphi \quad \forall i \geq 0 \\
(M, s_0) \models E(\varphi U\psi) & \Leftrightarrow & \exists \text{ a path } s_0, s_1, s_2, ... \text{ for which } \exists i \geq 0 \text{ such that } (M, s_i) \models \psi \\
 & & \quad \text{and } (M, s_j) \models \varphi \quad \forall 0 \leq j < i
\end{array}$$

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

- From any state it is possible to reboot the system.
  $AGEX$(reboot)    (not expressible in LTL)
- example 2

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 TODO 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 add 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 $\varphi$, 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 $\varphi$ 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:

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

## 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 $\varphi_P$.

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.

### 2.2.1 Explicit Model Checking

In this section we introduce a basic approach to model checking, the so-called *explicit* approach.

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

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

```
function SAT($\varphi$)
// returns $[\varphi]$
if $\varphi = \top$: return $G$
if $\varphi = \bot$: return $\varnothing$
if $\varphi = p$: return $\pi(p)$
if $\varphi = \neg\varphi_1$: return $G \setminus$SAT($\varphi_1$)
if $\varphi = \varphi_1 \wedge \varphi_2$: return SAT($\varphi_1$)$\cap$SAT($\varphi_2$)
if $\varphi = EX\varphi_1$: return SAT$_{\texttt{EX}}$($\varphi_1$)
if $\varphi = AF\varphi_1$: return SAT$_{\texttt{AF}}$($\varphi_1$)
if $\varphi = E(\varphi_1 U \varphi_2)$: return SAT$_{\texttt{EU}}$($\varphi_1, \varphi_2$)
if $\varphi = K_i\varphi_1$: return SAT$_{\texttt{K}}$($i, \varphi_1$)
if $\varphi = E\varphi_1$: return SAT$_{\texttt{E}}$($\varphi_1$)
if $\varphi = C\varphi_1$: return SAT$_{\texttt{C}}$($\varphi_1$)
end
```

Notice that this covers all formulae $\varphi$, 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$_{\texttt{EX}}$($\varphi$)
// returns $[EX\varphi]$
  X := {$g_0 \in G \mid g_0 \to g_1$ for some $g_1 \in$SAT($\varphi$) }
  return X
end

function SAT$_{\texttt{AF}}$($\varphi$)
// returns $[AF\varphi]$
  X := $G$
  Y := SAT($\varphi$)
  repeat until X = Y:
    X := Y
    Y := Y $\cup$ $\{g_0 \in G \mid$ for all $g_1$ with $g_0 \to g_1$, $g_1 \in$ Y $\}$
  end
  return Y
```

```
end

function SAT_EU($\varphi_1$, $\varphi_2$)
// returns $[E(\varphi_1 U \varphi_2)]$
  W := SAT($\varphi$)
  X := $G$
  Y := SAT($\psi$)
  repeat until X = Y:
    X := Y
    Y := Y ∪ (W ∩ $\{g_0 \in G \mid \exists g_1 \in$ Y such that $g_0 \to g_1\}$)
  end
  return Y
end

function SAT_K($i$, $\varphi$)
// returns $[K_i \varphi]$
  X := SAT($\neg\varphi$)
  Y := $\{g_0 \in G \mid \exists g_1 \in$ X with $R_i(g_0, g_1)\}$
  return $G \setminus$ Y
end

function SAT_E($\varphi$)
// returns $[E\varphi]$
  X := SAT($\neg\varphi$)
  Y := $\{g_0 \in G \mid \exists g_1 \in$ X with $R_i(g_0, g_1)$ for all $i = 1, ..., n\}$
  return $G \setminus$ Y
end

function SAT_C($\varphi$)
// returns $[C\varphi]$
  X := $G$
  Y := SAT($\neg\varphi$)
  repeat until X = Y:
    X := Y
    Y := $\{g_0 \in G \mid \exists g_1 \in$ X with $R_i(g_0, g_1)$ for all $i = 1, ..., n\}$
  end
  return $G \setminus$ Y
end
```

TODO write something about $R_i$.

The complexity of SAT is linear in the size of the model. However, the size of the model grows exponentially in the number of variables used to describe the system $\Sigma$, 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.*

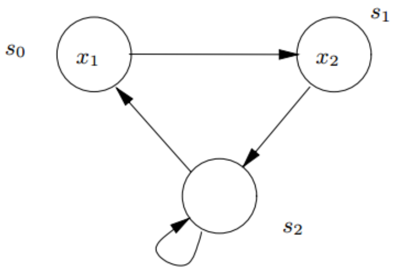

Figure 1: A model

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

The encoding process is best illustrated by an example [7] 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 → 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. In the example above, the transition relation → of the model is $\{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_2)\}$. Using our Boolean representations for $s_0, s_1$, and $s_2$, we compute the representation of → to be

$$(\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge \neg x_2') \vee (\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2') \vee (x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \vee (\neg x_1 \wedge x_2 \wedge \neg x_1' \wedge \neg x_2')$$

[needs more details on the computation]

### The case of Multi-Agent Systems

need to explain this

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 a theory of representations of Boolean functions using directed acyclic graphs (DAG).

### 2.3.1 Ordered Binary Decision Diagrams

After truth tables, one the most basic ways of representing Boolean functions 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 changes 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 will have $2^{n+1} - 1$ nodes. Fortunately, a BDT can be reduced to an *ordered binary decision diagram* (OBDD), a much more compact data structure.

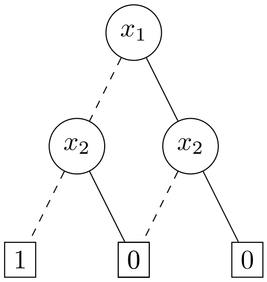

Figure 2: BDT for $f(x_1, x_2) = x_1 \wedge \neg x_2$ under the ordering $[x_1, x_2]$

### 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 $m$
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. The resulting diagram is said to be a ordered binary decision diagram.

[here need example of using reduction algorithm]

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

**Theorem 2.1** *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 A Knowledge Compilation Map

In computer science, the field of *knowledge compilation* is concerned with finding compact and efficient representations of propositional knowledge bases (such as symbolic representations of state-spaces, cf 2.2.2). Such a representation is refered to as a *target compilation language*, of which BDTs and OBDDs are examples.

The main properties that we look for in a target compilation language are the canonicity of the representation, the succinctness of the representation, and a polynomial-time complexity for queries and operations on the language. These properties will ensure that we can safely rely on a particular language for our model checking algorithm.

OBDDs satisfy these properties (with succintness remaining highly dependent on the variable order), and in fact a number of model checkers use them for state-space representation (e.g. NuSMV, MCMAS).

In 2002, A. Darwiche and P. Marquis published [4] a comparative analysis of most existing DAG-based target compilation languages in terms of their succintness and the polytime operations they support. They show that all of these languages are subsets of a broad language called *negation normal form* (NNF).

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

Remark: OBDDs are NNF sentences. Figure 3 shows an OBDD and its corresponding NNF sentence.

### 2.3.3 Project Directions

This “knowledge compilation map” was the starting point for this project, whose initial objective was to investigate new target compilation languages suitable for application to model checking, seeking a potential improvement on BDDs (which constitute so far the “industry standard”).

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 [5], BDD-trees [6], and several other subsets of NNF described in [4], we opted for a relatively novel target compilation language called *sentential decision diagram* (SDD).

SDDs are a subset of NNF, possess the required properties, and had not yet been experimented with in the context of model checking and state-space representation. Moreover, some of the experimental results presented in [1] and [3] demonstrate that SDDs can lead to a significant improvement on BDDs in terms of computation time and memory usage.

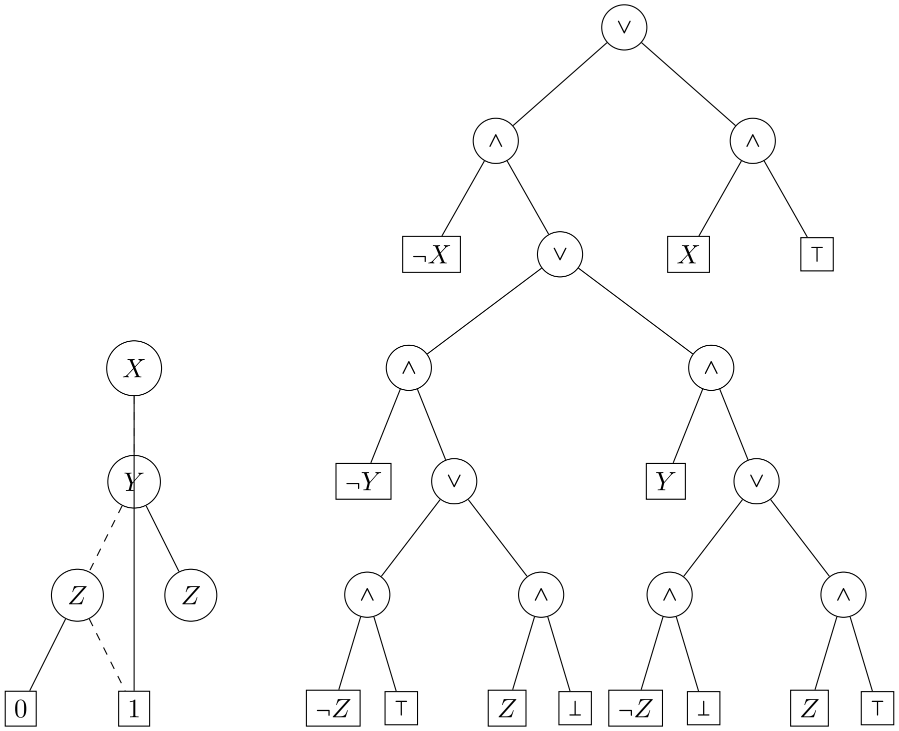

Figure 3: An OBDD (left) and its corresponding NNF sentence (right). Although the former seems much more compact, the difference in size is only linear and both representations are essentially the same. TODO sort this oout

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

## 2.4 Sentential Decision Diagrams

### 2.4.1 Preliminary Definitions

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

**Definition** Let $f$ be a Boolean function. If $X$ is a set of variables, the *conditioning* of $f$ on an instantiation $\mathbf{x}$, written $f|_{\mathbf{x}}$ is the Boolean function obtained by setting variables of $f$ in $X$ to their value in $\mathbf{x}$. 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 $\alpha$ is strongly deterministic on $X$. Then $\alpha$ 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 $\alpha$ 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 $\beta$ is compressed.

Remark: Notice 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 $p_i \wedge q_j \neq \bot$.

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 $f(\boldsymbol{X}, \boldsymbol{Y})$ has exaclty one compressed $\boldsymbol{X}$-partition.*

**Proof** proof

### Vtrees

Vtrees ("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.

**Example** Let $X = \{x_1, x_2, x_3, x_4\}$. The following are two vtrees for $X$.

Note that a vtree on $\mathbf{X}$ is stronger than a total order of the variables in $\mathbf{X}$. Example **??** shows two distinct vtrees which induce the same variable order.

Now, let $f(\mathbf{X})$ be a Boolean function and let $v$ be a vtree for $\mathbf{X}$. We can construct an SDD for $f$ using a recursive algorithm, as follows.

We first notice that variables in $\mathbf{X}$ are split into two disjoint sets, namely the variables in $v^l$ (say, $\mathbf{Y}$) and the variables in $v^r$ (say, $\mathbf{Z}$). Now, let $\alpha = \{(p_1, s_1), ..., (p_n, s_n)\}$ be a $\mathbf{Y}$-partition of $f(\mathbf{Y}, \mathbf{Z})$. We apply this decomposition again to each prime

[...] describe more

### 2.4.2 Syntax and Semantics of SDDs

**Definition** (Syntax) Let $v$ be a vtree. $\alpha$ 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 internal, and $\alpha$ 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 $v^r$.

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

**Definition** (Semantics) Let $\alpha$ 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 $\alpha$ and $\beta$ are *equivalent* (written $\alpha \equiv \beta$) if $\langle \alpha \rangle = \langle \beta \rangle$.

### 2.4.3 Construction of SDDs

### 2.4.4 Canonicity of SDDs

It is obvious that if SDDs $\alpha$ and $\beta$ are equal, then they are equivalent. We would however like to impose conditions on the construction of $\alpha$ and $\beta$ so that $\alpha \equiv \beta \Rightarrow \alpha = \beta$. 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.*

**Proof**

**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 $\alpha$.

An SDD can be trimmed by traversing it bottom-up and replacing decompositions of the form $\{(\top, \alpha)\}$ or $\{(\alpha, \top), (\neg\alpha, \bot)\}$ by $\alpha$.

**Theorem 2.5** *Let $\alpha$ and $\beta$ be compressed and trimmed SDDs. Then*

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

**Proof** by induction

### 2.4.5 SDD Operations

The `apply` algorithm to compose SDDs $\alpha$ and $\beta$ using a Boolean operator takes time $O(|\alpha||\beta|)$ provided both SDDs are normalised. See [1] for details of the algorithm.

The `exists` algorithm is the implementation of the $\exists X$ operator on function $f$ for variable $X$. Recall

$$\exists X f = f[0/X] \vee f[1/X]$$

This can be implemented using the `restrict` algorithm which returns the conditioning $f|_{\mathrm{x}}$ for an instantiation x of $X$.

# 3 Implementation

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 Preliminaries

### 3.1.1 The SDD Package

*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.

#### 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 [3] 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 [8]. 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 [3] (the one implemented in the SDD Package). We start with a vtree node $v$ (typically, the current vtree of the manager). The algorithm starts by making two recursive calls on $v_l$ and $v_r$. We then consider two subtrees of $v$:

TODO finish description of algorithm

#### 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 difference in design and implementation.

Fortunately, the libraries have a very similar design. Both are organised around a manager handling all internal operations.

- garbage collection (reference counts handled by user in SDDs, by )
- memory management and cache
- reordering - mention that BDD research is more advanced (lots of reordering options)
- a few functions don't exist in SDD package in the same way: exists and swap variables

### 3.1.2 MCMAS

MCMAS (Model Checker for Multi-Agent Systems) is a BDD-based model checker developed at Imperial College. It was specifically designed for 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).

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.

#### 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:

- v, pv, a, ...
- vRT
- initial and reachable states
- bdd cache
- formulae

The main steps in the program are

1. encode transition relation
2. encode reachable state space
3. compute SAT on modal formulas and compare with reach

encore quelques lignes encore quelques lignes encore quelques lignes encore quelques lignes encore quelques lignes encore quelques lignes encore quelques lignes encore quelques lignes encore quelques lignes encore quelques lignes encore quelques lignes

### 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* state), 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 great 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).

### 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:

- Ordering option 1:

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

- Ordering option 2:

$$\underbrace{x_1, ..., x_n}_{\text{agent } i}$$

- Ordering option 3:
- Ordering option 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, [9]) 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!), for this project we decided to exclude the 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.

## 3.2 Contributions

### 3.2.1 A Model Checker Based on SDDs

Reference to code in appendix:

- new variables and data structures (vtree, params)
- var_order
- vtree construction

### 3.2.2 Challenges

Mainly due to the following two issues, the time spent implementing the model checker was much longer than expected.

### Garbage Collection

As described in 3.1.1, 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.

### Debugging and Comparing

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 representations of 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 using a SAT checker (we used [13]). This is a good solution for small functions, but it becomes impractical very quickly when the number of variable exceeds about 10 (this 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 **??** for details), to ensure that the resulting structures are comparable. We then have two options:

- Comparing the graphical representation of each data structure, provided by both APIs via a DOT file [14]. 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.
- Comparing the size (i.e. the number of nodes) of each structure. Again, most of the time two non-equivalent representations will have very different sizes, but representations with comparable sizes are not necessarily equivalent (note that due to the way SDD and BDD nodes are represented, the size of a BDD will not be 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.

We were able to verify the correctness of our implementation by looking at the actual result of the execution. If our model checker yields the same answer as MCMAS on a large number of Boolean formulae, it is very likely that the encoding of the transition relation, the reachable state space, and the implementation of `SAT` are correct.

# 4 Evaluation

## 4.1 Setup

### 4.1.1 Example Models

- Description of models used for quantitative analysis

### 4.1.2 Machine and Configurations

### 4.1.3 Outline

## 4.2 Comparison with Right-Linear Vtrees

## 4.3 Comparison with Standard Vtrees

## 4.4 Towards a Better Vtree: Observations

- The environment generally contains the highest number of variables (and hence the largest evolution SDD).

- Applying the transition relation in the state space generation represents a significant amount (think more than 80%) of the overhead. In some cases `sdd_exists()` is also very slow, but only in some cases, which can lead to thinking that significant improvements are possible.

## 4.5 New Vtrees and Experiments

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!

## 4.6 Comparisons with Dynamic Reordering and Minimisation

## 4.7 More suggestions for speed improvement

- Call `sdd_apply()` on a reduced vtree
- Existentially quantify out variables in a smart order.

  The CUDD algorithm for this is recursive - need to sort sth out

# 5 Conclusions and further work

## 5.1 Review

## 5.2 Future work

I will see what I don't have time to do before the deadline. Potential 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.

# References

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

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

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

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

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