Contents

Abstract

1 Introduction

- Importance of autonomous systems and their verification
- The state explosion problem, summary of different approaches
- Our idea and contributions
- Project objectives

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

$$\begin{aligned}
(M, s_0) \models p &\quad\Leftrightarrow\quad s_0 \in \pi(p) \\
(M, s_0) \models \neg\varphi &\quad\Leftrightarrow\quad (M, s_0) \not\models \neg\varphi \\
(M, s_0) \models \varphi \wedge \psi &\quad\Leftrightarrow\quad (M, s_0) \models \varphi \text{ and } (M, s_0) \models \psi \\
(M, s_0) \models X\varphi &\quad\Leftrightarrow\quad (M, s_1) \models \varphi \text{ for all states } s_1 \text{ such that } R(s_0, s_1) \\
(M, s_0) \models G\varphi &\quad\Leftrightarrow\quad \text{for all paths } \rho = s_0, s_1, s_2, ..., \text{ we have } (M, s_i) \models \varphi \quad \forall i \geq 0 \\
(M, s_0) \models \varphi U \psi &\quad\Leftrightarrow\quad \text{for all paths } \rho = s_0, s_1, s_2, ..., \exists j \geq 0 \text{ such that } (M, s_j) \models \psi \\
&\qquad\qquad \text{and } (M, s_k) \models \varphi \quad \forall 0 \leq k < j
\end{aligned}$$

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:

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

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, φ is true at the next state, φ holds forever in the future, and φ holds until ψ becomes true.

The dual operator $AX\varphi \coloneqq \neg EX\neg\varphi$ 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:

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

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 $Fp \to Fq$ 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:

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

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:

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

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 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.

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 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 $(IS, s_0) \models \varphi$, for all initial states $s_0 \in I$.

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

```
function SAT($\varphi$)
// returns $[\varphi]$
if $\varphi = \top$: return $G$
if $\varphi = \bot$: return $\emptyset$
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 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 $g_0 \rightarrow g_1$ 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 := {$g_0 \in G \mid g_0 \rightarrow g_1$ 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 $g_0 \rightarrow g_1$, $g_1 \in$ 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 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*.

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

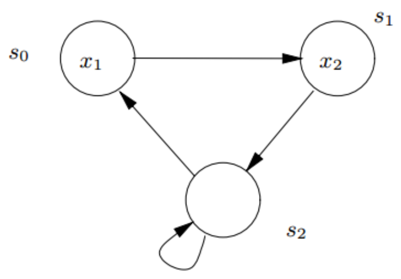

Figure 1: A model

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 [?] 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
$\varnothing$	$\perp$
$\{s_0\}$	$x_1 \wedge \neg x_2$
$\{s_1\}$	$\neg x_1 \wedge x_2$
$\{s_2\}$	$\neg x_1 \wedge \neg x_2$
$\{s_0, s_1\}$	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$
$\{s_1, s_2\}$	$(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$
$\{s_0, s_2\}$	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge \neg x_2)$
$\{s_0, s_1, s_2\}$	$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$

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 the 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. In the example above, the transition relation $\to$ of the model is $\{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_2)\}$. Using our Boolean representations for s_0, s_1, and s_2, we compute the representation of $\to$ to be

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

[needs more details on the computation]

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.

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

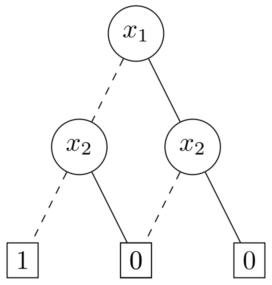

Figure 2: BDT for $f(x_1, x_2) = x_1 \wedge \neg x_2$ under the ordering $[x_1, x_2]$

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 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 **??** 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 a subset 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 trees-of-BDDs **??**, BDD-trees **??**, and several other subsets of NNF described in **??**, 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 **??** and **??** 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

2.4.1 Preliminary Definitions

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

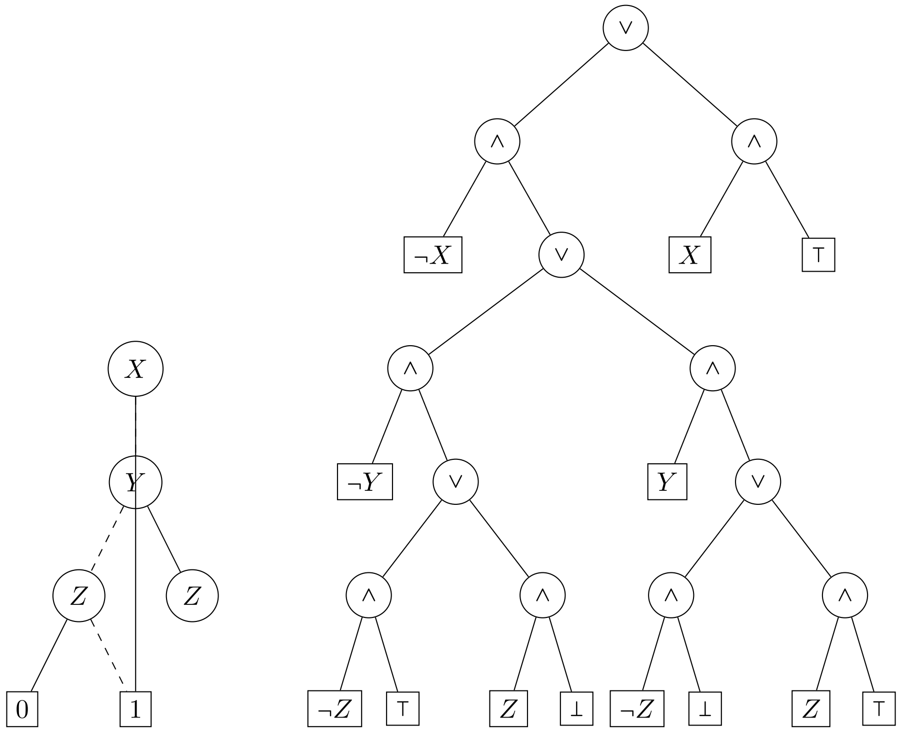

Figure 3: An OBDD (left) and its corresponding NNF sentence (right). Although the former seems much more compact, the difference in size is only linear and both representations are essentially the same. TODO sort this oout

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 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: 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. 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 internal, 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.

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 Construction of SDDs

2.4.4 Canonicity of SDDs

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 $\alpha \equiv \beta \Rightarrow \alpha = \beta$. 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 α.

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 α.

Theorem 2.5 *Let α and β 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 α and β using a Boolean operator takes time $O(|\alpha||\beta|)$ provided both SDDs are normalised. See [?] 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 Technical Preliminaries

3.1 MCMAS Specifics

- *Some* implementation details (enough to understand the steps needed for replacing BDDs with SDDs).
- Variable allocation and how we keep track of it
- Description of the 4 variable orderings available
- ADDs to represent algebraic expressions

3.2 The SDD Package

- Development, summary of features
- Dynamic minimization and automatic garbage collection
- Algorithm for vtree search (and operations for navigating the space of vtrees)

4 Contributions

4.1 Implementation of a model checker based on SDDs

- Functionality and limitations

4.2 Some heuristics

(This is one of the most important sections - but heavily dependent on the next couple of weeks)

5 Evaluation

5.1 Models

- Description of models used for quantitative analysis

5.2 Comparison without dynamic variable reordering

5.2.1 Comparison of various variable orderings with corresponding right-linear vtrees

- Built-in orderings

  Results in Table 1. Only need to do time comparisons since structures have the same size

- Orderings resulting from dynamic vtree search (TODO)

  Results should be in Table 2.

5.2.2 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.

5.2.3 Comparison of various variable orderings with equivalent non-right-linear *dissections* of these orderings

- Built-in orderings with standard vtrees:

  Left-linear and vertical vtrees proved to be very bad, so we will focus on balanced vtrees.

- 'Clever' vtrees

  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!

5.2.4 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

5.2.5 Analysis and conclusion

5.3 Comparison using dynamic minimization algorithms

- Experiments with different initial orderings and vtrees
- Analysis of heuristics proposed in previous section
- Comparison of different vtree minimization algorithms/settings (The SDD package allows the user to implement their own minimization function, or change the settings in the original functions. It would be interesting to find out what happens if we change the minimization thresholds for time and memory, or if we choose a different algorithm for searching the space of vtrees)

6 Conclusions and further work

6.1 Review

6.2 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.

7 Bibliography