Zero-suppression Decision Diagrams versus Binary Decision Diagrams on Dynamic Epistemic Logic Model Checking

Daniel Miedema
s2478579
November 2020

Master Thesis
Artificial Intelligence
Rijksuniversiteit Groningen

Supervised by:
Malvin Gattinger (ILLC, Universiteit van Amsterdam)
Rineke Verbrugge (Artificial Intelligence, Rijksuniversiteit Groningen)

Special thanks

I would like to express my sincere gratitude to:

- Hanneke Steen, who made me believe in myself and in my interest towards abstract information modelling.
- My two supervisors for their kind patience and openness throughout the long project, and of which specifically Malvin Gattinger - for his guidance in formal writing and in learning Haskell.
- My girlfriend, Lieve van Voorthuijsen, for supporting me throughout my burnout.

..without whom I could not have created this thesis.

Contents

Chapter 1

Introduction

1.1 Abstract

In this thesis we first give a theoretical analysis of translating Dynamic Epistemic Logic (DEL) to physical memory for model checking tasks, afterwards we give an analysis of Binary Decision Diagrams (BDDs) and the four Zero-suppressed Decision Diagram variants (ZDDs) and how they provide advantages for the memory and computation speed for model checking tasks.

The analysis sets forth an alternative explanation for the relation between BDDs and ZDD variants by comparing their node elimination rules with more general inference rules (when considering boolean function contexts over a possibly countable infinite variables). This is then used to explain why and when ZDDs and BDDs need to keep track of a context in their traversal algorithms and leads to proposing a new naming convention for the ZDD variants. Furthermore we identify a common graph-pattern indicative of ZDD efficiency (named XOR-domain), which can be considered novel in the context of symbolic model checking.

We conclude with an experimental comparison between BDD and ZDD compactness on multiple classical DEL problems. For this we extend the SMCDEL, a program and theoretical framework for symbolic DEL model checking using BDDs, with ZDD functionality.

Our results show that the right ZDD elimination rule, instead of the BDD elimination rule, can significantly reduce the decision diagrams used for model checking classical DEL puzzles (ranging from 6% to 54% fewer nodes). We also show that for our chosen DEL models, the most compact ZDD variant can be predicted by recognising strong or weak sparsity and the XOR-domain in the symbolic problem statement.

1.2 Model checking

Model checking [CE81; CES86; QS82] has been used as a verification technique since the 1980's. It refers to a set of algorithms that can verify properties of state transition systems by searching through the corresponding state transition graphs. However, these methods suffer from the state explosion problem [Cla+01b], wherein the number of system states grows exponentially with the number of system components. This realistically limits the systems to have small numbers of states, making them unfit for industrial application. Therefore more efficient methods of Model checking are needed. Efficiency in model checking refers to speed and memory usage of the program, which are the limiting factors to its applicability.

Techniques that used symbolic state space exploration were eventually developed [Bur+92; CMB90; Pix90] which improve efficiency. These reduced the amount of resources needed, as they removed redundant information from the model. A common approach is through use of Binary Decision Diagrams (BDDs) [Ake78], a class of structures that keeps track of sets of states, which allows for transition between sets of states instead of individual ones (see Section 2). Other methods have also found success, like ZDDs [Min93] and Bounded Model Checking [Cla+01a], each being particularly efficient for their own domain of problems (see Section 2).

Most developed model checkers using symbolic state exploration express properties of their models in Linear Temporal Logic (LTL) [Kam68] and Computational Tree Logic (CTL) [CES86]. LTL gives a language for reporting about the ordering of events in time, by introducing time explicitly in the model. Yet problems come in many forms, for which different logics can be more useful or convenient.

For problems typically described using epistemic operators (e.g. in multi-agent systems), Dynamic Epistemic Logic (DEL) is the standard logic. In DEL models, passage of time is usually represented as model transforming actions [VHR13].

Little work has explored optimisations for DEL model checking, especially using alternatives to BDDs. My thesis therefore analyses and compares the efficiency of ZDDs with BDDs within DEL model checking. This will be done by continuing the development of SMCDEL [Gat18b], a DEL model checker using BDDs.

1.3 Related research

Symbolic model checking was first implemented in SMV from [McM93] with BDDs, and its current improved version, NuSMV 2 [Cim+02], includes methods for Bounded Model Checking using SAT solvers. NuSMV uses temporal logics as input languages and does not support epistemic operators.

MCK [GV04] is one of the first symbolic model checkers supporting knowledge operators. It uses BDDs to represent temporal Kripke models. Recent versions also offer bounded semantics via SAT solving. It is still based on temporal logic but has knowledge operators defined for different kinds of agents (observational, clock and synchronous perfect recall) that can be used.

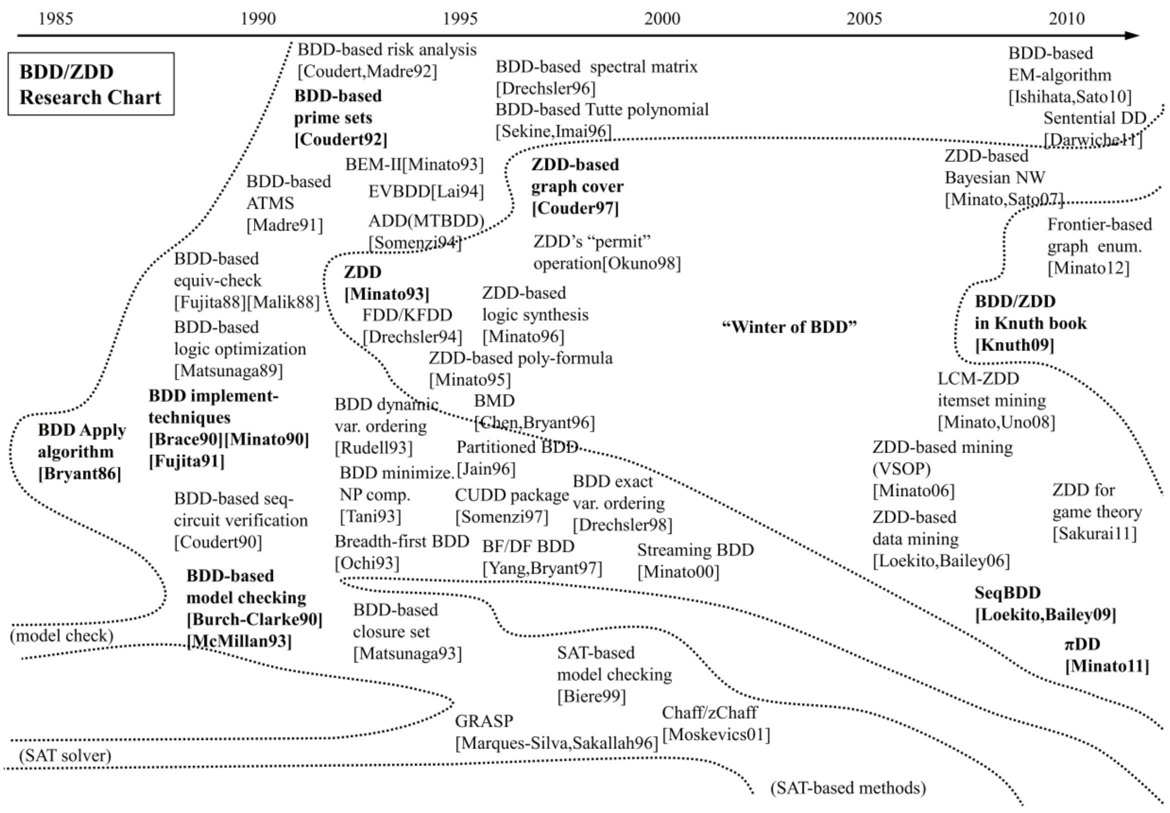
Other similar model checkers for epistemic temporal logics are MCTK (2004) [SLZ04] and MCMAS (first released in 2006) [LR06]. A comparison between MCMAS, MCK and MCTK is given in [LQR17].

For model checkers supporting Dynamic Epistemic Logic, the standard is set by the two explicit model checkers by Jan van Eijck: DEMO [Eij07] and its successor DEMO-S5 [Eij14], which is optimised for S5 logics using partitions instead of list of pairs to represent relations. Explicit implementation model checkers like DEMO are relatively easy to use as the models closely resemble the standard structure of Kripke models.

In addition, there is the symbolic model checker I will be extending with ZDD functionality in my project; SMCDEL, by Malvin Gattinger. It provides symbolic model checking by use of BDDs, and can translate between explicit and symbolic models. This program's methods are explained more in-depth in Subsection 2.1.

After Byrant's paper [Bry86] introducing BDDs in 1968, BDDs have become a hot topic and were rapidly developed. ZDDs were introduced by Minato in 1993 [Min93] as a adaption of BDDs, and showed how ZDDs are especially suitable for combinatorial problems. Patterns in combinatorial problems are discussed in [OMI98] and how ZDDs provide utility for these patterns is treated in [Cou97]. Many comparisons between the succinctness of BDDs and ZDDs exists but an excellent analysis of the most notable factors are given in "The art of computer programming, volume 4A: combinatorial algorithms" by Knuth [Knu11]. An overview of the progress on the field of BDD and ZDD research is given by Minato in his survey paper "Techniques of BDD/ZDD: brief history and recent activity" [MIN13].

An implementation for symbolic model checking using ZDDs is relatively unresearched, partly due to difficulties with domain dependency (as defined by Minato in [Min01]), which will be a central component to our ZDD analysis. There are no publications about testing ZDDs for DEL based symbolic model checking to our knowledge.


1985
1990
1995
2000
2005
2010
BDD/ZDD Research Chart
BDD-based risk analysis [Coudert,Madre92]
BDD-based prime sets [Coudert92]
BDD-based spectral matrix [Drechsler96]
BDD-based Tutte polynomial [Sekine,Imai96]
BDD-based EM-algorithm [Ishihata,Sato10]
Sentential DD [Darwiche11]
BEM-II[Minato93]
EVBDD[Lai94]
ADD(MTBDD) [Somenzi94]
BDD-based ATMS [Madre91]
ZDD-based graph cover [Couder97]
ZDD's "permit" operation[Okuno98]
ZDD-based Bayesian NW [Minato,Sato07]
Frontier-based graph enum. [Minato12]
BDD-based equiv-check [Fujita88][Malik88]
BDD-based logic optimization [Matsunaga89]
ZDD [Minato93]
FDD/KFDD [Drechsler94]
ZDD-based logic synthesis [Minato96]
ZDD-based poly-formula [Minato95]
"Winter of BDD"
BDD/ZDD in Knuth book [Knuth09]
BDD implement-techniques [Brace90][Minato90] [Fujita91]
BDD Apply algorithm [Bryant86]
BDD dynamic var. ordering [Rudell93]
BMD [Chen,Bryant96]
Partitioned BDD [Jain96]
LCM-ZDD itemset mining [Minato,Uno08]
ZDD-based mining (VSOP) [Minato06]
BDD minimize. NP comp. [Tani93]
CUDD package [Somenzi97]
BDD exact var. ordering [Drechsler98]
ZDD for game theory [Sakurai11]
BDD-based seq-circuit verification [Coudert90]
ZDD-based data mining [Loekito,Bailey06]
Breadth-first BDD [Ochi93]
BF/DF BDD [Yang,Bryant97]
Streaming BDD [Minato00]
BDD-based model checking [Burch-Clarke90] [McMillan93]
SeqBDD [Loekito,Bailey09]
πDD [Minato11]
(model check)
BDD-based closure set [Matsunaga93]
SAT-based model checking [Biere99]
GRASP [Marques-Silva,Sakallah96]
Chaff/zChaff [Moskevics01]
(SAT solver)
(SAT-based methods)

A historical overview of some of the important publications on BDDs and ZDDs, made by Minato [MIN13], used with permission.

Chapter 2

Theory

SMCDEL is a model checker for DEL. This means we give it a model and a formula on which it can perform model checking methods. This is used for example to proof properties of the model, or solve logic puzzles. We restrict ourselves to Dynamic Epistemic Logic (DEL). The theory in the chapter roughly builds up as follows:

- starting with a review on how semantics of DEL formulas are defined by Kripke models is given,
- followed by an explanation on how symbolic model checking is achieved in SMCDEL by use of belief structures and transformers,
- then how the boolean functions contained in the belief structures can be reduced in size by inference and merging is given,
- leading to an in-depth analysis of Simple Decision Diagrams (SDDs), Binary Decision Diagrams (BDDs) and Zero-suppressed Decision Diagrams (ZDDs),
- and finally, theoretical indicators for how ZDDs can be more compact for classical DEL problems are discussed.

The first subsection proceeds quickly and only minimal explanation is given to catch up with SMCDEL's theory. Next to most definitions in this subsection a reference is given to the definitions of Gattinger's PhD thesis 'New Directions for Model Checking Dynemic Epistemic Logic' [Gat18a] which they are adopted from, such that the reader can find more extensive examples and explanations. The sections analysing the different decision diagrams takes a slower pace in explaining as there we introduce the more novel theoretical contributions.

2.1 Symbolic model checking with DEL

2.1.1 DEL interpreted on Kripke models

The standard semantics of DEL is based on Kripke models. First the syntax for DEL is given, a boolean language in Backus Naur Form [Knu64] with extended with non-boolean operators for epistemic and dynamic relations. For this formal language we assume a countably infinite supply of fresh atomic propositional variables, where a subset of them is called a vocabulary.

Definition 2.1. *Given a vocabulary V, a set of epistemic agents I, and a set of actions (atomic events) A, the language of* ***Dynamic Epistemic Logic*** *is given by*

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid B_i\varphi \mid C_\Delta\varphi \mid [A, a]\varphi$$

where $p \in V$, $i \in I$, $\Delta \subseteq I$.
B_i and C_Δ are knowledge operators as in Definition 2.4, and $[A, a]$ is an action operator as in Definition 2.6. The common abbreviations $\bot := \neg\top$, $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$ and $\varphi \rightarrow \psi := \neg(\varphi \wedge \neg\psi)$ apply as usual.

With this language syntax, we can now define a semantics where syntactical formulas convey information about propositional variables and relations within a model. First we consider only the semantics of the language without epistemic and dynamic operators:

Definition 2.2 ([Gat18a]: 1.0.2.)**.** *The language containing the first four operators of Definition 2.1, is called **$\mathcal{L}_B(V)$**. It is a boolean language where a given **boolean assignment** determines what formulas hold. A boolean assignment over a vocabulary V assigns to each atomic proposition a truth value, usually denoted by 1 for True, and 0 for False. It is thus a function of type $V \rightarrow \{1, 0\}$. When the vocabulary V is fixed, we can identify assignments with its **1-set** (the subset of atomic propositions that it makes true), i.e. $s = \{p \in V \mid s(p) = 1\} \subseteq V$. We write $\models$ for the usual boolean semantics:*
1. $s \models \top$ always holds
2. $s \models p$ iff $p \in s$
3. $s \models \neg\varphi$ iff not $s \models \varphi$
4. $s \models \varphi \wedge \psi$ iff $s \models \varphi$ and $s \models \psi$
*where φ is an arbitrary formula in $\mathcal{L}_B(V)$. A formula φ is **valid** iff it satisfies all assignments and then we write $\models \varphi$. We call two formulas φ and ψ semantically equivalent and write $\varphi \equiv \psi$ iff they satisfy exactly the same assignments.*

We can extend this semantics to include epistemic operators when we interpret them on Kripke models instead of boolean assignments over V. Kripke models contain the previous semantics, by representing boolean assignments over the vocabulary as worlds. Kripke models then also allow for adding labelled relations between those worlds in order to describe epistemic information of an agent (each corresponding to a label). First a definition of Kripke models is given:

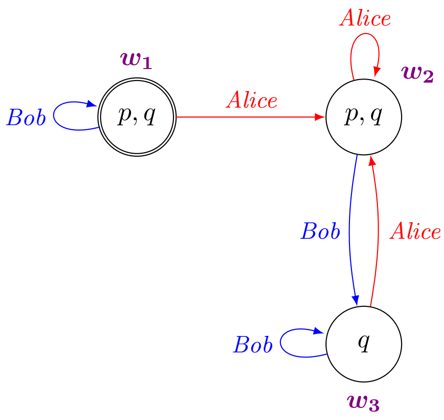
Definition 2.3 ([Gat18a]: 1.1.2.)**.** *A **frame** for a set of agents $I = \{1, ..., n\}$ is a tuple $\mathcal{M} = (W, R)$, where W is a finite non-empty set of possible **worlds** and R is a family of binary relations over W indexed by agents: $R_i \subseteq W \times W$ for each $i \in I$. A **Kripke model** for a set of agents I and vocabulary V is a tuple $\mathcal{M} = (W, \pi, R)$, where (W, R) is a frame for I and $\pi : W \rightarrow \mathcal{P}(V)$ is a **valuation function**: it describes a boolean assignment, $\pi(w) \subseteq V$, for each $w \in W$.*
*By convention, we use $W^{\mathcal{M}}, R_i^{\mathcal{M}}$ and $\pi^{\mathcal{M}}$ to refer to the components of $\mathcal{M}$ but we omit the superscript $\mathcal{M}$ if it is clear from the context which model we are concerned with. For any group of agents $\Delta \subseteq I$ we denote the transitive closure of the union of their relations by $R_\Delta := (\bigcup_{i \in \Delta} R_i)^*$, which we will use to interpret common knowledge. A model $\mathcal{M}$ is finite iff $W^{\mathcal{M}}$ is finite. A model is an S5 Kripke model iff, for every i, the relation R_i is an equivalence relation. A **pointed Kripke model** is a pair $(\mathcal{M}, w)$ where w is a world of $\mathcal{M}$.*

Each agent i has a labelled relation R_i representing which worlds the agent considers possible. Using this we can interpret the knowledge operator for such an agent: i knows "something" iff it is the case at all the worlds that i considers possible. "Something" here does not only refer to the propositions that hold at the worlds but also the relations, e.g. if at all worlds considered possible by an agent, Alice, there are relations labelled by another agent, Bob, to the same worlds, Alice knows that Bob considers the same worlds to be possible as her. In such a case, Bob also knows what Alice does, and so he also knows that Alice knows what he considers possible. This scenario is described by the common knowledge operator, which is interpreted by the transitive and reflexive relation R_Δ for a group of agents Δ. Were it not included in the language, an infinite set of infinite formulas would be needed to describe the same information; $B_{i_1} ... B_{i_\infty} \varphi$, where i_x can represent any agent in Δ whether it has been used before or not. Common knowledge thus also implies all its sub-formulas of knowledge operators up to a finite number n, e.g. if $C_\Delta\varphi$ holds then $B_i\varphi$ (with $n = 1$) and $B_i B_j\, \varphi$ (with $n = 2$), where $i, j \in \Delta$, also hold.

Example 2.1. *Alice and Bob have a conversation about Alice's master thesis. Alice believes she will* *pass with the current version but her friend Bob tells her he beliefs she will fail. Bob in fact has been lying to Alice, in order for her to try to improve the thesis even more. They both confirm to each other that the thesis is at least interesting.*

When we want to model this situation as a Kripke model we have:
p = passing with current version
q = thesis is interesting

$(B_{Alice}\, p \wedge B_{Alice} B_{Bob}\, \neg p) \;\wedge\; (B_{Bob}\, p \wedge B_{Bob} B_{Alice}\, p \wedge B_{Bob} B_{Alice} B_{Bob}\, \neg p) \;\wedge\; C_{Alice,Bob}\, q$



In the story of what world actually is the case is the one where Bob lies about his beliefs to Alice. This means it is a pointed Kripke model and we indicate the given world with a double lined circle.

The vocabulary of a Kripke model (implicitly) defines the co-domain of the valuation function π. This is important to remember, because soon we shall interpret the DEL language on structures containing valuation functions over different vocabularies. We can now define the semantics of DEL over Kripke models, such that it includes the knowledge operators:

Definition 2.4 ([Gat18a]: 1.1.1.)**.** *Semantics for an epistemic language, $\mathcal{L}(V)$, on pointed Kripke models are given inductively as follows:*
1. $(\mathcal{M}, w) \models \top$ always holds.
2. $(\mathcal{M}, w) \models p$ iff $p \in \pi^{\mathcal{M}}(w)$.
3. $(\mathcal{M}, w) \models \neg\varphi$ iff not $(\mathcal{M}, w) \models \varphi$.
4. $(\mathcal{M}, w) \models \varphi \wedge \psi$ iff $(\mathcal{M}, w) \models \varphi$ and $(\mathcal{M}, w) \models \psi$.
5. $(\mathcal{M}, w) \models B_i\varphi$ iff for all $w' \in W$, if $R_i ww'$, then $(\mathcal{M}, w') \models \varphi$.
6. $(\mathcal{M}, w) \models C_\Delta\varphi$ iff for all $w' \in W$, if $R_\Delta ww'$, then $(\mathcal{M}, w') \models \varphi$.

If we consider all Kripke models, the set of valid formulas obtained from these semantics is the logic usually called K. This general version reflects the idea of belief, since it allows for false beliefs, which is why we choose to use B as label for the general operator. Restricting the class of frames to specific kinds of relations corresponds to different epistemic properties. For example when restricting the frame to the equivalence relation we get the knowledge operator of S5: Whatever is known also has to be true, any agent who knows something also knows that she knows it and if an agent does not know something, she knows that she does not know it. We use K instead of B as operator to indicate equivalence relations.

The given semantics until now do not yet describe actions that change the epistemic and factual information in a given model. Once the agents and possible worlds are given for a Kripke model, formulas of $\mathcal{L}(V)$ can only characterise subsets of relations and worlds *within* the model. However, when describing possible change by action events, we have to represent (action) relations between the original model (before the event) and the resulting changed Kripke model (according to the actions taken). This can be characterised as an action model acting on (thus changing) the Kripke model.

Since we also want to be able to describe change with SMCDEL's DEL formulas, we extend the semantics by introducing action models:

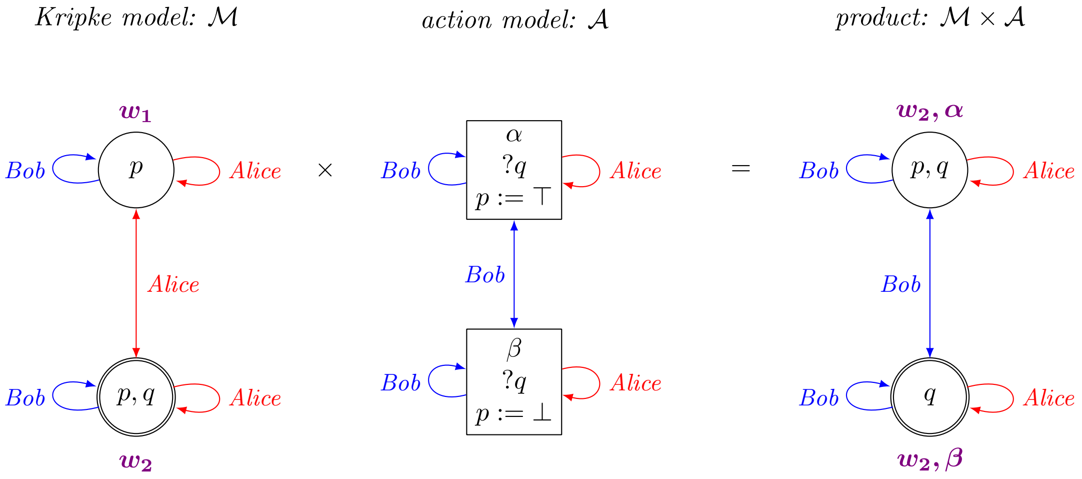
Definition 2.5 ([Gat18a]: 1.3.1.). *Suppose we have some vocabulary V. An* ***Action Model*** *is a tuple $\mathcal{A} = (A, R^A, pre, post)$ where:*
A is a set of ***atomic events****, R^A is a family of relations $R_i \subseteq A \times A$ for each i, $pre : A \to \mathcal{L}(V)$ is a function which assigns to each event a formula called the* ***precondition*** *and $post : A \times V \to \mathcal{L}_B(V)$ is a function which at each event assigns to each atomic proposition a boolean formula called the* ***postcondition****.*
We call $\mathcal{A}$ an S5 action model iff all the relations are equivalence relations. Given a Kripke model $\mathcal{M}$ and an action model $\mathcal{A}$ using the same vocabulary, we define their ***product*** *by $\mathcal{M} \times \mathcal{A} := (W^{new}, R_i^{new}, \pi^{new})$ where:*

- $W^{new} := \{ (w, a) \in W \times A \mid \mathcal{M}, w \models pre(a) \}$
- $R_i^{new} := \{ ((w, a), (v, b)) \mid R_i^{\mathcal{M}} wv \text{ and } R_i^A ab \}$
- $\pi^{new}((w, a)) := \{ p \in V \mid \mathcal{M}, w \models post_a(p) \}$

An ***action*** *is a pair $(\mathcal{A}, a)$ where $a \in A$. To update a pointed Kripke model with an action we define $(M, w) \times (\mathcal{A}, a) := (M \times \mathcal{A}, (w, a))$.*

An action causing factual change in the model corresponds to replacing the valuation function π for only those worlds where the old boolean assignment satisfies the postcondition. Reducing the total number of worlds is done by the precondition, e.g. when an action is impossible to execute from a possible world, and the action is in fact executed, we now know that that world was not possible and should be removed from the total of possible worlds. In contrast, adding more worlds can be achieved by having multiple events in an action model, such that we get multiple copies of $\mathcal{W}$ for each event. These can then be trimmed and adjusted again by the pre- and post-conditions. The epistemic relations of an agent, in turn, can be removed for an event when it has no reflexive relation in R^A, and can be added by having a relation between two events in R^A.

Example 2.2. *Consider a (S5) Kripke model where p is true and is common knowledge, q is known only to be true to Bob and Alice knows Bob knows whether q. An action with precondition q happens where p possibly changes, Alice witnesses the result of not p and Bob only witnesses the action happen.*



The action model is drawn here such that each square corresponds to an atomic event in A in which the name of the atomic event is given in the top line, the second line shows the precondition, the bottom line gives postcondition and the edges between all squares represents R^A.

With these possibilities we can model transitions between K-models (given a vocabulary, a set of agents and an action model), thus we can update our semantics to finally reflect the full DEL syntax from Definition 2.1:

Definition 2.6 ([Gat18a]: 1.3.7.). *The semantics for dynamic operators with use of action models*

extends $\mathcal{L}(V)$ to $\mathcal{L}_D(V)$ by adding the following clause:
7. $M, w \models [\mathcal{A}, a]\varphi$ iff $M, w \models pre(a)$ implies $M \times \mathcal{A}, (w, a) \models \varphi$

Note that Definition 2.6 does not allow for any dynamic operators in the preconditions, and the post-conditions are restricted even further, to only boolean formulas. This is to avoid infinitely recursive definitions of actions, and thus ensures that the language and its semantics are well-founded. Restricting pre- and post-conditions in this manner does not restrict the class of updates we can describe [VDK07].

Now that we have given the syntax and semantics for DEL, we can state the task of model checking for which we want SMCDEL to be efficient:

Definition 2.7. *For a given pointed Kripkemodel $(\mathcal{M}, w)$ and a **query**, a formula $\varphi \in \mathcal{L}_D$, the query is true iff $(\mathcal{M}, w) \models \varphi$ and the query is valid iff forall $w' \in W$, $(\mathcal{M}, w') \models \varphi$.*

2.1.2 Belief Structures

Gattinger has shown that symbolic model checking can be achieved for (DEL) Kripke models. For this, Gattinger proposed the use of belief structures and belief transformers. Belief structures are an alternative approach for representing a Kripke model, rewriting it in terms of a vocabulary and boolean functions over the vocabulary. The use of boolean formulas then allows for representing sets of states as laws over all possible states, resulting in symbolic model checking. How boolean functions can achieve this is shown later, in Subsection 2.2.3.

The goal here is thus to define belief structures such that: For any belief structure $\mathcal{F}$, any state s of $\mathcal{F}$ and any formula φ, we have $(\mathcal{F}, s) \models \varphi$ iff the translation of the belief structure to Kripke model (later defined in Subsection 2.17) also models φ; $(\mathcal{M}(\mathcal{F}), s) \models \varphi$

First the definition of knowledge structures, a S5 logic alternative, is given. Second, the symbolic encoding of Kripke worlds as boolean formulas is defined. Third, the generalisation to belief structures is made, and lastly, the encoding of belief relations as a boolean formula is given.

Definition 2.8 ([Gat18a]: 2.2.1.)**.** *Suppose we have a set I of n agents. A **Knowledge Structure** is a tuple $\mathcal{F} = (V, \theta, O)$ where V is a finite set of propositional variables, $\theta \in \mathcal{L}_B(V)$ is a boolean formula over V, called the **state law**, and O is a family of subsets of V indexed by agents, such that $O_i \subseteq V$ for each agent i. Any $s \subseteq V$ such that $s \models \theta$ is called a **state** of $\mathcal{F}$, corresponding to a boolean assignment evaluating to 1 for θ. Any $s \subseteq V$ such that $s \not\models \theta$ corresponds to a boolean assignment evaluating to 0 for θ. Given a state s of $\mathcal{F}$, we call $(\mathcal{F}, s)$ a **scene** and define the local state of an agent i at s as $s \cap O_i$.*

A scene corresponds to a pointed Kripke model.

Definition 2.9 ([Gat18a]: 1.8.1. and 1.8.2.)**.** *Suppose we have a finite vocabulary V, a set of possible worlds W and a valuation function $\pi : W \to \mathcal{P}(V)$ which is injective, i.e. all the valuations are different. A boolean formula $\theta \in \mathcal{L}_B(V)$ is a **symbolic encoding of W** iff for all $s \subseteq V$ we have: $s \models \theta \iff \exists w \in W : s = \pi(w)$.*

Whenever π is injective, a symbolic encoding can be computed as follows: Given V, W and π, the formula $\theta := \bigwedge_{w \in W} (\pi(w) \sqsubseteq V)$ and all formulas equivalent to θ are symbolic encodings of W, where $\sqsubseteq$ abbreviates a formula which says that out of the propositions in the second argument exactly those in the first argument are true: $A \sqsubseteq B := \bigwedge A \wedge \bigwedge\{\neg p \mid p \in B \setminus A\}$

In the case of S5 models (where equivalence relations are used for knowledge), we can define agent knowledge as a set of observable variables. A boolean formula can be constructed to represent the set of states which the agent considers possible. This is done for a given scene ($\mathcal{F} = (V, \theta, O), s$) by removing all states which do not share the evaluation of s on the observable variables of the agent, thus $R_i st$ iff $s \cap O_i = t \cap O_i$ for any $\mathcal{F} \models t$.

For brevity's sake we do not consider the symbolic encoding of observable variables any further, but generalise to belief structures for the rest of the thesis, which correspond to general K models:

Definition 2.10 ([Gat18a]: 2.6.1.). *A **Belief Structure** is a tuple $\mathcal{F} = (V, \theta, \Omega)$ where V is a finite set of propositional variables called the vocabulary, $\theta \in \mathcal{L}_B(V)$ is a boolean formula over V called the state law and Ω is a set of formulas indexed by agents such that for each agent i, $\Omega_i \in \mathcal{L}_B(V \cup V')$ is a boolean formula over the double vocabulary. We call this formula the **observation law** of i. Any $s \subseteq V$ such that $s \models \theta$ is called a state of $\mathcal{F}$.*

For representing belief, each agent has its own boolean formula representing the set of states the agent believes to be possible. This set of states is thus not restricted to a subset of the set of states of θ. To allow for directed edges, the boolean formulas are defined over a double vocabulary, where we can differentiate between the receiving state and originating state of a directed edge by identifying one of them with the duplicated vocabulary:

Definition 2.11 ([Gat18a]: 1.8.8.). *If s is an assignment for V, then s' is the corresponding assignment for V'. For example, $\{p_1, p_3\}' = \{p1', p3'\}$. If φ is a formula, $(\varphi)'$ is the result of priming all propositions. For example, $(p_1 \to \neg p_2)' = (p_1' \to \neg p_2')$. If s and t' are assignments for V and V' respectively such that $V \cap V' = \varnothing$ and φ is a formula over $V \cup V'$, we also write $st' \models \varphi$ instead of $s \cup t' \models \varphi$. Suppose we have a relation R on $\mathcal{P}(V)$. A boolean formula $\Omega \in \mathcal{L}_B(V \cup V')$ is a **symbolic encoding of R** iff we have for all $s, t \subseteq V$ that Rst iff $st' \models \Omega$.*

Now we can state the semantics of $\mathcal{L}(V)$ on belief structures:

Definition 2.12 ([Gat18a]: 2.2.3 and 2.6.2.). *Semantics for $\mathcal{L}(V)$ on scenes are defined inductively as follows.*
1. $(\mathcal{F}, s) \models \top$ always holds.
2. $(\mathcal{F}, s) \models p$ iff $s \models p$.
3. $(\mathcal{F}, s) \models \neg\varphi$ iff not $(\mathcal{F}, s) \models \varphi$
4. $(\mathcal{F}, s) \models \varphi \wedge \psi$ iff $(\mathcal{F}, s) \models \varphi$ and $(\mathcal{F}, s) \models \psi$
5. $(\mathcal{F}, s) \models B_i\varphi$ iff for all states t of $\mathcal{F}$: $s \cup t' \models \Omega_i$ implies $(\mathcal{F}, t) \models \varphi$
6. $(\mathcal{F}, s) \models C_\Delta\varphi$ iff for all states t of $\mathcal{F}$: $(s, t) \in \mathcal{E}^*_\Delta$ implies $(\mathcal{F}, t) \models \varphi$. where $\mathcal{E}_\Delta$ is the relation defined by $\mathcal{E}_{st}$: $\leftrightarrow \exists i \in \Delta : s \cup t' \models \Omega_i$ and its transitive closure is denoted by $\mathcal{E}^*_\Delta$.

*We write $(\mathcal{F}, s) \equiv_V (\mathcal{F}', s')$ iff these two scenes agree on all formulas. If we have $(\mathcal{F}, s) \models \varphi$ for all states s of $\mathcal{F}$, then we say that φ is **valid** on $\mathcal{F}$ and write $\mathcal{F} \models \varphi$.*

2.1.3 Local boolean translation of belief structures

In the previous section we have shown the correspondence of the valuation function of Kripke worlds with boolean assignments (giving us a boolean formula for all worlds that any agent could belief to be possible, and a single state reflecting what is actually the case) and the epistemic relations between worlds for a given agent with their symbolic encoding (giving a boolean formula for each agent's belief relations between worlds). Yet a belief structure represents a set of formulas in $\mathcal{L}(V)$ which can contain all these types of information interchangeably. In order to perform symbolic model checking, we thus need to completely translate $\mathcal{L}(V)$ formulas to their local boolean translation.

The goal is thus to define the local boolean translation such that: for any formula φ and any scene $(\mathcal{F}, s)$ where $\mathcal{F}$ is a belief structure, we have that $(\mathcal{F}, s) \models \varphi$ iff $s \models \|\varphi\|_\mathcal{F}$.

Definition 2.13 ([Gat18a]: 2.2.6.). *For any belief structure $\mathcal{F} = (V, \theta, O)$ and any formula $\varphi \in \mathcal{L}(V)$ we define its* ***local boolean translation*** *$\|\varphi\|_{\mathcal{F}}$ as follows.*

1. For the true constant, let $\|\top\|_{\mathcal{F}} := \top$.
2. For atomic propositions, let $\|p\|_{\mathcal{F}} := p$.
3. For negation, let $\|\neg\psi\|_{\mathcal{F}} := \neg\|\psi\|_{\mathcal{F}}$.
4. For conjunction, let $\|\psi_1 \wedge \psi_2\|_{\mathcal{F}} := \|\psi_1\|_{\mathcal{F}} \wedge \|\psi_2\|_{\mathcal{F}}$.
5. For belief, let $\|\Box_i\psi\|_{\mathcal{F}} := \forall V'(\theta' \to (\Omega_i \to (\|\psi\|_{\mathcal{F}})'))$.
*6. For common belief, let $\|C_\Delta\psi\|_{\mathcal{F}} :=$ **gfp**Λ, where Λ is the following operator on boolean formulas modulo equivalence and **gfp**Λ denotes a representative of its greatest fixed point:*
$\Lambda(\alpha) := \forall V'\Big(\theta' \to \Big(\bigvee_{i\in\Delta}\Omega_i \to (\|\psi\|_{\mathcal{F}} \wedge \alpha)'\Big)\Big)$

Aside of the $\mathcal{L}_B$ operators from Definition 2.2, the quantifier operators (universal and existential) also need to have a boolean translation:

Definition 2.14 ([Gat18a]: 1.0.3.). *For any two formulas φ and ψ and any propositional variable p, let $[p \to \psi]\varphi$ denote the result of replacing every p in φ by ψ. For any finite set of propositional variables $A = p_1, ..., p_n$, let $[A \mapsto \psi]\varphi$ denote the result of simultaneously substituting ψ for all elements of A in φ. For any two finite sets of the same size $A = p_1, ..., p_n$ and $B = q_1, ..., q_n$ let $[A \mapsto B]\varphi$ denote the result of simultaneously substituting each q_k for the corresponding p_k in φ for all $k \in 1, ..., n$. Note that strictly speaking A and B need to be ordered lists for this and we use an implicit bijection between them.*
The ***boolean quantifier*** *$\forall p\varphi$ abbreviates $[p \mapsto \top]\varphi \wedge [p \mapsto \bot]\varphi$. For any finite set $A = p_1, ..., p_n$, let $\forall A\varphi := \forall p_1 \forall p_2 ... \forall p_n\varphi$. We define its dual as $\exists p\varphi := \neg\forall p(\neg\varphi)$ which gives us the equivalence $\exists p\varphi \equiv [p \mapsto \top]\varphi \vee [p \mapsto \bot]\varphi$. Similarly for finite sets A, $\exists A$ is the dual of $\forall A$. We also define an "out of" substitution:*
For any two finite sets $A \subseteq B$, let $[A \sqsubseteq B]\varphi := [A \mapsto \top][(B \setminus A) \mapsto \bot]\varphi$.

Roughly stated, the belief operator can be transformed into a boolean function by quantifying out the double vocabulary for states implied by the belief law. For any existing state at the receiving end of a belief relation ($\forall V'$) that obeys the state law (θ'), if that state is implied by the belief (Ω_i) of agent i, then φ holds in that state. It is crucial to use the primed formula ($\|\psi\|_{\mathcal{F}})'$ in this translation for belief, as we check ψ at all reachable states, but it does not have to hold at the starting state: belief might be wrong.

The local boolean translation for the common belief operator is more tricky. Intuitively the $\mathbf{\Lambda}(a)$ is a symbolic encoding of the following process: take a given set of states, a, and remove all the states that do not adhere to the following rule: if there is a receiving state in accordance with theta, then also there must be an agent in Δ that believes the given formula, ψ, holds in that state. Iterating this process causes states with an outgoing belief relation to a state where ψ does not hold to be removed, which potentially results in a smaller new set of states where some states do not have an outgoing relation anymore, since they lost the relations with the previously removed states.

gfpΛ iterates this process until a does not change anymore. We can be sure that such an equilibrium is reached since each iteration either removes at least a state or is equal to the previous state, and seeing we consider a limited number of states (as V is finite, the lattice of boolean formulas modulo equivalence $\mathcal{L}_B(V)/\equiv$ is finite) we will either reach the empty set or a set of states which does not change when applying the process described above. Formally; Λ is monotone and $\mathcal{L}_B(V)/\equiv$ is a complete lattice, thus the Knaster-Tarski Theorem tells us that a greatest fixed point must exist. Its greatest fixpoint can be computed by starting with $\Lambda(\top)$ and then iterating Λ until we reach the first and thereby smallest k such that $\Lambda^k(\top) \equiv \Lambda^{k+1}(\top)$.

Formally, the result of our translation needs to be a formula again and not an equivalence class thereof, hence we let ***gfp***Λ be the representative of $\Lambda^k(\top)$ obtained by reading Λ as a syntactic operator.

It is crucial that Λ is not a syntactic operator on plain formulas, because then it would not have a fixpoint - the formula would just become more and more complex for the (countable) infinite don't-care propositions that logic formulas imply. Intuitively, our symbolic encodings in the form of boolean functions allows reasoning about set of states, which can contain these infinite number of states but can only differ on propositions in a finite context.

The quantification over V' removes all primed propositions. Hence Λ and the whole translation function are both of type $\|\cdot\| : \mathcal{L}_D(V) \to \mathcal{L}_B(V)$. As result, $\boldsymbol{gfp}\Lambda$ applied on $\|\psi\|_{\mathcal{F}}$ gives the symbolic encoding for the set of states where $C_\Delta\psi$ holds.

2.1.4 Transformers

A belief structure can thus represent a Kripke model symbolically, but for $\mathcal{L}_D(V)$ we still need to define an alternative for action models that acts on belief structures. We start by defining (purely epistemic) belief transformers:

Definition 2.15 ([Gat18a]: 2.7.1.)**.** *A **belief transformer** for V is a tuple $\mathcal{X} = (V^+, \theta^+, \Omega^+)$ where V^+ is a set of atomic propositions such that $V \cap V^+ = \emptyset$,*
*$\theta^+ \in \mathcal{L}(V \cup V^+)$ is a possibly epistemic formula called the **event law**, and*
$\Omega_i^+ \in \mathcal{L}_B(V \cup V^+)$ is a boolean formula for each $i \in I$.

*A **belief event** is a belief transformer together with a subset $x \subseteq V^+$, written as $(\mathcal{X}, x)$.*
The belief transformation of a belief structure $\mathcal{F} = (V, \theta, \Omega)$ with $\mathcal{X}$ is defined by $\mathcal{F} \times \mathcal{X} := (V \cup V^+, \theta \wedge \|\theta^+\|_{\mathcal{F}}, \{\Omega_i \wedge \Omega_i^+\}_{i\in I})$. Given a scene $(\mathcal{F}, s)$ and a belief event $(\mathcal{X}, x)$, let $(\mathcal{F}, s) \times (\mathcal{X}, x) := (\mathcal{F} \times \mathcal{X}, s \cup x)$.

The resulting observations are boolean formulas over a new double vocabulary $(V \cup V') \cup (V^+ \cup V^{+\prime}) = (V \cup V^+) \cup (V \cup V^+)'$, describing a relation between the new states which are subsets of $V \cup V^+$.

The single formula $\theta^+ \in \mathcal{L}(V \cup V^+)$ encodes preconditions for all events at once. Within that formula we use the new propositional atoms from V^+ to distinguish different events.

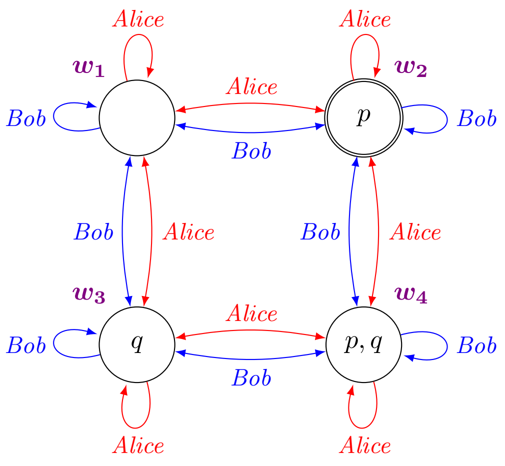
θ^+, if not $\top$, restricts the resulting state law, $\theta \wedge \|\theta^+\|_{\mathcal{F}}$. Recall from the action models, Definition 2.6, that this thus corresponds to the task of the precondition (removing worlds). The precondition is applied over the previous model, before the update. Similarly we want θ^+ to be evaluated over the previous structure $\mathcal{F}$, which is not necessarily the same as evaluating it on the new structure $\mathcal{F} \times \mathcal{X}$. This is why we first localise θ^+ to the boolean equivalent $\|\theta^+\|_{\mathcal{F}}$, since in this scope modal operators are used only indicate *when* events happen, and not *which*.

For example a belief operator $B_i\varphi$ in the event law, semantically corresponds to restricting to the possible worlds where $\Omega_i \to (\varphi)'$ holds when quantified over context V, and not where $\Omega_i^+ \to (\varphi)'$ quantified over context $V \cup V^+$.

In Kripke models we can have different worlds with the same evaluation, but in our defined belief structures we identify states by their evaluation function. Belief transformers are defined therefore to extend the vocabulary such that the difference in relations is characterised by the newly added propositions. This way all states of our structures satisfy different sets of atomic propositions.

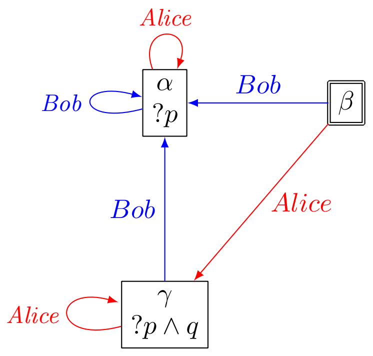
Example 2.3. *Consider a Kripke model where the following transformations will be applied; Alice and Bob get common (correct) belief that p and the agent Alice secretly (falsely) starts believing q. Formally the result becomes: $p \wedge \neg q \wedge B_{alice}\, q \wedge C_{bob,alice}\, p$,*
such that also $\neg B_{bob}\, q \wedge B_{bob} \neg B_{alice}\, q \wedge B_{alice} \neg B_{bob}\, q$

Kripke model: $\mathcal{M}$



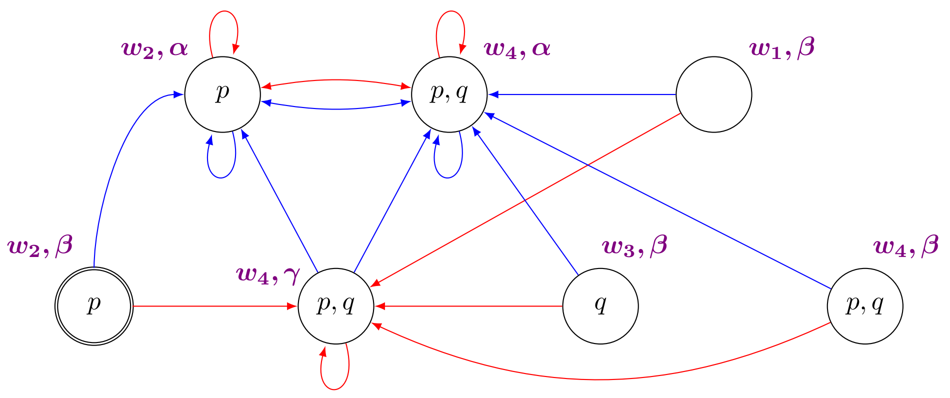
$\times$

action model: $\mathcal{A}$



result: $\mathcal{M} \times \mathcal{A}$

$=$



belief scene, $(\mathcal{F}, s)$*, where:*
$\mathcal{F} = (V = \{p, q\},\ \theta = \top,\ \Omega = \{\Omega_{alice} = \top, \Omega_{bob} = \top\})$
$s = \{p\}$

α	$\neg a \wedge \neg b$
β	$a \wedge b$
γ	$a \wedge \neg b$
not allowed	$\neg a \wedge b$

belief transformer, $(\mathcal{X}, x)$*, where:*
$\mathcal{X} = (V^+ = \{a, b\},$
$\quad \theta^+ = \neg(\neg a \wedge b) \wedge ((\neg a \wedge \neg b) \rightarrow p) \wedge ((a \wedge \neg b) \rightarrow (p \wedge q)) \wedge ((\neg a \wedge \neg b) \rightarrow \top),$
$\quad \Omega^+_{alice} = \neg(a \wedge b)' \wedge ((a \wedge b) \rightarrow (a \wedge \neg b)') \wedge ((\neg a \wedge \neg b) \leftrightarrow (\neg a \wedge \neg b)'),$
$\quad \Omega^+_{bob} = \neg(a \wedge b)' \wedge \neg(a \wedge \neg b)'\ \})$
$x = \{a, b\}$

resulting belief structure, $(\mathcal{F} \times \mathcal{X}, s \cup x)$*, where:*
$\mathcal{F} \times \mathcal{X} = (V = \{p, q, a, b\},\ \theta = \neg(a \wedge b) \rightarrow p,\ \Omega = \{$
$\quad \Omega_{alice} = \neg(a \wedge b)' \wedge ((a \wedge b) \rightarrow (a \wedge \neg b)') \wedge ((\neg a \wedge \neg b) \leftrightarrow (\neg a \wedge \neg b)'),$
$\quad \Omega_{bob} = \neg(a \wedge b)' \wedge \neg(a \wedge \neg b)'\ \}\ )$
$s \cup x = \{p, a, b\}$

Belief transformers, as we have defined them, only change what agents know, not what is actually the case; they do not provide a symbolic equivalent of postconditions for factual change. In product updates of Kripke and action models, the name of a resulting world (w, a_1) specifies its history; it "comes from" w with action/relation a_1. For epistemic action updates we can add propositions from V^+ to the state, but for factual change propositions from V have to be able to be modified. Thus a method for adding propositions to the state law is not enough, removing them is needed as well. Removing a proposition from the state law implies that after the transformation both evaluations for it (e.g. p or $\neg p$) are possible. Because the history of a state, with respect to updates from transformers with factual change, needs to be preserved (e.g. for when an agent does not witness the update event) symbolically describing this update becomes non trivial.

This problem is exemplified in Examples 1.3.5, 2.8.1 and 2.8.2 of [Gat18a].
The solution proposed by Gattinger is to use copies of the previous set of true proposition to store the information of the model before the update. Each old true proposition p is stored in a fresh variable p°, and which propositions are changed are described by V_- and θ_- describes which of those propositions were false. We can then rewrite the state law and observations to represent the factually changed states symbolically.

Note that all belief transformers (which do not have factual change) are exactly all transformers where $V_- = \varnothing$.

Definition 2.16 ([Gat18a]: 2.8.2.)**.** *A belief transformer with factual change, also just called **transformer**, for the vocabulary V is a tuple $\mathcal{X} = (V^+, \theta^+, V_-, \theta_-, \Omega^+)$ where*
V^+ is a set of fresh atomic propositions such that $V \cap V^+ = \varnothing$,
*θ^+ is a possibly epistemic formula from $\mathcal{L}(V \cup V^+)$ called the **event law**,*
*$V_- \subseteq V$ is a subset of the original vocabulary called the **modified subset**,*
*$\theta_- : V_- \to \mathcal{L}_B(V \cup V^+)$ is a map from modified propositions to boolean formulas called the **change law**,*
*$\Omega_i^+ \in \mathcal{L}_B(V^+ \cup V^{+\prime})$ is a boolean formula for each agent $i \in I$ called the **event observation law**.*

To transform a belief structure $\mathcal{F} = (V, \theta, \Omega_i)$ with $\mathcal{X}$, we define a new belief structure $\mathcal{F} \times \mathcal{X} := (V^{new}, \theta^{new}, \Omega_i^{new})$ where

1. $V^{new} := V \cup V^+ \cup V_-^\circ$

2. $\theta^{new} := [V_- \mapsto V_-^\circ](\theta \wedge \|\theta^+\|_{\mathcal{F}}) \;\wedge\; \bigwedge_{q \in V_-} (q \leftrightarrow [V_- \mapsto V_-^\circ](\theta_-(q)))$

3. $\Omega_i^{new} := [V_- \mapsto V_-^\circ][(V_-)' \mapsto (V_-^\circ)'](\Omega_i) \;\wedge\; \Omega_i^+$

*An **event** is a pair $(\mathcal{X}, x)$ where $x \subseteq V^+$. Given a scene $(\mathcal{F}, s)$ and an event $(\mathcal{X}, x)$, let $(\mathcal{F}, s) \times (\mathcal{X}, x) := (\mathcal{F} \times \mathcal{X}, s^x)$ where the new actual state is given by: $s^x := (s \setminus V_-) \cup (s \cap V_-)^\circ \cup x \cup \{p \in V_- | s \cup x \models \theta_-(p)$*

The new vocabulary V^{new} besides V and V^+ now also contains $V_-^\circ = \{p^\circ \mid p \in V_-\}$. These are the fresh copies of the modified subset.

The new state law θ^{new} describes a set of states in the resulting structure, which satisfies the old state law and the event law encoding the preconditions. The preconditions are "checked" in the model before the update, therefore the modified propositions need to be substituted for the old values. The postconditions are encoded by θ_- which is used for overwriting the modified propositions from V_-. The postconditions are also evaluated in the old model (as mentioned in Definition 2.6), thus they are substituted for this as well.

The new set of observations Ω_i^{new} describes a set of relations between states in the resulting structure, which satisfies the old observation law for the substituted vocabulary and the event observation law over the non-substituted vocabulary. Since Ω_i is in a double vocabulary (for encoding the direction of the relations), the propositions are substituted in both vocabularies.

The new actual states s^x describes a set of propositions, which is the union of the following four sets:
$(s \setminus V_-)$; unmodified propositions,

$(s \cap V_{-})^{\circ}$, copies of the modified propositions that were in the old state,
x; the propositions labelling the actual event,
$\{p \in V_{-} \mid s \cup x \models \theta_{-}(p)\}$; the modified propositions whose precondition was true in the old state.

2.1.5 Translating between Kripke models and belief structures

Until now we have only provided a loose interpretation on the correspondence of Kripke models and action models with belief structures and transformers, let's make this formal:

Definition 2.17 ([Gat18a]: 2.6.8.). *For any belief structure $\mathcal{F} = (V, \theta, \Omega)$, we define the* ***corresponding Kripke model*** *$\mathcal{M}(\mathcal{F}) := (W, \pi, R)$ as follows:*
1. W is the set of all states of $\mathcal{F}$.
2. For each $w \in W$, let the assignment $\pi(w)$ be w itself.
3. For each agent i and all $w, w' \in W$, let $R_i ww'$ iff $ww' \models \Omega_i$.

Definition 2.18 ([Gat18a]: 2.6.9.). *For any finite Kripke model $\mathcal{M} = (W, \pi, R)$ we define a* ***corresponding belief structure*** *$\mathcal{F}(\mathcal{M})$ as follows. Without loss of generality we assume unique valuations, i.e. that for all $w, w' \in W$ we have $\pi(w) \neq \pi(w')$. If this is not the case, we can add propositions to V and extend π in such a way that $\pi(w) \neq \pi(w')$ for all $w, w' \in W$. The maximum number of propositions we might have to add is $\lceil log_2 |W| \rceil$. Let $\mathcal{F}(\mathcal{M}) := (V, \theta_{\mathcal{M}}, \Omega)$ where:*

1. V is the vocabulary of $\mathcal{M}$, including extra propositions to make π injective,
2. $\theta_{\mathcal{M}} := \{s \sqsubseteq V \mid \exists w \in W : \pi(w) = s\}$
3. For each i the boolean formula $\Omega_i := \Phi(R_i)$ represents the relation R_i on $\mathcal{P}(V)$ given by $R_i st$ iff $\exists v, w \in W : \pi(v) = s \wedge \pi(w) = t \wedge R_i vw$.

Where:
$\Phi := \bigwedge\{(\ell(a) \sqsubseteq P) \rightarrow pre(a) \mid a \in A\}$,
ℓ is an injective labeling function $\ell : A \rightarrow \mathcal{P}(P)$,
and $\sqsubseteq$ abbreviates a formula which says that out of the propositions in the second argument exactly those in the first argument are true: $A \sqsubseteq B := \bigwedge A \wedge \bigwedge\{\neg p \mid p \in B \setminus A\}$

Definition 2.19 ([Gat18a] 2.9.1.). *The function* ***Act*** *maps transformers to action models as follows. Given an event $(\mathcal{X} = (V^{+}, \theta^{+}, V_{-}, \theta_{-}, \Omega^{+}), x)$, we define an action $(\boldsymbol{Act}(\mathcal{X}) := (A, pre, post, R), x)$ by*

$A := \mathcal{P}(V^{+})$

$pre(a) := [a \sqsubseteq V^{+}]\theta^{+}$

$$post_a(p) := \begin{cases} [a \sqsubseteq V^{+}](\theta_{-}(p)) & \text{if } p \in V_{-}, \\ p & \text{otherwise} \end{cases}$$

$R_i := \{(a, b) \mid a \cup b' \models \Omega^{+} i\}$

where b' denotes a copy of b as in Definition 2.11

Definition 2.20 ([Gat18a] 2.9.2.). *We define the function* ***Trf*** *mapping action models to transformers. Consider an action $(\mathcal{A} = (A, pre, post, R), a_0)$. Let $n := log_2 |A|$ and let $l : A \rightarrow P(\{q_1, ..., q_n\})$ be an injective labeling function using fresh atomic variables q_k.*

Then let $(\boldsymbol{Trf}(\mathcal{A}) := (V^{+}, \theta^{+}, V_{-}, \theta_{-}, \Omega^{+}), l(a_0))$ be the event defined by

$V^{+} := \{q_1, ..., q_n\}$

$\theta^{+} := \bigvee_{a \in A}(pre(a) \wedge l(a) \sqsubseteq V^{+})$

$V_- := \{p \in V \mid \exists a : post_a(p) \neq p\}$

$\theta_-(p) := \bigvee_{a \in A}(l(a) \sqsubseteq V^+ \land post_a(p))$

$\Omega_i^+ := \bigvee_{(a,b) \in R_i}(l(a) \sqsubseteq V^+ \land (l(b) \sqsubseteq V^+)')$

The equivalence proof for transformers the same class of updates as action models is given in Subsection 2.9 of [Gat18a] and is build upon the above definitions. This concludes the explanation for the key components of DEL belief structures, as given in [Gat18a].

2.2 Boolean function implementations and compactness

In the previous section it is described how the task of evaluating $\mathcal{L}_D$ on Kripke models can be reduced to evaluating boolean formulas from $\mathcal{L}_B$ on a (belief) structure. In this section we will discuss different boolean function representation implementations for the semantics of $\mathcal{L}_B$ and how to optimise them with respect to their size.

Optimising boolean function representations is generally done by merging as many similarities as possible and by inferring as much information as possible. This can be done in many ways: we start off by showing what inferences are possible by considering optimised boolean function notations (in Subsection 2.2.1), after which we consider the base case of truth tables as physical implementation for our boolean functions (in Subsection 2.2.2) and how these can merge their similarities such that we get simple decision diagrams (in Subsection 2.2.3). Afterwards we discuss how the simple decision diagrams can be optimised with the inference rules such that we end up with BDDs and ZDDs (in Subsection 2.2.4).

2.2.1 Cover notation and optimisation

Definition 2.21. *A* ***Boolean (valued) function*** *is a function of the type $f : X \to B$, where X is an arbitrary set and B is a Boolean domain, i.e. a generic two-element set, (typically $B \in 0, 1$), whose elements are interpreted as logical values, for example, $0 = false$ and $1 = true$.*

X is the input space of the boolean function. For boolean functions constructed from the boolean translation in SMCDEL, the input space contains all possible boolean assignments over a set of propositional variables (the vocabulary). Using this knowledge we can give a more specific definition for this type of boolean functions:

Definition 2.22. *A* ***Completely Specified Boolean Function (CSF)*** *is a function; $f : B^{|V|} \to B, B = \{0, 1\}$, for $|V|$ given variables. The set of variables, V, is called the* ***context*** *of the boolean function. A CSF is thus a boolean assignment over the complete set of possible boolean assignments over the context. Let the subscripts 1 and 0 denote the* ***1-set*** *and* ***0-set*** *respectively of f (sometimes also called the ON- and OFF- set), where:*
$f_1 = \{s \in \mathcal{P}(V) \mid f(s) = 1\}$ and $f_0 = \{s \in \mathcal{P}(V) \mid f(s) = 0\}$.
For a given input assignment, s, the set of variables $s_1 = \{p \in V \mid s(p) = 1\}$ are called ***the positive literals****, and its complementary set $s_0 = \{p \in V \mid s(p) = 0\}$ are called* ***the negative literals****.*

From here on we use the alternative naming *positive literals* (for s_1) and *negative literals* (for s_0), this re-branding is done to not confuse the 1- and 0-set of a boolean assignment with the 1- and 0-set of a CSF. It also shortens the term "the 0/1 evaluation of the variable". CSFs are denoted often in the form of Boolean Algebra or as covers. We use cover notation in this thesis:

Definition 2.23. *A CSF's 1- or 0- set can be written down as a* ***cover****, a set containing* ***cubes****. Cubes each correspond to a boolean assignment (for a given context, V) where the evaluations of the propositional variables are denoted by* ***literals****. When writing down covers, we use lowercase letters to denote the positive literals, and include an overhead bar to denote the negative literals (see Example 2.4). A* ***full description*** *of all information contained by a CSF thus contains two covers (each for the 0 and 1 set) and a context (set of all variables in the input space). Writing down CSF's in such a manner is referred to as* ***cover notation*** *in this thesis.*

See the first case in Example 2.4.

From here on cover notation is used when denoting a boolean function, even though it can also be identified by its logic formula counterpart. It specifies the context, saves space and indicates when we are talking about a boolean function instead of a logic formula (still to be translated to its boolean function).

Full descriptions are cannonical, but if you allow for inference of information, many forms of cover notation can correspond to the same function. The right form can then provide optimisations with respect to succinct notation of CSF's, which is useful as the methods also apply when considering physical implementations of boolean functions, reducing the memory and computation steps needed.

Example 2.4. *A complete description for any CSF consists of its context, its 1-set and its 0-set of boolean assignments (in which each member contains both its positive and negative literals). The cover notation can be shortened when allowing for inference of information, of which a couple of basic examples are given below for the $\mathcal{L}_B$ formula, $A \wedge \neg B$, with as vocabulary the propositional atoms A, B and C :*

0. complete description:

$f = \{\ f_1 = \{[a\bar{b}c], [a\bar{b}\bar{c}]\}, f_0 = \{[abc], [ab\bar{c}], [\bar{a}bc], [\bar{a}b\bar{c}], [\bar{a}\bar{b}\bar{c}], [\bar{a}\bar{b}c]\}, V = \{a, b, c\}\ \}$

1. inferred 0-set:

$f = \{\ f_1 = \{[a\bar{b}c], [a\bar{b}\bar{c}]\}, V = \{a, b, c\}\ \}$

2. inferred 1-set:

$f = \{\ f_0 = \{[abc], [ab\bar{c}], [\bar{a}bc], [\bar{a}b\bar{c}], [\bar{a}\bar{b}\bar{c}], [\bar{a}\bar{b}c]\}, V = \{a, b, c\}\ \}$

3. inferred context:

$f = \{\ f_1 = \{[a\bar{b}c][a\bar{b}\bar{c}]\}, f_0 = \{[abc], [ab\bar{c}], [\bar{a}bc], [\bar{a}b\bar{c}], [\bar{a}\bar{b}\bar{c}], [\bar{a}\bar{b}c]\}\ \}$

In the literature often only the f_1 cover is considered since the context tends to be a global static and the f_0, in turn, can be inferred.

With the context (V) we can determine all possible boolean assignments (or inputs). This forms a relation with the 1-set and 0-set such that if two components are specified the third can be inferred:

1. All possible inputs ($\mathcal{P}(V)$, where the missing variables are interpreted as negative literals) = True inputs (the 1-set) $\cup$ False inputs (the 0-set)

The context contains all variables that appear in the covers, and when having a complete description, each variable in the context has an evaluation in each cube. Hence, again, one of the three components is infer-able if the other two are known:

2. V (the context's variables) = Positive literals $\cup$ Negative literals (of any cube of a cover from the complete description).

We can utilise this possibility to infer information when representing the CSF in a yet another way, where we only write down the positive or negative literals in each cube. The methods of inferring can also be combined, for example by inferring both the 0-set and the negative literals:

Definition 2.24. *A boolean function (by convention, initially named f in our examples) containing boolean assignments (by convention, initially named s in our examples), can be described by a cover notation where the negative literals and the 0-set is inferred. This can only be done when the context is left specified. Each context variable not present in a cube will then be inferred to be a negative literal.*

*We chose to name this **$f_1 s_1$-optimised description**, since it is often used when denoting a family of sets and it specifies in its notation:*
$f_1 = \{s_1 \mid f(s_1) = 1\}$, *where each* $s_1 = \{s(v) = 1 \mid v \in V\}$

In the literature a cover with only positive literals is sometimes called a ***positive unate*** *cover.*

See case 6 in Example 2.5.

Until now we have worked with f_1s_1-optimised cover notation in our translation of Kripke models to belief structures; we defined states as boolean assignments identified by their set of positive literals ($s_1 = \{s(v) = 1 \mid v \in V\}$), and have defined laws or sets of states as the set of boolean assignments that evaluate to true ($f_1 = \{s_1 \mid f(s_1) = 1\}$). Thus the negative literals ("s_0") and the 0-set ("f_0") are inferred.

When using transformers during boolean translation we may need to introduce new propositional variables for the vocabulary. As a consequence we will have to be able to operate on CSF's with different contexts / (extended) vocabularies. How should our cover notation and their inference methods adapt to this option? We need to include a rule for how we treat the new variables such that the characteristical function is left unchanged. This allows us to define CSF's for an arbitrary context, where we can interpret a boolean formula over all possible contexts (from a countable infinite set of variables):

Definition 2.25. *A CSF, $f = B^{|V_{arb}|} \to B$, over an* ***arbitrary context****, gives $B = \{1, 0\}$ evaluations for every context, $\{v \in V_{arb} \mid V_{arb} \supseteq V\}$, with V as a given* ***fixed*** *(non-arbitrary) context, where:*

the evaluation is determined by 1- and 0-set over all ***deterministic variables****, $v_d \in V$*
and the ***don't-care variables*** *can be inferred $\{v_{dc} \in V_{dc} \mid V_{dc} = V_{arb} \setminus V\}$.*

A CSF over an arbitrary context can always infer its dont-care variables (each variable that does not appear in the fixed part of the context), and the fixed context can either be specified or inferred via the previous methods.

This inference of don't-care variables can be used as another way to reduce cover notation; we can reduce the fixed context to be as small as possible in order to remove mentioning the dc-vars in our cover (see case 8 from Example 2.5), we call this an *optimised fixed context.*

On top of this we can choose to remove pairs of opposite literals in a pair of otherwise equal cubes in the same 1/0-set (See case 7 from Example 2.5). This method infers that a missing literal in a cube does not matter for the evaluation, and thus cannot be combined with inferring the positive or negative literals.

This gives us a combination of four methods (1/0-set inference, pos/neg literals inference, context optimisation and inference, dc-variables inference) with which we can infer information, and thus optimise the succinctness of a boolean description. These methods applicability dependent on each other, some can be combined others cannot. Which combination is most efficient is dependent on what CSF is represented.

Example 2.5. *More examples of optimised cover notation for the $\mathcal{L}_B$ formula, $A \wedge \neg B$, with $V = \{A, B, C\}$:*

5. unspecified context, inferred 0-set:

$f = \{\ f_1 = \{[a\bar{b}c], [a\bar{b}\bar{c}]\}\ \}$

In order to infer both the context and the 0-set, each cube needs to contain evaluations for all variables. This description corresponds directly to the full disjunctive normal form (FDNF) of a corresponding $\mathcal{L}_B$ formula. The cubes are equal to the disjointed terms in the FDNF.

6. specified context, inferred 0-set and negative literals *(f_1s_1-optimised):*

$f = \{\ f_1 = \{[ac], [a]\}, V = \{a, b, c\}\ \}$

7. unspecified context, inferred dc-vars:

$f = \{\ f_1 = \{[a\bar{b}]\}\ \}$

8. optimised and specified context, inferred 0-set and negative literals:

$f = \{\ f_1 = \{[a]\}, V = \{a, b\}\ \}$

These methods of inferring information can also be applied to optimise other representations of boolean functions, as we will show in later sections. We start by considering the implications of physically implementing CSF's with truth tables.

2.2.2 Truth tables as physical implementations.

Even after defining the local boolean translation, the construction of an efficient and correct representation of a boolean function for a given $\mathcal{L}_B$ formula in concrete memory is not trivial. Checking the truth evaluation of $\mathcal{L}_B$ formulas can be done naively by using a truth table as boolean function, where each column either contains a positive or a negative literal of each variable. The columns then directly correspond to a state, which in a Kripke model is a world with its possibly extended vocabulary. This notation is highly inefficient. However, as all boolean function implementations can be represented as truth-tables, it helps understanding how the operators of our language influence them and how improvements are built upon this basis.

Definition 2.26. *A **truth table look up** function represents a physical implementation of a CSF, that stores a boolean output value for each combination of boolean values for the variables in the context. The physical implementation introduces an **ordered context**, $\mathcal{V}$, such that a method along the following lines returns the corresponding evaluation to an given input boolean assignment and a given truth table:*

- *for inputs with a value for each variable of the context;*
- *in order, variable-by-variable the columns that do not agree with the value of the current variable in the input are removed;*
- *the single column left at the end of this process contains the value that the function will return in its last row.*

A truth table is thus comparable to a complete description for a boolean function, it specifies the ordered context in its first column, and it specifies both the 1-set and 0-set as columns with each their corresponding 1 or 0 value stored in the last row (see example 2.6).

Definition 2.27. *The method for removing columns based on values of variables is generally referred to as **restrictSet**, where a given represented CSF, f, removes all assignments which do not agree with a given input boolean assignment, s^*, and removes the variables from its context :*

suppose $\mathcal{V} = (p_1, \ldots, p_n)$, then let
***restrictSet**$(f, s^*) := f_n$, where*

$f_0 := f$, and for each i such that $0 < i \leq n$ we let
$f_i := restrict(f_{i-1}, p_i, s^*(p_i))$, where

$$\begin{aligned}\boldsymbol{restrict}(f, p, 1) &:= \boldsymbol{out}_p(\{s \in f \mid s(p) = 1\}),\\ \boldsymbol{restrict}(f, p, 0) &:= \boldsymbol{out}_p(\{s \in f \mid s(p) = 0\}).\end{aligned}$$

And, lastly, $\boldsymbol{out}_p(f)$ removes the variable p from the context of f, that is; it removes p from all boolean assignments contained in f.

Example 2.6. *Consider storing boolean functions with an ordered context, $\mathcal{V} = (a, b, c)$, in a truth table where the propositions' values are given as rows such that each column is a unique valid boolean assignment, accompanied by its evaluation in the last row:*

$f = A \wedge C$, *in cover notation:*
$f = \{f_1 = \{[abc], [a\overline{b}c]\},\ f_0 = \{[ab\overline{c}][a\overline{b}\overline{c}][\overline{a}bc][\overline{a}b\overline{c}][\overline{a}\overline{b}c][\overline{a}\overline{b}\overline{c}]\},\ \mathcal{V} = (a, b, c)\}$

A	0	0	0	0	1	1	1	1
B	0	0	1	1	0	0	1	1
C	0	1	0	1	0	1	0	1
f	0	0	0	0	0	1	0	1

$\boldsymbol{restrict}(f, a, 1)$, *in cover notation:*
$f = \{f_1 = \{[bc], [\overline{b}c]\},\ f_0 = \{[b\overline{c}][\overline{b}\overline{c}]\},\ \mathcal{V} = (b, c)\}$

B	0	0	1	1
C	0	1	0	1
f	0	1	0	1

$\boldsymbol{restrictSet}(f, s)$ *where* $s = [a\overline{b}c]$, *results in* $\top$ *(with empty context), or in cover notation:*
$f = \{f_1 = \{[\,]\},\ f_0 = \{\},\ \mathcal{V} = ()\}$

f	1

Now we can define the building process for truth table boolean functions of $\mathcal{L}_B$ formulas:

Definition 2.28. *Let the subscripts 1 and 0 denote the 1-set and 0-set of a corresponding boolean function, f, over a vocabulary, V. Each state, $s \in \mathcal{P}(v)$, corresponds to a unique boolean assignment, which is a column of the truth table of f and represents a cube in cover notation:*

	$f =$	$f_1 :=$	$f_0 :=$
0.	g	g_1	g_0
1.	$\top$	$s \in \mathcal{P}(V)$	$\varnothing$
1b.	$\bot$	$\varnothing$	$s \in \mathcal{P}(V)$
2.	*(atomic)* p	$\{s \subseteq V \mid p \in s\}$	$\{s \subseteq V \mid p \notin s\}$
3.	$g \wedge k$	$g_1 \cap k_1$	$g_0 \cup k_0$
3b.	$g \vee k$	$g_1 \cup k_1$	$g_0 \cap k_0$
4.	$\neg g$	g_0	g_1
5.	$g \rightarrow k$	$g_0 \cup k_1$	$g_1 \cap k_0$

As we have seen in the local boolean translation methods from Definition 2.13, we also need the $\forall$ operator. To avoid performing syntactical operations on the given logic formula before building a boolean function, we define a method for abstracting out that fits within the recursive boolean construction method of Definition 2.28:

Definition 2.29. *Let the subscripts 1 and 0 denote the 1-set and 0-set of a corresponding boolean function, f, over a vocabulary, V. The following clause extends the boolean function construction method in Definition 2.28:*

	$f =$	$f_1 :=$	$f_0 :=$
6.	$\forall p \psi$	$\boldsymbol{in}_p\ (\ \boldsymbol{restr}(f_1, p, 1) \rightarrow \psi\ \cap\ \boldsymbol{restr}(f_1, p, 0) \rightarrow \psi)$	$\boldsymbol{in}_p\ (\ \boldsymbol{restr}(f_0, p, 1) \rightarrow \psi\ \cup\ \boldsymbol{restr}(f_0, p, 0) \rightarrow \psi)$
6b.	$\exists p \psi$	$\boldsymbol{in}_p\ (\ \boldsymbol{restr}(f_1, p, 1) \rightarrow \psi\ \cap\ \boldsymbol{restr}(f_1, p, 0) \rightarrow \psi)$	$\boldsymbol{in}_p\ (\ \boldsymbol{restr}(f_0, p, 1) \rightarrow \psi\ \cup\ \boldsymbol{restr}(f_0, p, 0) \rightarrow \psi)$

where $\boldsymbol{in}_p(f)$ adds the variable p to the context of f, that is; each boolean assignment is replaced by 2 new boolean assignments, one containing the positive literal for p and the other the negative literal. The evaluations of the other variables are kept from the old assignment. ***restr*** *is used here instead* *of* ***restrict*** *to fit the table on the page,* $.. \to \psi$ *is used instead of* $.._0 \cup \psi_1$ *to avoid confusion and resemble more familiar notation. Again, for any finite set* $A = p_1, ..., p_n$*, let* $\forall A\varphi := \forall p_1 \forall p_2 ... \forall p_n \varphi$*.*

The methods $\boldsymbol{in}_p$ ($\boldsymbol{restrict}(f, p, 1)$) and $\boldsymbol{in}_p$ ($\boldsymbol{restrict}(f, p, 0)$) do not change the context of f, and are also known as the positive and negative Shannon cofactors [Sha49]. This method for resolving these operators results in the same boolean function as the syntactical method:

Example 2.7. *Consider the formula* $\forall b\ (B \to (A \vee C)) \wedge (\neg B \to C)$*, where the function* f *corresponds to the boolean sub-formula* $(B \to (A \vee C)) \wedge (\neg B \to C)$*.*

When only inferring the 0-set, we write in cover notation:
$f = \{f_1 = \{\ [abc], [\overline{a}bc], [\overline{a}\overline{b}c], [ab\overline{c}], [a\overline{b}c]\}, \mathcal{V} = (a, b, c)\}$

When applying the steps before the intersection, we would get:
$\boldsymbol{restrict}(f_1, b, 1)\ =\ \{f_1 = \{\ [ac], [\overline{a}c], [a\overline{c}]\},\ \mathcal{V} = (a, c)\}$
$\boldsymbol{restrict}(f_1, b, 0)\ =\ \{f_1 = \{\ [ac], [\overline{a}c]\},\ \mathcal{V} = (a, c)\}$

Thus when applying $\forall b$ *we get:*
$\forall b\ f = \{f_1 = \{[ac], [\overline{a}c]\},\ \mathcal{V} = (a, b, c)\}$*, which can be written in* $\mathcal{L}_B$ *as* $(A \vee C) \wedge C$ *or just* C

This is clearly a reduced version of the result obtained by applying the substitution discussed in Definition 2.14: $((\top \to (A \vee C)) \wedge (\bot \to C)) \wedge ((\bot \to (A \vee C)) \wedge (\top \to C))$

Now that we have defined methods for recursively building physical implementations for our logic, lets conclude this section with a brief look forward to the upcoming improvements. The truth table function, as we defined it, has its context, its positive and negative literals and both its 1-set and 0-set explicitly stored. It effectively has to store each specific state this way. We can optimise on this by introducing a method for symbolic representations of sets of states instead (merging along similarities of the states), which will be achieved with Simple Decision Diagrams, as given in the next section. On top of that we can use the inference methods (as discussed for cover notation in Subsection 2.2.1) such that we end up with BDDs and ZDDs, which will be discussed in the section afterwards.

2.2.3 Simple Decision Diagrams

Simple Decision Diagrams (SDD) are introduced here as a basis for the more efficient BDDs and ZDDs, they just serve an explanatory purpose here. BDDs and ZDDs are subclasses of SDDs; the reduction rule of SDDs also hold for BDDs and ZDDs. In practice SDDs are not used, in the literature they are sometimes refered to as Quasi-Reduced Ordered Binary Decision Diagrams.

Instead of keeping track of all possible states with their evaluation (2^{n+1} combinations, when noting both true and false sets), we can group sets of states based on their similarities. This can be done by rewriting the truth table as a graph where multiple table cells can be represented as a single node inside the graph:

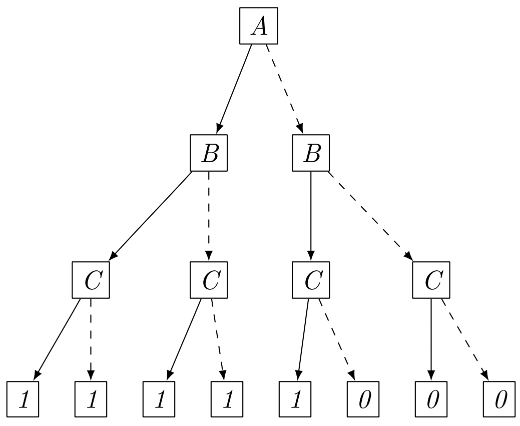
Definition 2.30. *A* ***Simple Decision Diagram*** *(SDD) is a rooted, directed, acyclic graph, for a given ordered list of variables (the ordered context:* $\mathcal{V}$*), with a boolean evaluation as each path's terminal node. The variables represent nodes at each level of the graph, where each node has 2 labelled edges for either the 1 evaluation (THEN edge) or its complement, 0 (ELSE edge). Each path thus represents a boolean assignment from* $\mathcal{P}(V)$*. After constructing the initial graph the following* ***reduction rule*** *is applied: Merge any isomorphic sub-graphs.*

Each path in the graph corresponds directly to a column in the truth table. The nodes' edges effectively represent the negative and positive literals from the truth table. The decision diagram before applying the reduction rule takes form of a binary decision tree and can be seen as a truth table where all cells that share evaluations on the same propositional variable and on all propositional variables above them in the given order are merged into a single node.

Example 2.8. *Consider the boolean function f for the boolean formula $A \wedge (\neg A \rightarrow (B \wedge C))$ with the ordered context $\mathcal{V} = (a, b, c)$. The states and their evaluations for f can be visualised as a truth table where each state is represented as a column.*

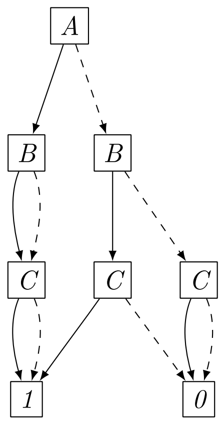
A	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*
B	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*0*
C	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*
f	*1*	*1*	*1*	*1*	*1*	*0*	*0*	*0*

When considering f in its "tree" form, and afterwards merging the isomorphic sub-graphs to get f's SDD form, we get:



Binary decision tree

$\Longrightarrow$



Merge isomorphic sub-paths

When drawing the graphs we consider the solid arrows to represent the THEN edges and the dashed arrows to represent the ELSE edges. SDDs have a fixed context (no inference rule for missing literals) which can be inferred from each path (as all variables are present). The paths in an SDD are thus equal to the cubes in a cover notation with only its context unspecified:

$f = \{f_1 = \{[abc][ab\overline{c}][a\overline{b}c][a\overline{b}\overline{c}][\overline{a}bc]\},\ f_0 = \{[\overline{a}b\overline{c}][\overline{a}\overline{b}c][\overline{a}\overline{b}\overline{c}]\},\ \textit{unspecified}\ \mathcal{V}\}$

An added benefit is that SDDs and its extensions are *canonical*: for a given variable order, an SDD describes functions in a single way only. In contrast you can have many different formulas and covers describing the same function. Storing functions in cannonical structures allows for quick equivalence checks between them.

Operating on Decision Diagrams

Using SDD-like structures we can now represent as paths, states that are considered valid for a local boolean translation of any given DEL formula.

Before we considered inputs to our boolean functions only as boolean assignments (a single state), for which in SMCDEL the evaluation represents the validity of that state. We can also define methods for reporting the validity of a set of states:

Definition 2.31. *Applying the local boolean translation methods (for a given scene, $(\mathcal{F}, s)$, storing information of the vocabulary and agents) to a DEL formula φ (as described in Definition 2.12), and then constructing a SDD for it results in the following equivalences:*

Truth:
$(\mathcal{F}, s) \models \varphi \leftrightarrow \boldsymbol{restrictSet}(\boldsymbol{SDD}(\|\varphi\|_{\mathcal{F}}), s) = \boldsymbol{SDD}(\top)$ *(with empty context)*

Validity:
$\mathcal{F} \models \varphi \leftrightarrow \boldsymbol{SDD}(\theta \rightarrow \|\varphi\|_{\mathcal{F}}) = \boldsymbol{SDD}(\top)$ *(with unchanged context)*

When constructing a SDD from a DEL formula, not only the boolean translation is dependent on the belief structure, but also the SDD. This is because the context of the boolean formula (stored by the vocabulary in the belief structure) changes the resulting SDD. The correct way of writing would be to add a $\mathcal{F}$ as super script to $\boldsymbol{SDD}(\|\varphi\|_{\mathcal{F}})$, but this would be too cluttering (considering we also use the subscript to indicate the 1 and 0 set) and in SCMDEL we alway operate on the same belief structure thus it provides no extra information. If the logic formula φ only uses boolean operators we can leave out the boolean translation notation as well.

Reading out validity for single state inputs in SDDs resembles the truth table look up method given in 2.26, but a method for reading out the validity of a set of states would use an implication function implemented along the lines of:

Example 2.9. *Example simultaneous traversal algorithm for implication checks between two CSF (with equal fixed contexts), each representing a set of states:*

```
set Implication ( set A, set B, list Context )
{
- - consider trivial cases
if ( A = {} ) return Top
if ( A = B ) return Top
if ( A = CreateSDD( Top, context ) ) return B

- - when allowing additional calls to other traversal algorithms we can add the following trivial case:
if ( B = {} ) return complement( A )
if ( B = CreateSDD( Top, context ) ) return complement( A )

- - get the the next variable of A and B
var x = Get_first ( context )

- - restrict w.r.t. x
set A0 = restrict( A, x, 0 )
set A1 = restrict( A, x, 1 )
set B0 = restrict( B, x, 0)
set B1 = restrict( B, x, 1 )

- - recursively solve subproblems, apply negation to first term
set R0 = implication( A1, B0, remove(x, Context) )
set R1 = implication( A0, B1, remove(x, Context) )

- - union between the two terms, such that x = 1 implies R1 and x = 0 implies R0
set R = CreateSDD( x, R1, R0, context)

- - return the result
return R
}
```

As result $\boldsymbol{SDD}(f \rightarrow g)$ corresponds to $\boldsymbol{SDD}(f) \rightarrow \boldsymbol{SDD}(g)$ with the implication as defined in 2.28.

The algorithms that traverse and perform computations on SDD-like structures have been subject to much research, the definition above is only a sketch for the general task that needs to be performed.

These types of algorithms are called *traversal algorithms*, as opposed to the non-recursive methods that do not use the SDD structure directly (but may call the recursive procedures). A simultaneous depth-first traversal is usually used when a traversal algorithm takes two SDD structures as arguments.

The physical implementation for the set operators (as we used liberally for defining the translation of boolean logic to boolean structures) are also classified as traversal algorithms. A general implementation of these operators for BDDs is discussed in [DS01], and for ZDDs in [Min01].

There are many different implementations available, each with their own efficiency trade offs. Discussing all the possible improvements for operations on decision diagrams is outside the scope of this paper, they also differ for each version of decision diagrams, such as BDDs and ZDDs.

The most common improvement applied to these algorithms is the use of hash tables to store intermediate results, such that during depth-first recursive procedures on the SDD structure each node only has to be visited once.

Another method for optimising the compactness of the structures is by improving on the variable order. There is no direct way of determining the best order, but many metrics and methods have been found for both static variable ordering [HAA18][AMS04][Jai+94][SC06], where the order does not change during manipulation of the graph, and dynamic ordering [Rud93] [PSP94] [IMZ07], where the order is optimised during manipulation.

2.2.4 BDD's and ZDD's

The previous section discussed advantages of SDDs for boolean reasoning, but we can do even better; further optimisations can be achieved by removing nodes from the graph whenever these can be inferred. This can be done with similar methods as introduced for inferring information for cover notation, i.e.: we could remove the nodes that do not influence the truth value of the path and infer them (inferring dc-vars), or specify only the positive nodes that cause the truth value of a sub path to be false and infer those (f_1s_1-optimised). The SDDs using one of these two rules of inference are called BDDs or ZDDs respectively.

Definition 2.32. *A* ***Binary Decision Diagram*** *(BDD) is a SDD structure where the following* ***BDD elimination rule*** *is applied: nodes whose two children are isomorphic are eliminated. When evaluating an input on a BDD the following inference is used: whenever a variable is not found in the traversed path, it is treated as a dont-care variable.*

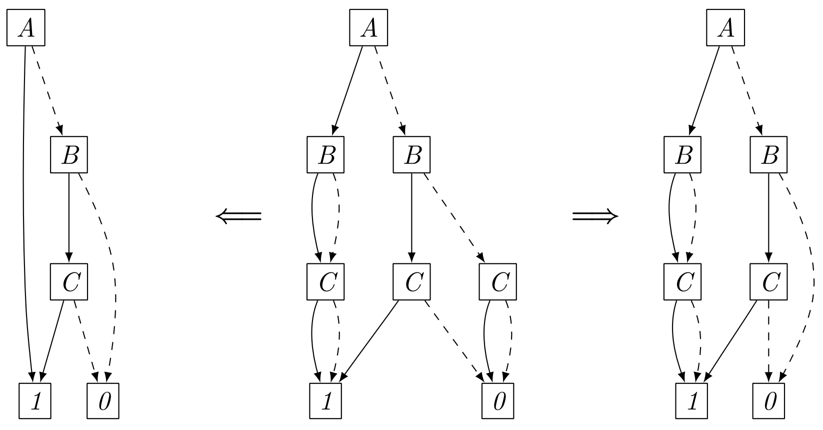
The eliminated nodes do not have to be stored as they can be inferred during the traversal of the graph due to its ordered property. When a variable is encountered in the input which is not present in the corresponding path, it is treated as a dont-care variable and the traversal algorithm skips it.

Definition 2.33. *A* ***Zero-suppressed Decision Diagram*** *(ZDD) is a SDD structure where the following* ***ZDD elimination rule*** *is applied: if the THEN edge of a node leads to the False node, it is eliminated. When evaluating an input on a ZDD the following inference is used: whenever a variable node is not found in the traversed path (leading to the Truth leave node), it is treated as a negative literal.*

Like BDDs, the eliminated nodes can be inferred during the traversal of the graph due to its ordered property; when a positive literal from the input is not encountered in a path, a 0 is returned immediately as evaluation, and when a negative literal from the input is not encountered during ordered traversal of the graph of a ZDD, it is assumed to be correct and the process proceeds with checking the next variable evaluation.

Example 2.10. *Take for example a SDD, BDD and ZDD construction for $\varphi = A \vee (B \wedge C)$ with context*

$\mathcal{V} = (a, b, c)$.



$\boldsymbol{BDD}(\varphi)$ $\boldsymbol{SDD}(\varphi)$ $\boldsymbol{ZDD}(\varphi)$

Liaw et al. [LL92] and Wegener [Weg94] have shown that in general (for random Boolean functions), the merging rule makes a much more significant contribution to storage saving (with a factor of $1/|V|$ in the worst-case size) than the BDD elimination rule (contributing no more reduction than 1% for large $|V|$). This is partly because the elimination rule is applied before merging. If we were to infer nodes based on the $f_1 s_1$-optimised form or inferred dc-vars form first and afterwards apply the merge rule we would get the same result as just constructing a ZDD or BDD in the normal way (with the elimination rules applied after merging), but the inference rules would have been responsible for more node reductions than the elimination rules in this view.

When comparing the compactness of BDDs and ZDDs, Knuth [Knu11] has shown that for any function, f, the relative size (including the leaf nodes) of these decision diagrams to each other have a bound of:

$$\begin{array}{lcl} \boldsymbol{size}(\boldsymbol{BDD}(f)) \;/\; \boldsymbol{size}(\boldsymbol{ZDD}(f)) & \leq & n/2 + o(n) \\ \boldsymbol{size}(\boldsymbol{ZDD}(f)) \;/\; \boldsymbol{size}(\boldsymbol{BDD}(f)) & \leq & n/2 + o(n) \end{array}$$

These bounds show a significantly large difference in compactness (with a factor of $n/2$) between ZDDs and BDDs, where n indicates the number of unique variables in the context of f and $o(..)$ represents the little o notation.

The trade off between them is dependent on how many nodes can be eliminated. BDD's are more efficient when there are many nodes that do not influence the truth value of a formula (non-deterministic nodes). ZDD's are more efficient when there are many deterministic nodes in a few set of paths, of which many are negative literals pointing towards a the False terminal node.

These are only heuristics for comparing their efficiency, as the actual graph size is also dependent on the merging rule, which can play out differently depending on the graph (due to variable ordering). The number of nodes eliminated by the BDD or ZDD elimination rule could have been reduced by the merging, which in turn could change which elimination rule removed more nodes from the total structure. Also here no method, faster than construction, exists for determining which reduction rule optimally reduces a decision diagram.

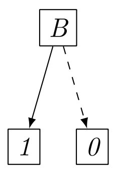
2.2.5 ZDD context dependency

Since ZDDs and BDDs do not explicitly specify their fixed context but still use inference rules for any variables not contained by them, there are consequences to representing logic systems as these structures. Particularly ZDDs contain a type of context dependency, in contrast to BDDs which can handle inputs of different contexts yet still represent the same characteristical function (recall Definition 2.25).

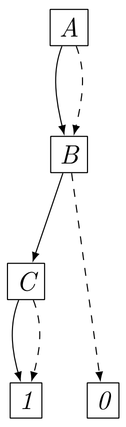
In this section we discuss un-intuitive scenarios for ZDDs due to context dependency, and aim to clearly illustrate the consequences and connection between ZDD structures and their inference rule. For this we first show and later refer to the following examples of basic boolean functions for ZDDs and BDDs:

Example 2.11. *The inference rules do not correspond one-to-one with the BDD and ZDD elimination rules as BDDs and ZDDs also merges nodes whenever possible and this "interferes" with the inference rules. Cover cubes are thus not directly equivalent to ZDD/BDD paths, but cover notation is included in these examples for highlighting the similarities and differences.*

Consider the BDDs and ZDDs for the following basic formulas in $\mathcal{L}_B$:

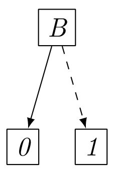


BDD(B)

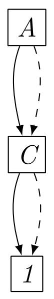


ZDD(B)

dc-var inferred: $f_1 = \{[b]\}$, *unspecified* $\mathcal{V}$
$f_1 s_1$*-optimised:* $f_1 = \{[abc][ab][bc][b]\}$, $\mathcal{V} = (a, b, c)$

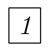


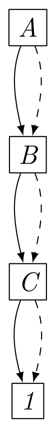
BDD$(\neg B)$



ZDD$(\neg B)$

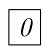
dc-var inferred cover: $f_1 = \{[\overline{b}]\}$, *unspecified* $\mathcal{V}$
$f_1 s_1$*-optimised cover:* $f_1 = \{[ac][a][c][\,]\}$, $\mathcal{V} = (a, b, c)$





BDD$(\top)$ ***ZDD***$(\top)$

dc-var inferred : $f_1 = \{[\,]\}$, *unspecified* $\mathcal{V}$
$f_1 s_1$*-optimised :* $f_1 = \{[abc][ab][ac][a][bc][b][c][\,]\}$, $\mathcal{V} = (a, b, c)$



empty-ZDD

BDD$(\bot)$ ***ZDD***$(\bot)$

dc-var inferred : $f_1 = \{\}$, *unspecified* $\mathcal{V}$
$f_1 s_1$*-optimised :* $f_1 = \{\}$, $\mathcal{V} = (a, b, c)$

Propositional atoms as ZDD

Because ZDDs do not eliminate nodes representing dont-care variables, they are particularly ill-suited for recursively constructing from $\mathcal{L}_B$ formulas. Each atomic proposition in such a formula can be represented as a CSF over an arbitrary context, only stating that the variable corresponding to the propositional atom leads to True and otherwise it leads to False, every other variable is seen as a dont-care variable. ZDDs explicitly have to keep of all these dont-care variables in their paths leading to True, otherwise the paths would infer them to be negative literals. See the first 2 cases in Example 2.11.

This means that during the atomic steps of recursively constructing, each such "atomic" ZDD has a lower bound of nodes of $n - 1$, whereas BDDs have just 1 (where n is the number of variables in the context, the leaf nodes not included). To improve on this we would have optimise the fixed context of the ZDD as in case 8 of Example 2.5.

Introducing new propositions and items

If the fixed context of ZDDs is left unspecified every unknown variable in a path is inferred to be a negative literal which is undesirable as we have discussed how translating DEL-logic to boolean logic may need to add new variables to the vocabulary (the context).

This forces us to keep the fixed context of a ZDD specified and provide methods for updating each ZDD's fixed context to the union of both. Then every unknown variable outside the fixed context can be interpreted as a dont-care variable, and the unknown variables inside the fixed context as negative literals. Afterwards the usual traversal-algorithms can be used for operating on ZDDs. This additional method is not needed for BDDs.

There is a flip side to this though, when we represent models with a potentially infinite range of items instead of propositions. If we interpret each item as a positive literal, a finite set of items from an undefined (thus potentially infinite) range, ZDD representations are context independent instead and BDD representations are context dependent - otherwise BDDs would have to specify all (infinite) negative literals.

Thus if we use the dont-care inference rule (BDDs) we can leave the context unspecified for interpreting boolean algebra, combinational logic and class membership. If we infer the negative literals (ZDDs), we can leave the context unspecified for sets of agents, sets of numbers or other types of instances that come from an undefined range.

Negation for ZDDs

ZDDs without their fixed context specified cannot infer their 0-set, this means that the negation operation (swapping the 1 and 0 sets) is non trivial. This can be fixed by taking the difference of the ZDD to-be-negated with the universal ZDD (a ZDD with all paths in the 1-set, in our case the Top ZDD, constructed for the complete vocabulary stored in the belief structure). The universal ZDD then acts as a psuedo fixed context, where taking the difference with the original ZDD (describing the 1-set) results in the negated ZDD (describing the 0-set). Again a ZDD with size related to the fixed context needs to be used to perform a basic operation. Were standard implementations for ZDDs designed with a specified context in mind, a more efficient traversal algorithm could be used for this operation.

In comparison, for BDDs the 0-set is the same as the 1-set but with the 0 and 1 leaf nodes swapped everywhere. Some methods that further reduce BDDs and ZDDs by complementing is elaborated on in the next section, here only the problem for applying negation to ZDDs (without specified context or universal ZDD) is shown.

Bot as ZDD

In the case that all paths end up at the False terminal node, the function is by Definition (2.28) representing Bot. ZDDs usually eliminate the 0 leaf node if there are no edges leading to it (after node elimination), otherwise you would have a floating 0 with all ZDDs containing only negative literals and dont-care variables (see second case of Example 2.11). The Bot ZDD has all its nodes eliminated, since every node has a THEN edge leading to the 0 node. Normally you would expect the Bot ZDD to contain only the 0 node (like the Bot BDD), but, for efficiency's sake, we do not want to preform a separate check whether the 0 node needs to be added afterwards if the function represents the Bot ZDD.

Thus, in practice, ZDDs are constructed such that in the case that all variables are set to false (all-negative-literals), all nodes are eliminated aside of the Truth node, and whenever we have Bot, a ZDD is completely empty. We can visualise this difference in cover notation as the Bot ZDD's 1-set representing an empty cover, whereas an all-negative-literals ZDD's 1-set represents a cover with an empty cube (see the last case in Example 2.11).

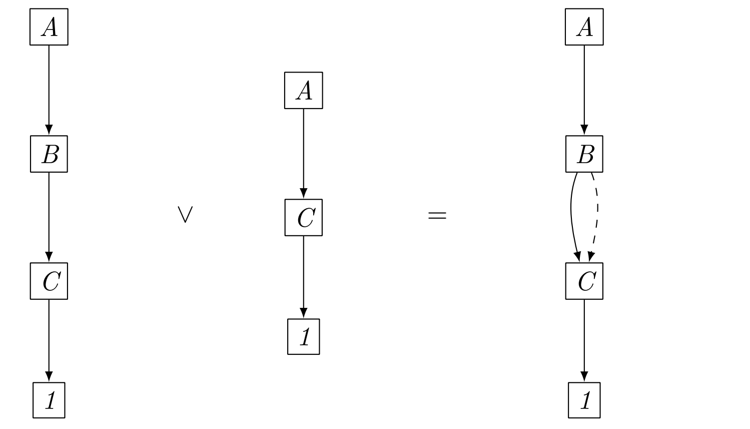
Again this corresponds to the idea that ZDDs represent a family of sets, in which the all-negative-literals function corresponds to a family containing an empty set and the bot function corresponds to an empty family.

Cover notation

By comparing ZDD to $f_1 s_1$-optimised covers, we imply that the paths in ZDDs represent states where the negative literals are removed. Yet the paths in the graph do also contain negative literals.. How come?

These are consequences of the merge rule. Two paths in the same ZDD that differ on a literal sharing all previous variable evaluations, cannot show the difference between the two paths if you do not allow the node to contain an ELSE edge. If we treat every path separately (thus without merging), the ZDD elimination rule exactly gives us the nodes/paths (not counting the leaf nodes) equal to the literals/cubes in the $f_1 s_1$-optimised cover.

Example 2.12. *Take for example two ZDDs representing two paths, $f_1 = \{\, [abc] \,\}$ and $g_1 = \{\, [a\bar{b}c] \,\}$ with $\mathcal{V} = (a, b, c)$, the ZDD containing both those paths, $h_1 = \{\, [abc],\ [a\bar{b}c] \,\}$, needs to have an ELSE edge to indicate the difference. It cannot eliminate the node with the $\bar{b}$ edge.*

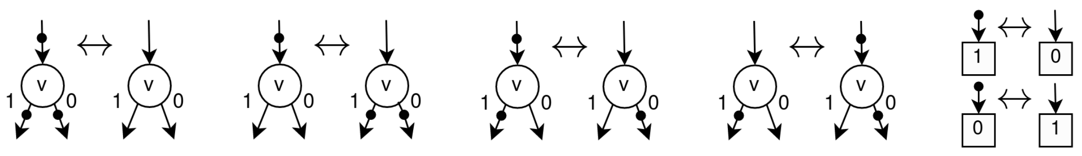


path 1: $f_1 = \{\, [abc] \,\}$ *path 2:* $g_1 = \{\, [a\bar{b}c] \,\}$ *combined:* $f_1 \cup g_1 = \{\, [abc],\ [a\bar{b}c] \,\}$

The edges that lead to the False node are left out for clarity sake. Leaving them out does not change the characteristic function the graph describes if inference is used when an edge is not present (much like inferring a node that is not present). This method of drawing is also used by some literature and ZDD libraries (like CUDD).

2.2.6 Complementing sub-graphs

A further improvement on BDD's is usually realised by adding rules allowing for complement edges. Akers [Ake78] first described using complement edges for hand-generated BDDs. Sets of rules were formulated in Karplus [Kar88] and Madre [MB+88] to guarantee canonical BDDs using complement edges. A complement edge is an ordinary edge with an extra bit (complement bit) set to indicate that the connected subgraph is to be interpreted as its complement. Adding complement edges to BDDs can be applied to 6 patterns in the graph, while preserving the characteristic function. 4 patterns for non-terminal nodes in the graph, and 2 patterns of an incoming complement edge to the leaf nodes, which are equal to a non-complement edge leading to the terminal node with an inverted truth value. The traversal algorithms forward the complement bit according to these patterns until the terminal nodes have been reached.



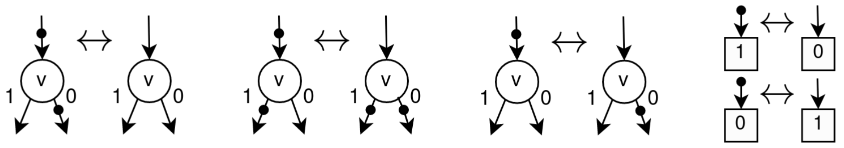
4 patterns of non-terminal nodes for recursively interpreting complement edge cases, and 2 terminal node patterns.

Consider two BDD nodes which are isomorphic if their children nodes are interchanged. These two nodes can then be merged as a single node with a complement edge leading to the parent whose child's nodes should be interchanged. Each time this is applied it reduces the BDD graph with one node. We still have to specify which child should be inverted in order to preserve the cannonical property. For this the simple rule is added that whenever we use the complement bit we apply it to the ELSE edge. Applying complement bits on all edges (first pattern) does not result in any node merging, thus cannot be used for reducing the size. Note that adding interpretation of complement edges means only one terminal node is needed, a complement edge can invert the truth value if needed.

Adding the complement bit to the top of a subgraph results in the same function as applying negation (the logic operator) to that subgraph, thus it also provides "free" computation for that operator. This reflects the statement of the previous section that a negated BDD is the same as the original BDD with its leaf nodes swapped.

At first glance negation cannot be applied to provide complement edge optimisation for ZDDs, as the node elimination rule could then remove nodes with a complement edge. Consider the following case: when a node with its THEN edge leading to 0 contains a complement edge, it cannot be reduced since then the position of the complement edge is lost. This restricts the freedom of 'applying complement edges wherever it causes a merge' with 'applying ZDD elimination', or vice-versa. No matter which method is first applied, the second will change its application based on the graph's contents, resulting in non-canonical ZDDs.

Another type of complement edges, named 0-element edge [Min93], circumvents this problem for ZDDs; this complement bit is only forwarded to the ELSE edge. This complement bit thus does not equal negation and it results in 3 patterns for non-terminal nodes. The pattern with the complement bit on the incoming edge and the ELSE edge (first pattern of image below) does not result in any node merging, thus it is not used.



ZDD interpretation of complement bit patterns.

We do not show the complement optimisations in later drawings of decision diagrams in this thesis.

Generalising ZDD elimination rules

The original method of complementing can result in cannonical ZDD graphs still for 2 patterns; when applying the complement edge on the top most level, reversing the 0 and 1 evaluation for each path in the ZDD tree (equal to the difference with the Top ZDD), or complementing each node in the tree. If we apply this change the elimination rule has to be inverted accordingly. The original ZDD elimination rules infers the negative literals and the 0-set whenever possible, although this is an arbitrary choice and the positive literals and 1-set could also be inferred instead. All options combined gives us 4 patterns in total for reducing nodes in ZDDs. This generalises the ZDD elimination rules to:

Definition 2.34. *We can define 4 ZDD elimination rules that preserve the characteristical function and the canonical property. Starting with the standard variant we get:*

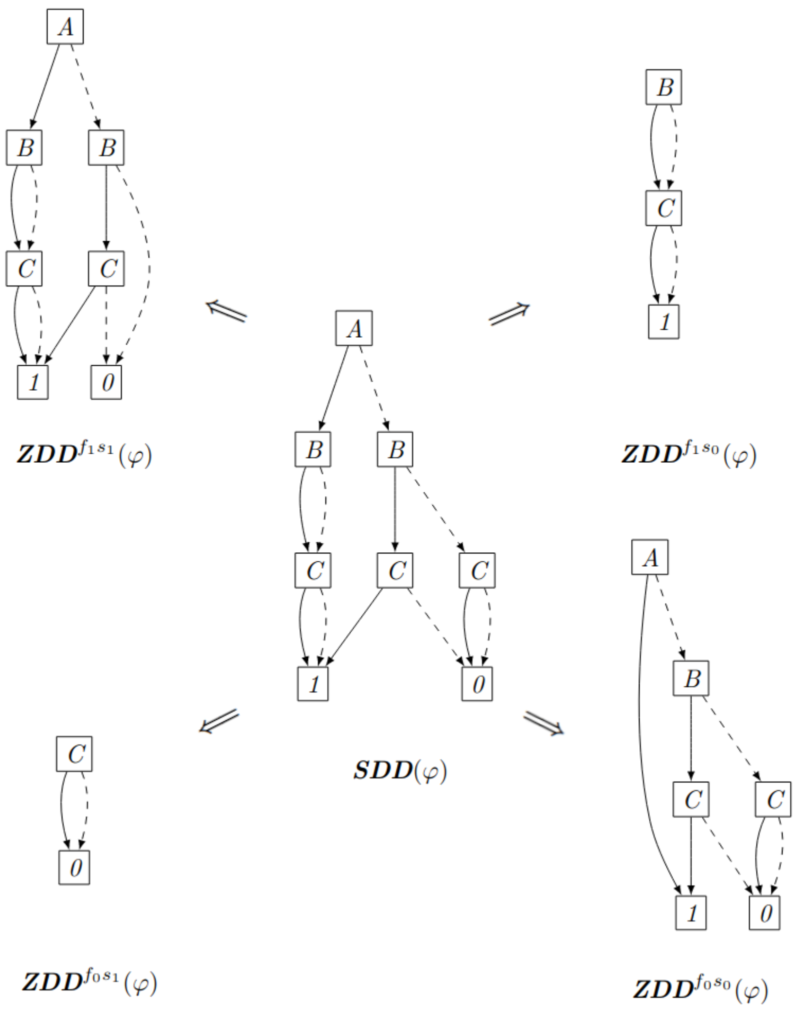
1. *"if THEN of n leads to 0, eliminate n."*
   Inferring the negative literals and the 0-set:
   $ZDD^{f_1 s_1} := f_1 = \{s_1 \mid f(s_1) = 1\}$ *where* $s_1 = \{s(v) = 1 \mid v \in V\}$
2. *"if ELSE of n leads to 0, eliminate n."*
   Inferring the positive literals and the 0-set:
   $ZDD^{f_1 s_0} := f_1 = \{s_0 \mid f(s_0) = 1\}$ *where* $s_0 = \{s(v) = 0 \mid v \in V\}$
3. *"if THEN of n leads to 1, eliminate n."*
   Inferring the negative literals and the 1-set:
   $ZDD^{f_0 s_1} := f_0 = \{s_1 \mid f(s_1) = 0\}$ *where* $s_1 = \{s(v) = 1 \mid v \in V\}$
4. *"if ELSE of n leads to 1, eliminate n."*
   Inferring the positive literals and the 1-set:

$$ZDD^{f_0 s_0} := f_0 = \{s_0 \mid f(s_0) = 0\} \text{ where } s_0 = \{s(v) = 0 \mid v \in V\}$$

n is any node in the graph. The different variants are identified by using $f_x s_y$, since they are characterised by their underlying inference rule, for which we have used f_x to indicate the 1- or 0-set of a function, and we have used s_y to indicate the positive or negative literals of a boolean assignment. (where x and $y \in \{0, 1\}$)

Which kind of inferences are made asymmetrically influences which nodes are eliminated and thus can improve the compactness of the ZDD significantly.

Example 2.13. *Consider a SDD for $\varphi = A \vee (B \wedge C)$ with $\mathcal{V} = (a, b, c)$.*



It is clear that the 4 variants of the ZDD elimination rule correspond to the combinations of the two methods for complementing the ZDDs (complementing the entire function or complementing all literals). These different variants of ZDDs and their relation to negation in formulas gives us the following conversion methods:

Definition 2.35. *The following graph-identity equivalences hold for the ZDD variants described in Definition 2.34:*

$$\begin{array}{ccccc}
\boldsymbol{ZDD}^{f_1 s_1}(\neg\varphi) & \leftrightarrow & \neg\boldsymbol{ZDD}^{f_1 s_1}(\varphi) & \leftrightarrow & \boldsymbol{ZDD}^{f_0 s_1}(\varphi) \\
\boldsymbol{ZDD}^{f_1 s_1}(\forall p\,[p \mapsto \neg p]\;\varphi) & \leftrightarrow & \boldsymbol{swap}\;(\boldsymbol{ZDD}^{f_1 s_1}(\varphi)) & \leftrightarrow & \boldsymbol{ZDD}^{f_1 s_0}(\varphi) \\
\boldsymbol{ZDD}^{f_1 s_1}(\forall p\,[p \mapsto \neg p]\;\neg\varphi) & \leftrightarrow & \boldsymbol{swap}\;(\neg\boldsymbol{ZDD}^{f_1 s_1}(\varphi)) & \leftrightarrow & \boldsymbol{ZDD}^{f_0 s_0}(\varphi)
\end{array}$$

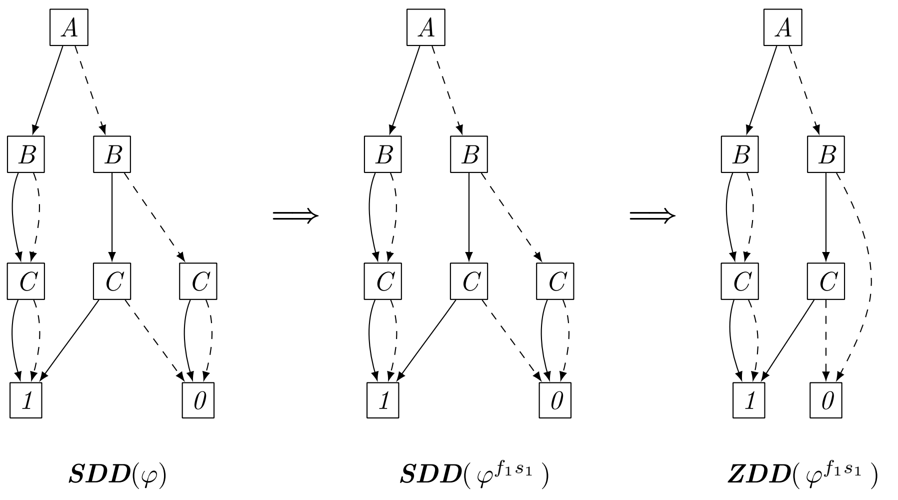
*where **swap** flips all out-going edges (children) of each node in the ZDD, and $\neg$ applied to a boolean function swaps the 0- and 1-set (inverts the False and True leaf nodes) as in Definition 2.28.*

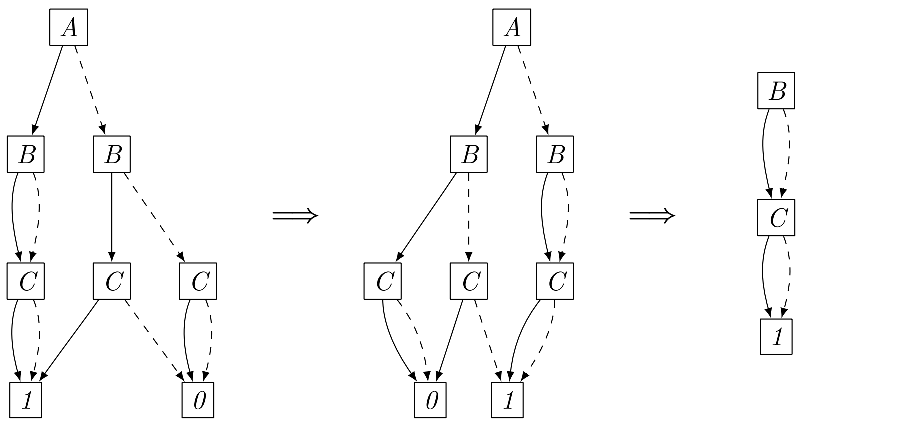
From these equivalences we obtain methods for conversion between all variants. These can easily be deduced from Definition 2.35 and thus will not be mentioned here. Conversion via adjusted interpretation of the logic formulas is useful for libraries that do not support generalised rules for ZDDs, like CUDD.

We want to be able to build these ZDD variants for a logic formula without first making syntactical changes to the logic formula or apply changes to the ZDD after its build. This is possible: if we were to interpret every propositional atom and its directly negated version as each other it would result in the same ZDD as when swapping all children nodes, and if we were to interpret every operator as its dual on top of the previous interpretation we get the same as if we were to negate the entire formula.

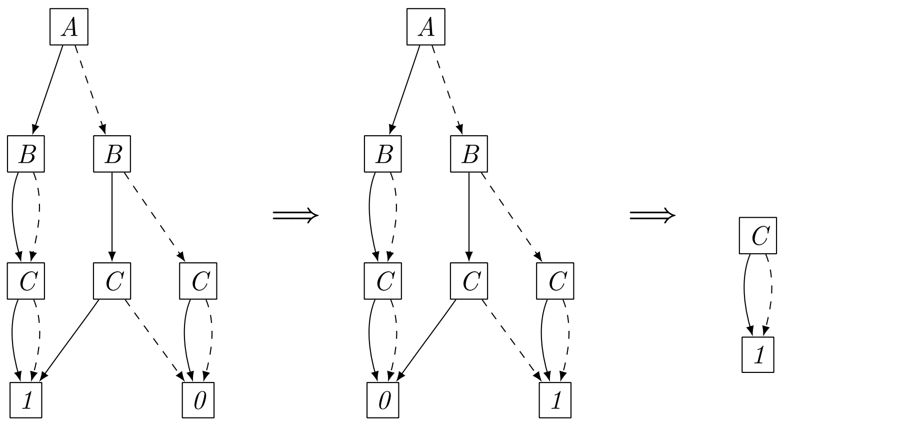
As long as the ZDDs given as arguments in traversal algorithms are of the same type, the same semantical results are produced.

Example 2.14. *Take for example a SDD representing φ with $\varphi = A \lor (B \land C)$ and $\mathcal{V} = (a, b, c)$. We can change interpretation of the formula (indicated here by a superscript on φ) such that the ZDDs are build as each of the variants.*

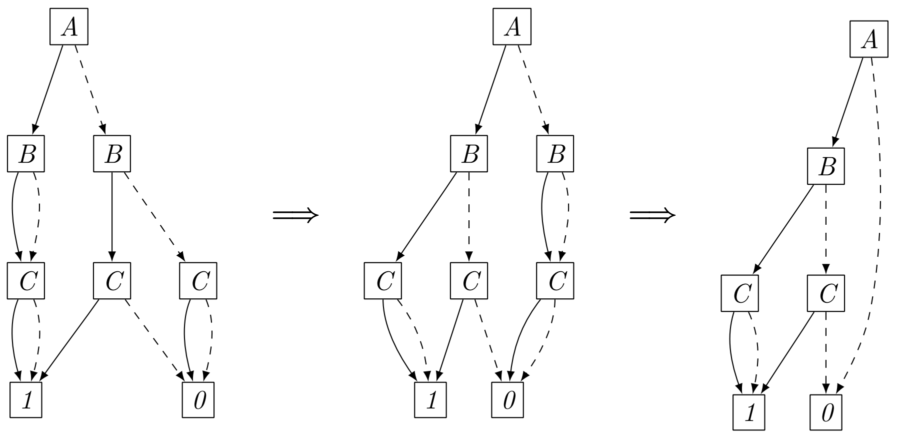




$\boldsymbol{SDD}(\varphi)$ $\boldsymbol{SDD}(\varphi^{f_1 s_0})$ $\boldsymbol{ZDD}(\varphi^{f_1 s_0})$



$\boldsymbol{SDD}(\varphi)$ $\boldsymbol{SDD}(\varphi^{f_0 s_1})$ $\boldsymbol{ZDD}(\varphi^{f_0 s_1})$

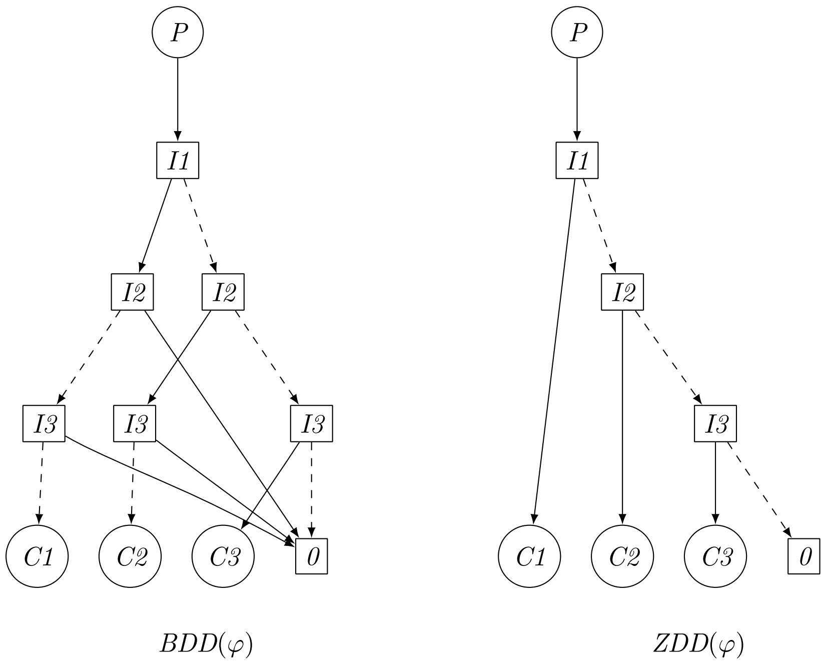


$\boldsymbol{SDD}(\varphi)$ $\boldsymbol{SDD}(\varphi^{f_0 s_0})$ $\boldsymbol{ZDD}(\varphi^{f_0 s_0})$

2.2.7 Sparseness and XOR domains

We have identified patterns in the task of symbolic model checking where BDDs are more compact than ZDDs, such as formulas existing out of atomic propositions for a large vocabulary (or context), and formulas with many negation operators (applied to more than just a propositional variable). Are there also characteristics of logic systems for which ZDDs are more compact? Yes, lets first consider the following pattern:

Example 2.15. *Consider a $\mathcal{L}_B$ formula, φ, containing an XOR-domain for the (grouped and ordered) propositions (I_1, I_2, I_3), each with the same parent and children sub-graphs that are different from each other.*



In this sub-graph one and only one of the set of "I" variables has to be true, otherwise 0 is returned. This is equal to an XOR-relation if we disregard the rest of the graph. Whether XOR patterns appear anywhere in the graph is difficult to predict before construction. It becomes easier if the XOR-relation holds on all branches of the graph for a set of variables. If we were to isolate these variables in the logic formula, the following pattern would emerge:

Definition 2.36. *We define the* ***exclusive or*** *(XOR) relation as $\varphi \oplus \psi := (\varphi \land \neg\psi) \lor (\neg\varphi \land \psi)$. We define its dual (XNOR) as $\varphi \odot \psi := (\varphi \land \psi) \lor (\neg\varphi \land \neg\psi)$.*

This is commonly taken to generalise to an n-ary connective that is true iff an odd number of the φ_i formulas is true: $\oplus\{\varphi_1, ..., \varphi_n\} := (...(\varphi_1 \oplus \varphi_2)... \oplus \varphi_{n-1}) \oplus \varphi_n$

However, we can alternatively generalise to a n-ary connective that is true iff a single one of the φ_i formulas is true:
$\oplus!_1\{\varphi_1, ..., \varphi_n\} := (\varphi_1 \land \neg\varphi_2 \land ... \land \neg\varphi_{n-1} \land \neg\varphi_n) \lor (\neg\varphi_1 \land \varphi_2 \land ... \land \neg\varphi_{n-1} \land \neg\varphi_n) \lor ... \lor (\neg\varphi_1 \land \neg\varphi_2 \land ... \land \varphi_{n-1} \land \neg\varphi_n) \lor (\neg\varphi_1 \land \neg\varphi_2 \land ... \land \neg\varphi_{n-1} \land \varphi_n)$

This relation among the propositions $\in \mathcal{D}$ becomes visible in any formula φ when quantifying over all propositions $\notin \mathcal{D}$:

$\forall\{x \in V \mid \notin \mathcal{D}\}\ (\varphi\ \rightarrow\ \oplus!\{p \mid p \in \mathcal{D}\})$

If a set of propositional variables contains only this relation among themselves in the graph, such *that all variables are positioned next to each other in the order we refer to them as a* ***XOR-domain****.*

To indicate we are talking about the sub-graph of the levels i to j which are respectively the lowest and highest positions of the propositional variables in $\mathcal{D} \subseteq \mathcal{V}$ following the same ordering of $\mathcal{V}$, we write $\boldsymbol{BDD}^{\mathcal{D}}(\varphi)$ and $\boldsymbol{ZDD}^{\mathcal{D}}(\varphi)$. If this section contains a XOR pattern we write $\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)$ and $\boldsymbol{ZDD}^{\mathcal{D}\oplus!}(\varphi)$

For any number of propositional variables in the domain we have that the $\boldsymbol{size}(\boldsymbol{ZDD}^{\mathcal{D}\oplus!}(\varphi)) = \frac{n}{2} \cdot (\boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)) + 1)$, indicating a close to maximum difference according to the bounds given by Knuth. This indicates that for this pattern ZDDs have close to the maximum possible advantage over BDDs.

In logic formulae an XOR-domain can be written down in many forms (since logic formula are not cannonical). Ideally we want to be able to recognise the existence of such domains in the problem formulation before construction. For this it helps to see the pattern as a dictionary (in the example, the variables of the domain are named with I to represent their function of indexing). If we use our belief structures to represent systems where dictionary-like relations hold, ZDDs can have an advantage - depending on how much of the problem falls within the domain.

DEL models with XOR-domains present in their belief structures usually have limits build in the problem statement, e.g. “only one move a turn is possible”, or “one letter per position in a word”, such that the order matters and there are few answers correct. The function will then have to return false (impossible) for multiple inputs in a single time step.

Sparseness of valid states

The literature on comparing efficiency between ZDDs and BDDs often states that ZDDs are more efficient for sparse combination sets. For example as written by Iwashita and Minato in [IM13]; “ZDDs are especially suitable for representing families of sparse item sets. If the average appearance rate of each item is 1%, ZDDs are possibly up to 100 times more compact than BDDs”. This property has been shown by application of ZDDs for the 8 queens problem [Min94], fault simulation [Min94], sets of strings (lexicons) [Knu11] and simple paths [Knu11], which all outperform their BDD counterparts.

To be more precise; the sparsity is the proportion of minterms (SDD paths leading to 1) compared to the total number of paths. In the state law of our belief structures this would be the number of valid states compared to all 2^n states.

Yet just using sparseness as cause to why ZDDs can outperform BDDs is not doing it justice. In order to answer how the sparseness is related to the number of dont-care nodes present in the SDD representation, consider that each BDD eliminable node has isomorphic children and this node is “copied” for each different maximal set of isomorphic paths (otherwise the merging rule would apply).

To maximise BDD eliminable nodes we need as many different sets of isomorphic nodes possible, and since “different” and “ismorphic” are in contrast to each other, BDDs have most eliminable nodes for random boolean functions when the sparseness is 0.5.

In contrary, ZDDs have the opposite relation with sparseness; as less merging of isomorphic nodes happen proportionally more nodes can be eliminated. A sparse 1-set (strong sparsity) indicates that many 0-set nodes can be eliminated and a sparse 0-set (weak sparsity) indicates that many 1-set nodes can be eliminated.

We have showed this property in its more extreme case with the XOR pattern: The number of valid states (paths) are maximised while there are no isomorphic nodes.

To calculate the number of BDD eliminable nodes directly is as much work as constructing the BDD and ZDD directly, yet sparseness (as heuristic for which representation is likely to be more compact) is often easy to calculate: we just need to look at the proportion of valid states compared to all possible states. This sparseness property gives an indication for when ZDDs might be more efficient. How reliable this indications is will be tested in our experiments.

Generalisation of the XOR-domain

Before we only considered the basic XOR-domain where each child node is unique, there only exists 1 parent path and only 1 index can be selected. We can see whether the advantages of ZDDs over BDDs still hold when we relax each of these three requirements, and consider the different variants of ZDDs:

1. When we generalise to any given number, m, of indexes that have to be selected (over n variables) for an XOR-domain present in φ, where all unique index combinations correspond to unique children we get:

$$\boldsymbol{size}(\boldsymbol{ZDD}^{\mathcal{D}\oplus!}(\varphi)) := \binom{n+1}{m} - 1$$
$$\boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)) := \binom{n+2}{m+1} - 1 - \binom{n}{m}$$

The main idea for deriving this closed form solution is that the length of each path in the XOR-domain, as long it has not ended yet, is invariant for the order of the selected indexes. All valid paths (in both ZDD and BDD representation) have exactly m positive literals (selected indexes) and have to terminate before the $n+1$'th variable. This means that the sizes of the child sub-graphs are the same for both paths, (with slight misuse of notation:) $[..., i, \overline{i+1}]$ and $[..., \overline{i}, i+1]$, which in turn allows us to model this situation as paths through Pascal's triangle. Each valid path shares its nodes with each other valid path whenever possible, which corresponds to larger paths in Pascal's triangle sharing nodes with the smaller paths that have taken the same steps.

There are $n-m$ steps possible to the right, where no index is selected and thus a negative literal is added to the path. There are m steps possible to the left, where an index is selected and a positive literal is added to the path. For any coordinates for $y = (n-m)$ ELSE steps and $x = m$ THEN steps we get the number of possible paths to those coordinates by using the binomial formula: $\binom{x+y}{x}$.

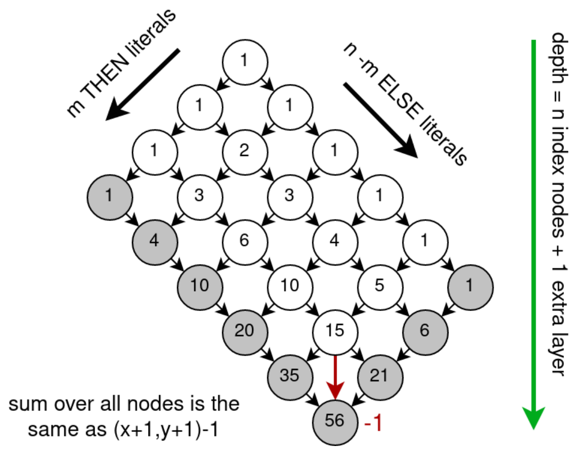
To get the number of nodes in the XOR$_m$ domain, we do not include in our count the resulting child nodes (such that the dimensions become $m-1$ and $n-m-1$), then take the sum of all numbers in a Pascal's triangle within those dimensions. This is because each unique set of coordinates, (x, y), that are possible adds a single node (in the decision diagrams) per path leading there to the previous possible path sizes, $(x-1, y)$ and $(x, y-1)$. Instead of summing over all these values for a given x and y, we can calculate the binomial of the coordinates $(x+1, y+1) - 1$. This gives us the equation for the ZDD XOR$_m$-domain size:

$$\sum_{x=0}^{(m-1)} \sum_{y=0}^{(n-m-1)} \binom{y+x}{x} \quad \Rightarrow \quad \binom{n-(m-1)+1+(m-1)+1}{m-1+1} - 1 \quad \Rightarrow \quad \binom{n+1}{m} - 1$$

For BDDs we add an extra layer to the m dimension, because after m indexes have been selected in a BDD path, it continues with an ELSE chain till $y = n-m-1$ is reached. Unlike with ZDDs, the BDD elimination rule does not eliminate these nodes. We do not consider the last node from that layer as the maximum depth of n is reached, thus we subtract the last node's value from the total, giving us the BDD XOR$_m$-domain size:

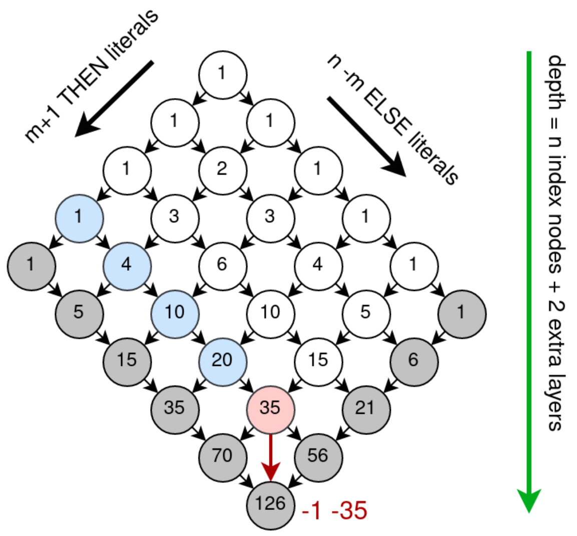
$$\left(\sum_{x=0}^{(m)} \sum_{y=0}^{(n-m)} \binom{x+y}{x}\right) - \binom{(n-m)+m}{m} \quad \Rightarrow \quad \binom{n+2}{m+1} - 1 - \binom{n}{m}$$

Example 2.16. *Consider a ZDD for an XOR-domain with 7 (n) index variables from which 3 (m) have to be selected. We can represent the process of deriving its size by the following Pascal's triangle:*


m THEN literals
n -m ELSE literals
depth = n index nodes + 1 extra layer
sum over all nodes is the same as (x+1,y+1)-1
-1

Plugging in the numbers for our formula indeed gives us: $\binom{7+1}{3} - 1 = 55$

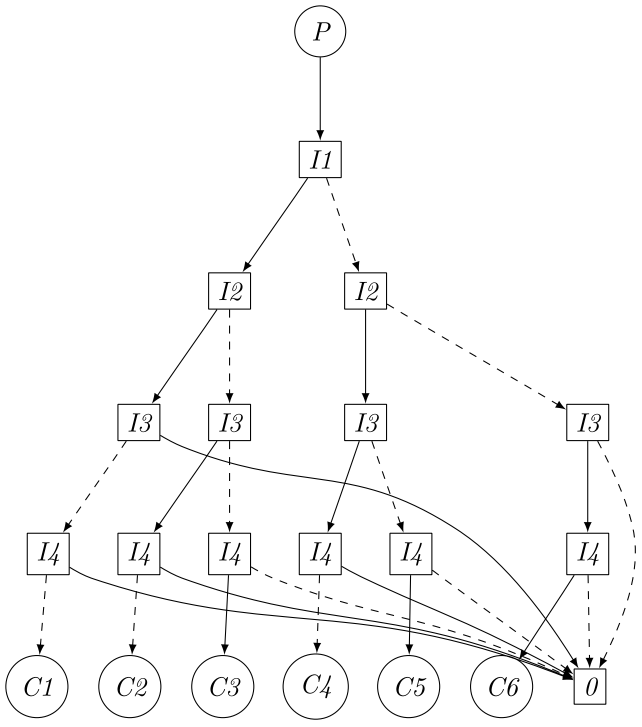
If we want the BDD size we get:



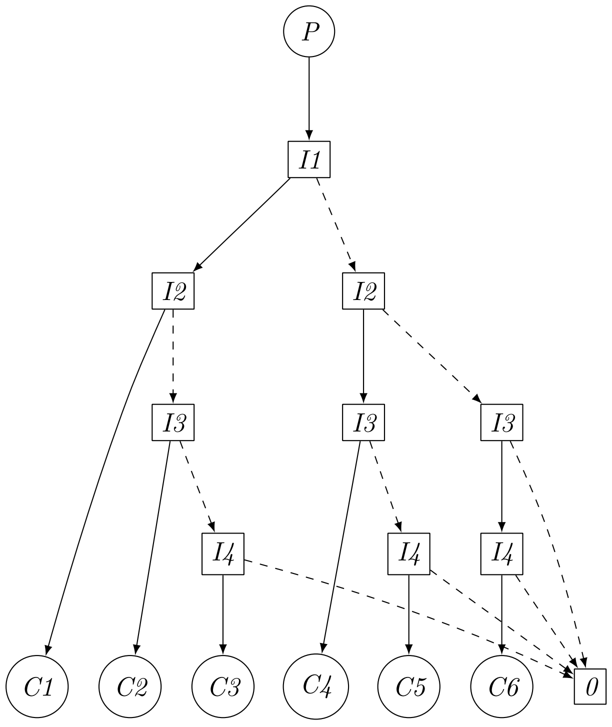
Plugging in the numbers for our formula gives us: $\binom{7+2}{3+1} - 1 - \binom{7}{3} = 90$ *nodes*

The improved efficiency for ZDDs falls off as m approaches n. This fits with the idea of advantage of sparse combination sets for ZDDs; as m gets larger the number of valid paths become larger, past the maximum where isomorphic nodes can be avoided.

Example 2.17. *A smaller example for the actual decision diagrams: Consider a ZDD and BDD example constructed for a formula φ containing an XOR-domain where 2 out of 4 propositional variables $\in \mathcal{D}$ need to be selected:*



$\boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)) = \binom{4+2}{2+1} - 1 - \binom{4}{2} = 13$



$\boldsymbol{size}(\boldsymbol{ZDD}^{\mathcal{D}\oplus!}(\varphi)) = \binom{4+1}{2} - 1 = 9$

2. When in turn we generalise over any combination of resulting children sub-graphs being equal, we see how variable order starts mattering. To calculate the new sizes we should add the following to the equation: If multiple paths end at the same child, find for all paths the lowest shared coordinates with any other path in the set, where they are isomorphic (share the same variable evaluations until n), and then remove the path from the set and remove from the node count the length of this isomorphic part. In the end there will be at least one path left in the set as there will remain at least 1 path into which the others merge.

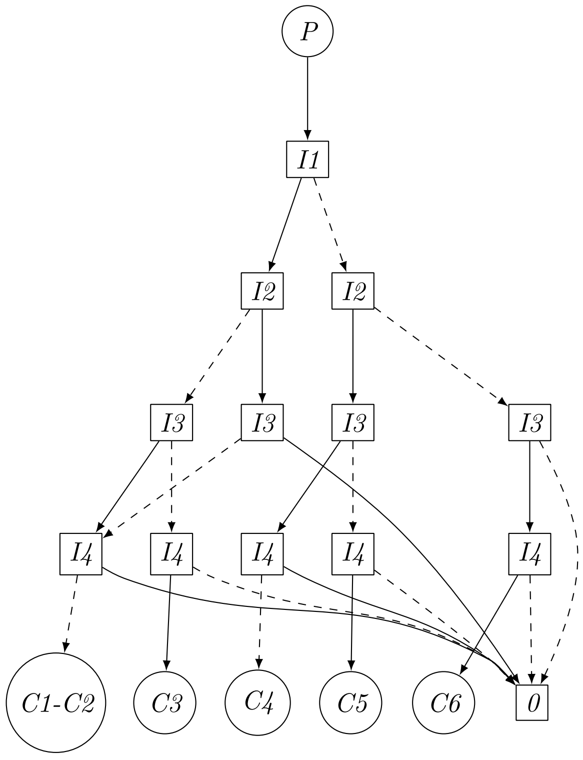
Finding these isomorphic relations between coordinates is not trivial, as the reduction is dependent on variable order in this scenario. Even though for this we cannot give a closed form solution we can get upper and lower bounds, since the size for XOR$_m$-domain sub-graph lies in between the cases of 'all unique children' and 'all the same child'. When all resulting sub-graphs of the indexes are equal (but not $\perp$), the BDD size is $n \cdot m + (n - m)$, and the ZDD size is $n \cdot m$.

This gives the bounds on the size of the subgraph of any decision diagram representing φ containing an XOR$_m$ domain, such that $\forall x \notin \mathcal{D}\ (\varphi \to \oplus!_m\{p \mid p \in \mathcal{D}\})$:

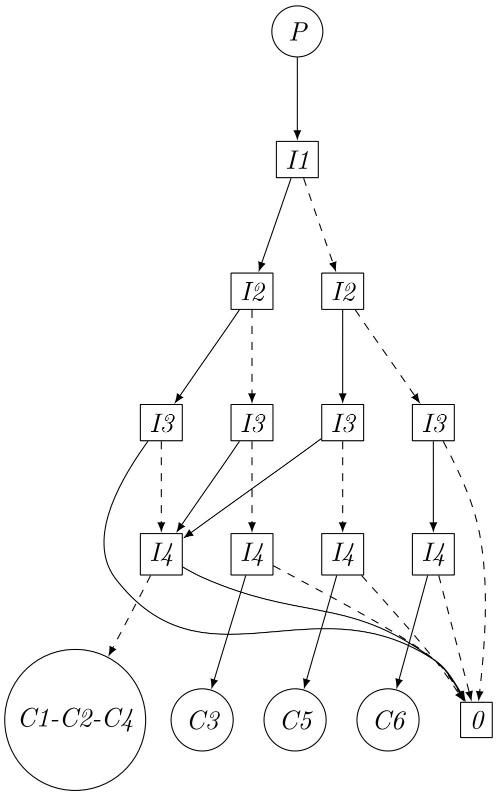
$$\begin{array}{ccccc} n \cdot m + (n-m) & \leq & \boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)) & \leq & \binom{n+2}{m+1} - 1 - \binom{n}{m} \\ n \cdot m & \leq & \boldsymbol{size}(\boldsymbol{ZDD}^{\mathcal{D}\oplus!}(\varphi)) & \leq & \binom{n+1}{m} - 1 \end{array}$$

This is nice because we can make predictions for our experiments on XOR domains and avoid diving into optimising variable order, which leads to a large rabbit hole we would rather avoid in this thesis.

Example 2.18. *Consider the above BDD with choose 2 out of 4; have 3 children be the same and 2 children spread out:*



$\boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)) = \binom{4+2}{2+1} - 1 - \binom{4}{2} - 1 = 12$



$\boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)) = \binom{4+2}{2+1} - 1 - \binom{4}{2} - 2 = 11$

3. The size equation when relaxing the requirement of only having one or no parent path leading to the XOR domain can still be described in closed form when again we consider the simple cases where either:

If the set of all resulting children sub-graphs is completely different for each parent path we get a copy of the sub-graph for each parent path.
If the set of all index combination - children subgraph pairs are equal for all parent paths, due to the merge rule, all parent paths lead to the same XOR-domain sub-graph and no size increase happens.

This means we can still determine the lower and upper bounds for the size of the decision diagrams when having to relax this constraint, where we use c for the number of copies (unique incoming parent paths) and where $\{p \mid p \in \mathcal{D}\}$ are any n consecutive propositional variables in the given ordered context:

$$\begin{array}{ccccc} n \cdot (n-1) & \leq & \boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)) & \leq & c \cdot (\binom{n+2}{m+1} - 1 - \binom{n}{m}) \\ n & \leq & \boldsymbol{size}(\boldsymbol{ZDD}^{\mathcal{D}\oplus!}(\varphi)) & \leq & c \cdot (\binom{n+1}{m} - 1) \end{array}$$

4. Naturally if the f_1s_1 variant of ZDDs is more efficient for XOR-domains (than BDDs) we can invert the pattern among the inference symmetries such that we get 3 more patterns for which the corresponding other variants are equally more efficient:

Definition 2.37. *When using complement edges equivalences from Definition 2.35, we state that the ZDD efficient pattern comes in 4 variants:*

$f_1s_1 : \forall x \notin \mathcal{D}\, (\varphi \to \oplus!_m\{p \mid p \in \mathcal{D}\})$ *if more than m literals are true, then* $\bot$.
$f_1s_0 : \forall x \notin \mathcal{D}\, (\varphi \to \oplus!_m\{\neg p \mid p \in \mathcal{D}\})$ *if more than m literals are false, then* $\bot$.
$f_0s_1 : \forall x \notin \mathcal{D}\, (\varphi \to \odot!_m\{p \mid p \in \mathcal{D}\})$ *if more than m literals are true, then* $\top$.
$f_0s_0 : \forall x \notin \mathcal{D}\, (\varphi \to \odot!_m\{\neg p \mid p \in \mathcal{D}\})$ *if more than m literals are false, then* $\top$.

All for which the corresponding pattern provides a size advantage to ZDDs equal to:

$$(n-1) \quad \leq \quad \boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus!}(\varphi)) - \boldsymbol{size}(\boldsymbol{ZDD}^{\mathcal{D}\oplus!}(\varphi)) \quad \leq \quad c \cdot (\binom{n+2}{m+1} - \binom{n}{m} - \binom{n+1}{m})$$

for a domain $\mathcal{D} \subseteq \mathcal{V}$ continuous in the given order of $\mathcal{V}$ with size n, an number of "indexes" to be "selected" m, and some undetermined and variable order dependent integer c.

Recall that the size of BDDs does not change over these transformations, thus in any of these 4 scenarios there is an appropriate ZDD variant with a smaller XOR sub-graph. We finalise the analysis with two examples where knowledge about XOR-domains in the task indicates advantages for ZDDs over BDDs.

2.2.8 Hypothesised advantageous model checking with ZDDs

Dining cryptographers using XOR-domains

The dining cryptographers logic problem exemplifies an (easily-recognisable) XOR-domain:

"Three cryptographers gather around a table for dinner. The waiter informs them that the meal has been paid for by someone, who could be one of the cryptographers or the National Security Agency (NSA). The cryptographers respect each other's right to make an anonymous payment, but want to find out whether the NSA paid."[Cha88]

The problem states that only 1 of the agents or a the NSA has paid their bill. This rule holds throughout updates (e.g. no updates in this logic puzzle are possible that introduce a new scenario where more than 1 has paid). As soon as more than 1 would have paid this would not be possible thus the decision diagram representing the state-law would lead to the 0 leaf. This gives an XOR-domain with $m = 1$ in the state law with the 4 propositional variables (3 diners + 1 NSA) indicating who paid included in $\mathcal{D}$.

If no further restrictions are set on the possible states, we have that all chosen indexes lead to the same child, namely the 1 leaf. In this case, over this domain in the graph ZDDs will be more efficient by exactly $n - 1$ nodes.

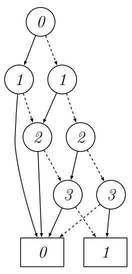
If further restrictions are set on the possible states dependent on propositional variables *outside* the domain, yet which in themselves are dependent on variables *inside* the domain (e.g. information about publicly observable steps in the protocol dependent on who potentially paid), we get that the children may differ for each index while keeping $m = 1$, thus increasing the ZDD efficiency for the sub-graph in the domain.

It is also possible that the children sub-graphs themselves may have more advantage from using BDDs thus even though efficiency of ZDDs for the domain sub-graph is guaranteed (by at least $n - 1$), it might not outperform BDDs when it comes to the size of the complete graph.

Furthermore the protocol for solving the dining cryptographers consists of updating the model such that all agents learn whether the generalised XOR of the shared bit is 1 or 0. Taking p_1 to be a payer and p_4 and p_5 his shared bits, then any agent knowing whether the XOR is 1 or 0 is modelled by $\oplus\{p_1, p_4, p_5\} \vee \odot\{p_1, p_4, p_5\}$, the "$\oplus\{p_1, p_4, p_5\}$" part indicates an XOR domain-like pattern, and thus potential advantage for ZDDs, however the "$\vee \odot \{p_1, p_4, p_5\}$" part changes the pattern and maybe also the expected advantages. This will be tested in our experiments. The complete symbolic representation including these updates and the calculated advantages are given in the next chapter before running our experiments (Subsection 3.1.2).

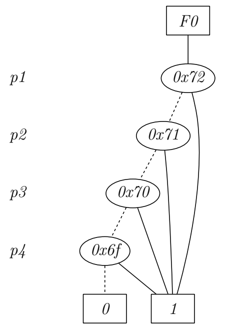
Example 2.19. *The initial knowledge structures containing the BDD state law and ZDD $f_1 s_1$ state law, where the " :: " indicates which type of boolean representation is used.*

$$\mathcal{F}_0 :: BDD = \Bigg(\{p, p_1, p_2, p_3, p_4, p_5, p_6\},$$



$$, \begin{matrix} \{p_1, p_4, p_5\} \\ \{p_2, p_4, p_6\} \\ \{p_3, p_5, p_6\} \end{matrix} \Bigg)$$

$$\mathcal{F}_0 :: f_1 s_1 = \Bigg(\{p, p_1, p_2, p_3, p_4, p_5, p_6\},$$



$$, \begin{matrix} \{p_1, p_4, p_5\} \\ \{p_2, p_4, p_6\} \\ \{p_3, p_5, p_6\} \end{matrix} \Bigg)$$

The CUDD library visualises its decision diagrams differently than the CacBDD library: it starts its variables at 1, has a top node (F0) in case we want to represent more boolean functions in the same decision diagram and denotes the variable names on the side such that the nodes carry their unique address in the CUDD manager.

Indexing agent beliefs for Sally-Anne

Currently we store a boolean function for each agent representing their belief in belief structures. We can reduce this storage by merging each Ω_i, where $i \in \Delta$, such that a single Ω contains all paths indexed by an XOR-domain of new variables representing all agents. As result the children sub-graphs from the index variables can share paths. For example, two agents having the same beliefs would be represented by two different index nodes leading to a single sub-graph (instead of storing the sub-graph twice). For this pattern ZDDs would be more efficient because it is essentially an XOR-domain (containing all agents as propositional variables in its domain), where at every operation only one agent's beliefs are selected ($m = 1$). This would mean we would have to adjust our methods for operating with agent beliefs.

Definition 2.38. *We change the local boolean translation of the Belief operator and Common belief operator to:*

5. For belief, let $\|\Box_i \psi\|_{\mathcal{F}} := \forall V'(\theta' \to ((p_i \wedge \Omega) \to (\|\psi\|_{\mathcal{F}})'))$.
6. For common belief, let $\|C_\Delta \psi\|_{\mathcal{F}} := \boldsymbol{gfp}\Lambda$, *where* Λ *is the following operator on boolean formulas modulo equivalence and* $\boldsymbol{gfp}\Lambda$ *denotes a representative of its greatest fixed point:*
$\Lambda(\alpha) := \forall V' \Big(\theta' \to \Big(\bigvee_{i \in \Delta} (p_i \wedge \Omega) \to (\|\psi\|_{\mathcal{F}} \wedge \alpha)' \Big) \Big)$

where p_i *is a fresh proposition added to* V *corresponding to an given agent.*

Furthermore, we can introduce a limitation on Ω *such that we cannot select 2 agent indexes at the same time; it is up to us to define how to combine the individual agents' beliefs together:*

$\Omega \to \oplus_1 ! \{p_i \mid i \in \Delta\}$

If we do not introduce the XOR rule between agent propositions in Ω, we get agent belief intersection "for free", however the sub-graph containing the agent propositions will become more complex. Intersection of agent beliefs is particularly useful for models keeping track of distributed belief. For example: for two agents represented by propositions p_i and p_j with belief accessibility relations (in the double vocabulary, $V \cup V'$) represented by formulas b_i and b_j, we set Ω (the boolean function representing all (indexed) agent beliefs) to be $\Omega \to ((p_i \to b_i) \wedge (p_j \to b_j))$. From this it follows that

$\Omega \rightarrow ((p_i \land p_j) \rightarrow (b_i \land b_j))$.

In the same spirit of improvement one could ask why we do not store Ω in θ directly, but as Ω is defined in the double vocabulary, $V \cup V'$, it will will enlarge the size of θ. Many of our methods for manipulating the belief and knowledge structures act only on the θ, thus we believe these will be considerately slower if θ is also used for storing agent beliefs where many nodes cannot be eliminated (e.g. for $\boldsymbol{ZDD}(\top)$ and for $\boldsymbol{BDD}$("all negative literals").

We will investigate both the improvement of indexing agent belief and adding the XOR rule (while adapting our traversal algorithms) on the following scenario, requiring factual change and belief when modelled in SMCDEL:

"Sally has a basket, Anne has a box. Sally also has a marble and puts it in her basket. Then Sally goes out for a walk. Anne moves the marble from the basket into the box. Now Sally comes back and wants to get her marble. Where will she look for it?" [WP83]

The Sally-Anne false belief task is a famous example from Psychology used to illustrate and test for a theory of mind. An observer of the story needs to be understand that Sally keeps the false belief that the marble stays in her basket, even though the observer knows otherwise.

Modelling the story shows the use for transformers and belief; agents need to be able to have false beliefs and facts need to be able to change. A common way for extending the model is by including higher-order beliefs; e.g. a study showed that 5 year old kids could be trained on second-order theory of mind by adapting the Sally-Anne story [Ars+20]. They upscale the problem by including more agents (which can or cannot see the changes happening), which in turn allows for higher-order beliefs (as they can hold beliefs about beliefs of other agents). Let us take a look at one of their stories testing second-order beliefs for finding a general method for up-scaling this problem:

Example 2.20. *Adapted "chocolate bar" story from [Ars+20]*

a) *Kevin and Marieke are in the living room.*
b) *Their mother gives a chocolate bar to Kevin.*
c) *Kevin eats some of his chocolate and puts the remainder into the drawer.*
d) *After that, Kevin goes to help his mother in the kitchen. Marieke is alone in the room. She takes the chocolate from the drawer and puts it into the toy box. While she is putting the chocolate into the toy box, Kevin is passing by the window. He sees how Marieke takes the chocolate out of the drawer and puts it into the toy box. Marieke does not see Kevin.*
e) *After that, both go to the kitchen. While Kevin and Marieke are in the kitchen, their mother goes to the living room to watch TV. While she is searching for the remote control, she sees the chocolate in the toy box. She takes the chocolate from the toy box and puts it into the TV stand. After watching TV, she goes to her room.*
f) *Now, Kevin and Marieke go back to the living room. Kevin wants to eat some of his chocolate. At this point the first-order false belief question "Where will Kevin look for the chocolate?" (in the toy box) and the second-order false belief question: "Where does Marieke think that Kevin will look for the chocolate?" (in the drawer) can be asked.*

Having second-order false belief reasoning suggests that children understand that beliefs can be about other beliefs and not just events in the world, meaning that they start to reason about beliefs recursively, e.g.: "Marieke (falsely) believes that Kevin believes that the chocolate is in the drawer", not just "Kevin (falsely) believes that the chocolate is in the drawer".

For a general method of up-scaling the Sally-Anne task we would like to have the level of higher-order beliefs as a parameter. Let's introduce the following requirement: for each agent $i \in \Delta$; each agent contains a i'th order theory of mind about some set of "real-world" propositions. For example when having a model with a third child, Niels, we would like to have Niels falsely believe that Marieke falsely believes that the chocolate is in toy box, and also falsely believes Kevin believes the chocolate is in the drawer. This still allows for many types of models with many parameters, thus to limit and linearize the up-scaling we impose the following structure:

We initialise our model to the moment where the chocolate has been placed in the first box by the first child. All n (our parameter representing $|\Delta|$) children observe this as common knowledge. Then the first child leaves the room, but before going away completely she or he looks back through the window. The child then secretly sees the next child of the group moving the chocolate to a new hiding spot. The next child then also leaves the room, looks back and sees the following child of the group repeating their behaviour. This continues until the last child moves the chocolate and one before last child secretly observes this. If we want to make sure each belief is false we introduce a new propositional variable for each new "box" the chocolate is moved to, which all previous agents do not consider.

Following this we update the model, in order, for each agent $i \in (1, .., n)$, with the transformer such that:

- agent $(i+1)$ moves the chocolate from spot hiding i to hiding spot $(i+1)$,
- all children $> (i+1)$ openly observe this event and believe all children $<= i$ do not observe this,
- i secretly observes this event happening,
- all children $< i$ do not observe anything.

With these rules we can model higher-order Sally-Anne in SMCDEL, with a parameter for how many children participate in the scenario and at the end of the updates all children will have false beliefs about all children's beliefs. For this model the different decision diagrams sizes of the laws can then be compared and the influence of belief indexing can be investigated. The precise initial belief structure and transformers are also given in the next chapter, subsection 3.1.2.

Chapter 3

Experiments

In this chapter Section 3.1 explains the research question and experiments, and Section 3.2 presents and discusses the results.

3.1 Experimental setup

3.1.1 Hypothesis and research questions

Following the motivation given in the introduction (Subsection 1.2), our research goal is to compare ZDDs with BDDs for DEL Symbolic Model Checking using SMCDEL [Gat18b]. Our research question and hypothesis are thus as follows:

Can ZDDs be more compact than BDDs for common DEL Model Checking tasks, and if so, can we predict from the task's symbolic representation, which ZDD variant is more compact?

There exist common DEL Symbolic Model Checking tasks for which ZDDs will be more compact than BDDs and that the most compact variant can be predicted by how proportionally large an XOR-domain is present in the symbolic problem statement and by how sparse the true set of states (the 1-set) for a L_B formula is in comparison with the set of all possible states.

Based on our analysis of the previous chapter, we design our experiments to answer four sub-questions, in order to illustrate the relations mentioned above on a set of classical logic problems where parameters are varied to up-size various aspects of the problems:

1. Can ZDDs reduce the average and worst-case size of the boolean representation of the state law for common DEL problems?

2. How much reduction in size does the XOR-pattern in classic DEL problems provide for ZDDs, compared to BDDs?

3. How much reduction in size does the sparseness of states in classic DEL problems provide for ZDDs, compared to BDDs?

4. How much reduction does "indexing agent beliefs" (as discussed in the previous subsection, 2.2.8) provide for ZDDs, compared to BDDs?

We test the sizes of the different decision diagrams for boolean representations used during the model checking tasks for the Muddy Children, Sum and Product, Dining cryptographers and Sally-Anne logic puzzles, for which we identify in Subsection 3.1.3 the sparseness and XOR domains present in the symbolic representation. From this we form predictions about size differences between the decision diagrams for the state laws used in the belief structures representing each puzzle's model. To generalise our results more we also vary an additional parameter for the muddy children and dining cryptographers.

The sub-questions are tested by using the CUDD and CacBDD library to manipulate the BDDs and convert them to ZDDs, where we compare the sizes of the graphs for each update step in the puzzles. Considering the decision diagram sizes instead of run-time of the program allows us to compare the decision diagrams types irrespective of which libraries they were build with.

The Muddy Children, Sum and Product, and Dining Cryptographers can all be modelled in S5, thus we use (the more efficient) knowledge structures for their tasks. Sub-question 4 can only be modelled with belief structures. With the results of all experiments we will learn when and why ZDDs can outperform BDDs on DEL model checking tasks.

Our additions to SMCDEL (written in Haskell, summary of which can be found in Section 4.1) allows us to represent the belief and knowledge structures for BDDs and all variants of ZDDs while using the libraries CUDD and CacBDD for storing and manipulating the ZDDs and BDDs. The symbolic representation for all logic problems used in testing is given in the upcoming subsection.

3.1.2 Symbolic model checking for classical dynamic epistemic logic puzzles

To keep the thesis somewhat concise we assume the familiarity of the reader with the solutions of the logic puzzles. Before giving their symbolic representation we refer to the more detailed explanation available in Gattinger's work and the original literature on the logic problems.

The first logical approach to changes of knowledge is given in [Pla07]. The logic presented there is nowadays called Public Announcement Logic (PAL) and extends epistemic logic with a modality to describe incoming information.

Most dynamic epistemic logic puzzles are translated using PAL, where the $[!\psi]\varphi$ operator is added to the syntax, which is interpreted as "at the states where ψ holds, everyone now knows that φ holds" (thus $\psi \to C_\Delta\varphi$). In our experiments we treat these public announcements to be external truthful updates to the belief structure: the state law is updated to its conjunction with φ and, in doing so, set evaluations of the paths/states where $\neg\psi$ holds to false.

Muddy Children

Adapted from [Gat18a]; Section 2.3 and Section 4.1, early version first introduced in 1953 by [LM53]

"Imagine n children playing together. The mother of these children has told them that if they get dirty there will be severe consequences. So, of course, each child wants to keep clean, but each would love to see the others get dirty. Now it happens during their play that some of the children, say k of them, get mud on their foreheads. Each can see the mud on others but not on his own forehead. So, of course, no one says a thing. Along comes the father, who says, "At least one of you has mud on your forehead," thus expressing a fact known to each of them before he spoke (if $k > 1$). The father then asks the following question, over and over: "Does any of you know whether you have mud on your own forehead?" Assuming that all the children are perceptive, intelligent, truthful, and that they answer simultaneously, what will happen?" [Fag+95]

Let p_i stand for "child i is muddy". Consider the case of three children $\Delta = \{1, 2, 3\}$ who are all muddy, i.e. the actual state is $\{p1, p2, p3\}$. At the beginning, the children do not have any further information, hence the initial knowledge structure F_0 has the state law $\theta_0 = \top$ and the set of states is the full powerset of the vocabulary, i.e. $\mathcal{P}(\{p_1, p_2, p_3\})$. All children can observe whether the others are muddy but do not see their own face. This is represented with observational variables: Agent 1 observes p_2 and p_3, etc.

$$\mathcal{F}_1 = \{\mathcal{V} = \{p_1, p_2, p_3\},\ \theta_1 = (p_1 \vee p_2 \vee p_3),\ O_1 = \{p_2, p_3\}\ O_2 = \{p_1, p_3\}\ O_3 = \{p_1, p_2\}\}$$

Now the father says "At least one of you is muddy", which we model as a public announcement of $p_1 \vee p_2 \vee p_3$. This limits the set of states by adding this statement to the state law. The father now asks "Do you know if you are muddy?" but none of the children does. As it is common in the literature, we understand this as a public announcement of "Nobody knows their own state.":

$$\bigwedge_{i \in I}(\neg(K_i p_i \vee K_i \neg p_i))$$

In SMCDEL we update the state law of the knowledge structure with:

$$\theta_j = \theta_{j-1} \wedge || \bigwedge_{i \in I}(\neg(K_i p_i \vee K_i \neg p_i))||_{\mathcal{F}_1}$$

After updating the state law by this public announcement as many times as there are muddy children - 1, the resulting knowledge structure has as state law that all muddy-children propositions must be true. For example in the case of three muddy children we get $\theta_3 = (p_1 \wedge p_2 \wedge p_3)$, which marks the end of the story: The only state left is the situation in which all three children are muddy. Moreover, this is common knowledge among them because the only state is also the only state reachable via $\mathcal{E}_I^*$ in Definition 2.12. Alternatively, note that the fixed point mentioned in Definition 2.13 in this case will be the same as θ_3.

In our experiments we consider the two parameter version of muddy children where we vary the number of children and how many of them are muddy. The final state, where every child knows whether they are muddy or not, is after $m - 1$ updates, where m is the number of muddy children.

Dining Cryptographers

Adapted from [Gat18a]; Section 4.3. The full logic problem was first published in 1988 [Cha88], solving it with explicit Kripke model checking is done in [VO07].

We use boolean variables and the XOR function as follows; Let p_0 mean that the NSA paid, p_i for $i \in \{1, 2, 3\}$ that i paid and let p_k for $k \in \{4, 5, 6\}$ represent the shared random bits. The solution protocol uses these random bits (e.g. flipping a coin) each being common knowledge between a duo of agents (diners). In each round of the protocol, a pair of diners announce the XOR of their shared bits and if a diner wants to transmit an untraceable message to the group, they invert their publicly announced bit. The scenario can then be modelled by the knowledge structure:

$$\mathcal{F} = (V = \{p_0, ..., p_k\}, \theta = \oplus!_1\{p_0, p_1, p_2, p_3\}, O_1 = \{p_1, p_4, p_5\}, O_2 = \{p_2, p_4, p_6\}, O_3 = \{p_3, p_5, p_6\})$$

where intuitively the state law θ is saying that someone must have paid but not two of the agents or the NSA at the same time.

Consider the operator for the generalisation of the exclusive disjunction $\oplus$ (without exclamation mark, it returns true if m is odd - from Definition 2.36), and the operator, $[!?\psi]\varphi$, as an abbreviation for announcing whether $([!\psi]\varphi \wedge [\neg!\psi]\varphi)$. The announcements made by the three dining cryptographers can then be formalised as three public announcements: $[?!(\oplus p_1, p_4, p_5)][?!(\oplus p_2, p_4, p_6)][?!(\oplus p_3, p_5, p_6)]$

We can translate the statement "If cryptographer 1 did not pay, then after the announcements are made, 1 either knows that no cryptographers paid, or that someone paid, but in this case 1 does not know who did." to $\mathcal{L}_P$ as

$$\neg p_1 \rightarrow [?!(\oplus p_1, p_4, p_5)][?!(\oplus p_2, p_4, p_6)][?!(\oplus p_3, p_5, p_6)](K_1(\bigwedge_{i=1}^{n=3} \neg p_i) \vee (K_1(\bigvee_{i=2}^{n=3} p_i) \wedge \bigwedge_{i=2}^{n=3}(\neg K_1 p_i)))$$

where p_i says that agent i paid and n is the number of agents participating in the model.

The protocol can be generalised to a group of n participants, each with a shared secret bit in common with each other participant. Because the protocol works for any odd number of payers we include this as an additional parameter. The protocol does not work when the number of payers is even because the payers' messages will cancel each other out.

Sum and Product

Adapted from [Gat18a]; Section 4.6. The original riddle was first introduced in 1969 by H. Freudenthal [EDV09]. Solving it with explicit model checking is achieved in [DRV07].

A translation of the original formulation given in [EDV09] is:

"J says to S and P: I have chosen two integers x and y such that $1 < x < y$ and $x + y \leq 100$. In a moment, I will inform S only of $s = x + y$, and P only of $p = xy$. These announcements remain private. You are required to determine the pair (x, y). He acts as said. The following conversation now takes place:
P says: "I do not know it."
S says: "I knew you didn't."
P says: "I now know it."
S says: "I now also know it."
Determine the pair (x, y)."

To represent numbers, we use binary encodings for $x, y, s := x + y$ and $p := x \cdot y$. Recall that $\lceil ... \rceil$ denotes the smallest natural number not less than the argument. We need $\lceil log_2 N \rceil$ propositions for every variable that should take values up to N. For example, suppose to represent $x \leq 100$ we use $p_1, ..., p_7$. The statement $x = 5$ is then encoded as $p_1 \land p_2 \land p_3 \land p_4 \land \neg p_5 \land p_6 \land \neg p_7$, corresponding to the bit-string 0000101 for 5.

The state law for Sum and Product is a big disjunction over all possible pairs of x and y with the given restrictions. It is here where we ensure that s and p are actually the sum and the product of n and m : $\theta_1 := \bigvee \{x = n \land y = m \land s = n + m \land p = n \cdot m \mid 2 \leq n < m \leq 100, n + m \leq 100\}$

To let the agents S and P know the values of s and p respectively, we define the observational variables $O_S := \{s_1, ..., s_7\}$ and $O_P := \{p_1, ..., p_{12}\}$.

The initial structure then becomes:
$\mathcal{F}_0 = (V = \{x_1, ..., x_7, y_1, ..., y_7, s_1, ..., s_7, p_1, ..., p_{12}\}, \theta, O_S = \{s_1, ..., s_7\}, O_P = \{p_1, ..., p_7\})$
where θ represents the (large) state law as described above.

We can use formulas to say that an agent knows a variable and that the statements of the dialogue can be truthfully announced (see the muddy children and dining cryptographers examples above). The solutions to the puzzle are those states where this conjunction holds.

Higher order Sally-Anne

This version is inspired by the chocolate bar story in [Ars+20]. The first publication of the Sally-Anne story in its modern form was in 1985 by [BLF85]. The original version translated to belief structures is given in [Gat18a], Section 4.7.

We use the vocabulary $V = \{p_0, ..., p_n\}$ for the possible places where the chocolate might be. In the first belief scene, the chocolate has been placed in the basket by the first agent ($i = 0$), and all n other children know where:

$(\mathcal{F}, s)_0 := (V = \{p_0, ..., p_n\},\ \theta = p_1 \land (\neg p_2 \land ... \land \neg p_n),\ \Omega_0, ..., \Omega_n = \top,\ \{p_0\})$

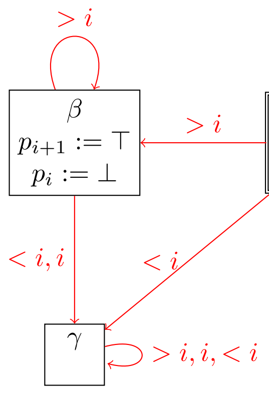
The transformer, $(\mathcal{X}, x)_i$ for each agent i secretly seeing $(i+1)$ moving the chocolate (while the children $> i$ in the room openly observe the factual change) is:

$$
\begin{aligned}
\mathcal{X} = (\quad & V^+ = \{a, b\} \\
, \quad & \theta^+ = \neg(a \land \neg b) \\
, \quad & V_- = \{p_i, p_{i+1}\} \\
, \quad & \theta_-(p_i) := (b \to \bot) \land (\neg b \to p_i) \\
, \quad & \theta_-(p_{i+1}) := (b \to \top) \land (\neg b \to p_{i+1}) \\
, \quad & \Omega^+_i = ((a \land b) \to (a \land b)') \land ((\neg a \land b) \to (\neg a \land \neg b)') \land ((\neg a \land \neg b) \to (\neg a \land \neg b)') \\
, \quad & \Omega^+_{>i} = ((a \land b) \to (\neg a \land b)') \land ((\neg a \land b) \to (\neg a \land b)') \land ((\neg a \land \neg b) \to (\neg a \land \neg b)') \\
, \quad & \Omega^+_{<i} = ((a \land b) \to (\neg a \land \neg b)') \land ((\neg a \land b) \to (\neg a \land \neg b)') \land ((\neg a \land \neg b) \to (\neg a \land \neg b)')) \\
x = \{a, b\} &
\end{aligned}
$$

α	$a \land b$
β	$\neg a \land b$
γ	$\neg a \land \neg b$
not allowed	$a \land \neg b$

The change law tells us that for those belief states where factual change has been observed, labelled with b, the presence of the chocolate bar changes from spot p_i where child i has placed it, and it is moved to the hiding spot $(i+1)$, where child $(i+1)$ has placed it.

The action model for child i secretly observing child $(i+1)$ moving the chocolate:



3.1.3 Expectations

In this section we state our motivation (how these problems relate to answering our research questions) and expectations for the resulting patterns of size differences as we vary the parameters of our models. We are particularly interested in the average size of the state law during the logic puzzle updates. We scale up the puzzles by increasing their first parameter and look for consistent size differences in order to quantify our found results. The sparsity of the model can be calculated by taking the number of minterms of the state law and dividing it by the 2^n states possible (as discussed in Sub-subsection 2.2.7), which is independent of what type of decision diagram is used. The values chosen for the parameters that enlarge the model are dependent on whether the task can finish in a reasonable time (48+ hours); the BDDs without complement edge from the CacBDD library are much slower thus will not have as high parameter values.

Muddy children

Parameter 1 := number of children (3-60).
Parameter 2 := number of dirty children. (3-60)

The muddy children puzzle is the most commonly used logic problem for DEL in the literature. Testing on it will serve as a "control" problem that does not have any recognisable XOR-pattern in (the initial situation and updates of) the problem statement. Valid states in the state law do become more sparse after each update (see Table 1). The average sparsity over all updates in the scenario where all children are muddy is 0.5. Without an extreme average sparsity and no XOR-patterns (or any of its complemented variants, as given in Definition 2.37), we expect BDDs to be more compact on average for this puzzle. Testing this will be the main contribution towards answering research question 1.

We can find the influence of sparsity when looking at the sizes per update: here we expect the f_1 ZDD variants to be more compact in the final state of the puzzle and the f_0 ZDD variant in the initial state. As muddines is indicated by a positive literal in our translation and the latter states of the model contain the states where many of children are muddy (thus few negative literals), we expect the f_1s_0 variant to be more compact than the f_1s_1 variant. Likewise in the initial state of the model we have few states that are impossible, starting from the only state where all three children are clean (the all negative literals state), thus we expect the f_0s_1 variant to be more compact than the f_0s_0 variant.

If we allow to vary the number of dirty children (parameter 2), we get that the model just terminates earlier but keeps the same sizes for every decision diagram. This indicates that the smaller the number of children with mud on their forehead is, the more advantage the f_0s_1 variant has. Plotting the average sizes for an increasing second parameter and plotting the specific update sizes (for given 1st and 2nd parameter values) contributes towards answering sub-question 3.

dirty\children	3	10	30	60	
3	0.875	$1-9.766e^{4}$	$1-9.313e^{10}$	$1-8.6736e^{-19}$	initial
10	∅	$1-9.766e^{4}$	$1-9.313e^{10}$	$1-8.6736e^{-19}$	
30	∅	∅	$1-9.313e^{10}$	$1-8.6736e^{-19}$	
60	∅	∅	∅	$1-8.6736e^{-19}$	
3	0.125	0.945	$1-4e^{7}$	$1-1.5543e^{-15}$	final
10	∅	$9.766e^{-4}$	0.979	$1-1.54258e^{-8}$	
30	∅	∅	$9.313e^{-10}$	0.5513	
60	∅	∅	∅	$8.6736e^{-19}$	

Table 1: *Muddy children state law sparsity for number of total children (row) and number dirty children (column).*

Dining Cryptographers

Parameter 1 := number of dining cryptographers (3-13).
Parameter 2 := number of payers (1-13).

The dining cryptographers has a pure XOR domain in its initial state law and as discussed in Subsubsection subsection 2.2.8 we see a combination of XOR and XNOR its update rule. We thus expect to see ZDDs being more compact in the initial state, and this advantage is possibly preserved throughout the update sequences and thus average size.

Because we cannot conclude specifically who paid with the protocol, there will still be a number of valid states after all updates. This reduces the natural occurring sparsity. Even so, as we upscale the model we see (Table 2) that the sparsity gets more extreme, predicting an eventual f_1 advantage.

Because the shared bits can be 0 or 1 in the possible states, the only asymmetrical restriction on positive or negative literals comes from the initial state law, where we restrict the number of payers possible. We can thus expect fewer eliminable nodes for the f_1s_1 variant and more eliminable nodes for the f_1s_0 variant as we increase this second parameter (from small m in $\oplus!_m\{\text{all\_diners}\}$ to large m). Lastly it is expected that the f_0s_0 and f_0s_1 variants will not have much difference as the XOR domain is opposite of the XNOR domain and will not have any eliminable nodes in this case.

Plotting the state law size data for the initial state (containing the XOR-domain), the average sizes for increasing the number of diners with only 1 payer (XOR pattern mixed with other patterns) and the state law size data when varying the second parameter (the number of payers) will form our answer to sub-question 2.

payers\diners	3	5	7	9	11	
1	0.25	0.0938	0.0313	0.0098	0.0029	initial state
3	0.25	0.3125	0.2188	0.1172	0.0537	
5	∅	0.0938	0.2188	0.2461	0.1934	
7	∅	∅	0.0313	0.1172	0.1934	
9	∅	∅	∅	0.0098	0.0537	
11	∅	∅	∅	∅	0.0029	
1	0.0625	0.0059	$4.883e^{-4}$	$3.815e^{-5}$	$2.861e^{-6}$	final state
3	0.0625	0.0195	$3.418e^{-3}$	$4.578e^{-4}$	$5.245e^{-5}$	
5	∅	$5.859e^{-3}$	$3.418e^{-3}$	$9.613e^{-4}$	$1.888e^{-4}$	
7	∅	∅	$4.883e^{-4}$	$4.578e^{-4}$	$1.888e^{-4}$	
9	∅	∅	∅	$3.815e^{-5}$	$5.245e^{-5}$	
11	∅	∅	∅	∅	$2.861e^{-6}$	

Table 2: *Dining cryptographers state law sparsity for number of diners (row) and payers (column).*

Sum and product

Parameter 1 := upper bound sum of x and y. (50-350)

From Gattingers results we know that explicit model checking is faster for this problem. It is then hypothesised to be because of a well-known problem already mentioned in [Bry86]: BDD representations of products tend to be large. This is because it contains many dependent variables instead of dont-care variables. In Subsection 2.2.4 and Sub-subsection 2.2.7 we have suggested that ZDDs are likely more compact for this scenario. We can upscale the problem by raising the upper bound of the sum of x and y, for which we expect to see an increase in the size difference for ZDDs compared BDDs.

update \ max sum	50	65	100	128	200
1	$1.0729e^{-6}$	$1.1188e^{-7}$	$1.3976e^{-7}$	$1.4439e^{-8}$	$1.7828e^{-8}$
2	$8.5682e^{-8}$	$9.1968e^{-9}$	$8.4401e^{-9}$	$7.4579e^{-10}$	$9.5315e^{-10}$
3	$5.9605e^{-8}$	$5.9372e^{-9}$	$5.0059e^{-9}$	$4.1473e^{-10}$	$4.8203e^{-10}$
4	0	$1.1642e^{-10}$	$5.8208e^{-11}$	$3.6380e^{-12}$	$1.8190e^{-12}$

Table 3: *Sum and product state law sparsity for maximum sum (row) and updates (column).*

There are only 4 states for the Sum and Product puzzle no matter what parameter value is chosen. For any sum lower than 65 there are no answers thus the last update for sum = 50 has 0 states that are possible. From Table 3 we can see how the initial update is the most restrictive, which makes sense: we limit y to be higher than x (limiting the possible paths in the y variable domain by half) and specify that for each valid xy pair only one specific path evaluates to true in the sum and product variable domains (limiting the possible paths to 1 out of 2^n, where n is the size of the union of the sum and product variable domains). 128 is included in the table to show the sparsity right before the range of the numbers in bit representation is extended. The updates afterwards reduce the decision diagram further until only one path is left: the single solution to the puzzle.

Because this is an extremely sparse model we expect the f_1 ZDD variants to outperform BDDs. On top of this, we expect the bit representation for the numbers within the puzzle to cause a difference between the s_0 and s_1 types of ZDDs.

Sally-Anne

Parameter 1 := number of Sally's friends / additional children. (2-18)

As discussed before, Sally-Anne allows us to compare models using belief and factual change. At any state there is only one positive literal (one chocolate bar), thus a proportionally low number of possible states are considered possible (n states compared to 2^n states). This predicts that the f_1s_1 variant will be the most compact of the ZDDs.

Because the Higher Order Sally Anne is not a puzzle (thus does not contain a single final possible state), it is interesting to see whether BDDs or ZDDs are more compact for this model. We expect to see an additional size reduction by indexing the agent beliefs such that only one boolean function for all agents beliefs is sufficient.

3.2 Results and discussion

3.2.1 Interpretation of data

For each puzzle we show plots that give us information about (**1**) ZDDs potentially outperforming BDD representation, (**2**) predictability of advantages based on the XOR-relation and (**3**) predictability of advantages based on the sparsity. The conclusions and their significance we can deduce from this data are discussed in the next subsection. Because we look at 5 types of boolean representations and 4 puzzles (most of which have 2 parameters), we get many possible angles of interpretation, thus additional plots are given in the appendix and we only consider the results that help us form conclusions about the 3 questions. The plots for the worst-case size (instead of average) show very similar plots and similarly support our conclusions, thus these are left out and can also be found in the appendix.

Muddy Children

First we look at the 1-parameter version of the puzzle where all children are muddy. For this case the 3D plots (Figure 1.1, see the appendix for the larger version) show a consistent parabola for all numbers of total children, which we can better compare when taking a slice for each variant and adding them in the same graph. For 30 (muddy) children (see Figure 1.2), the f_0s_1 and f_1s_0 ZDD variants seem to have equal average size as the BDD representation but their curves are shifted to the right and left respectively. The f_1s_1 and f_0s_0 ZDD variants have larger sizes than the BDD on average, again with a similar horizontal shift noticeable in their pattern. The horizontal shift confirms our expectation about the presence of negative and positive literals in the initial and final states of the model, and further shows that the advantage of which literals are inferred gradually transitions as the model is updated.

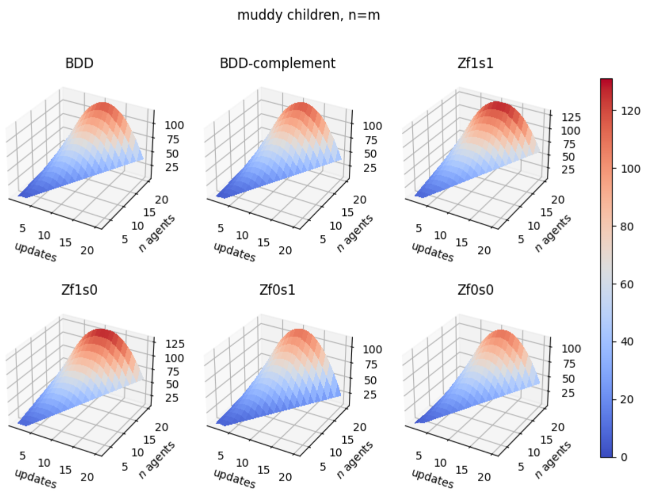


Figure 1.1: Muddy children state law size per update per nr. of children, where nr. of muddy = nr. of children.

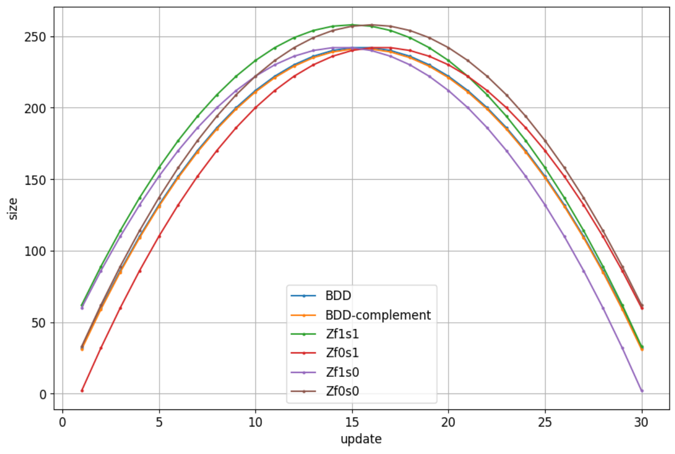


Figure 1.2: Muddy children state law size per update, where nr of muddy and nr. of children = 30.

Figure 1.3 allows us to better compare the differences of the average size, where we see that the f_0s_1 and f_1s_0 ZDD variants are equal in average size with the BDD boolean representation, and this is consistent as the model gets larger. Figure 1.4 shows that the difference in percentage of the (worse) $f_0 s_0$ and $f_1 s_1$ ZDD variants relative to the BDD size diminishes to a seeming asymptote at around +7%.

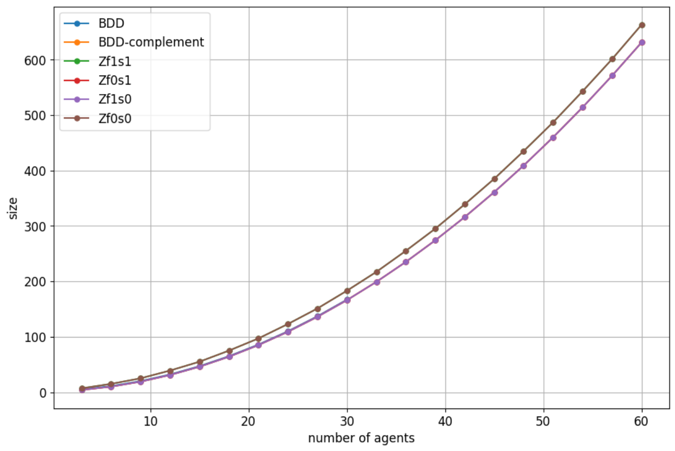


***Figure 1.3**: Average sizes per nr. of children, where nr. of muddy = nr. of children. The BDD-complement, $f_1 s_0$ and $f_0 s_1$ lines are overlapping, and the $f_1 s_1$ and $f_0 s_0$ lines are overlapping.*

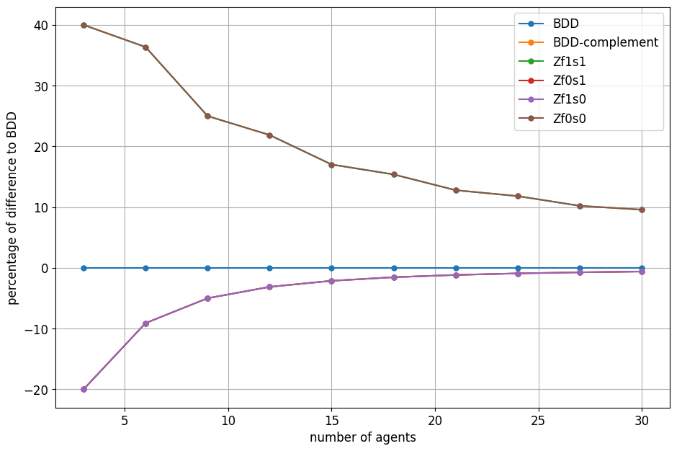


***Figure 1.4**: Difference of average sizes in percentage w.r.t. the BDD representation, per nr. of children, where nr. of muddy = nr. of children. The BDD-complement, $f_1 s_0$ and $f_0 s_1$ lines are overlapping, and the $f_1 s_1$ and $f_0 s_0$ lines are overlapping.*

If we vary the number of muddy children the sizes for each update step stay the same but there are fewer update steps (the plot for this can be found in the appendix), which according to our plot in Figure 1.2 influences the averages to favour the $f_0 s_1$ ZDD variant. We confirm this with the plots in Figure 1.5 and 1.6, where we can see the sizes and the relative differences for an increasing amount of children of which 3 are muddy respectively. If we keep the number of muddy children at 3 we see an relative improvement of around -50% for the $f_0 s_1$ ZDD variant. It is a surprising insight that a ZDD variant outperforms BDDs on the muddy children puzzles where the number of muddy children is fewer than the number of children.

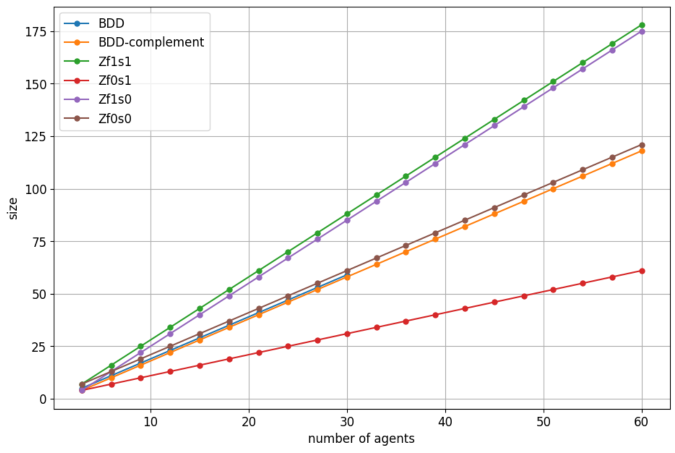


***Figure 1.5**: Average sizes per nr. of children, where nr. of muddy = 3. The data for the BDD representation stops at 30 children.*

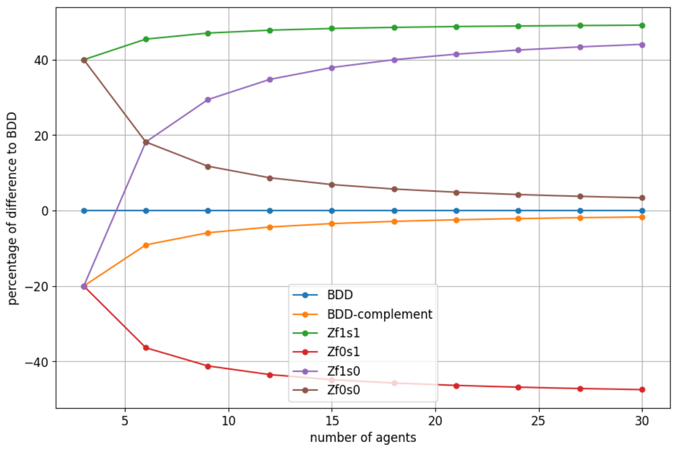


***Figure 1.6**: Difference of average sizes in percentage w.r.t. the BDD representation, per nr. of children, where nr. of muddy = 3.*

Dining Cryptographers

Because the decision diagram sizes for this puzzle get very large very quickly, using the CacBDD library for BDDs without complement edges is not feasible. The 3D plots (Figure 2.1, see the appendix for the larger version) show that around the updates of 13 all decision diagrams get a very steep exponential growth, and this is independent of the number of dining cryptographers, as long as there are enough to get to the later updates. Figure 2.2 shows the sizes per update for 13 diners and 1 payer, where the f_0 ZDD variants are equal in size to the BDD and the f_1 ZDD variants are equal in size, while a bit more compact than the BDD representation.

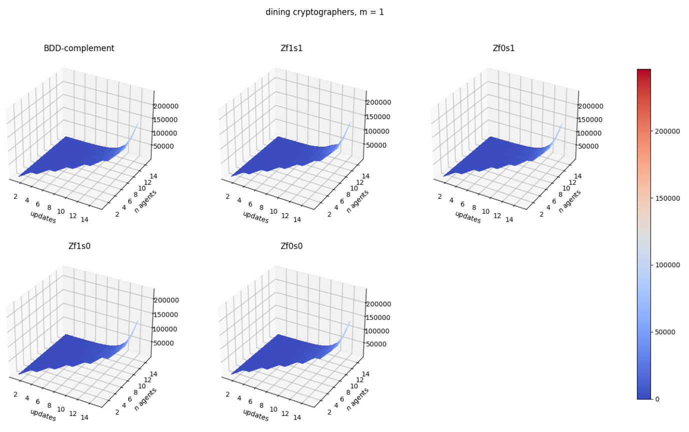


***Figure 2.1**: Dining cryptographers state law sizes per update per nr. of diners, where nr. of payers = 1.*

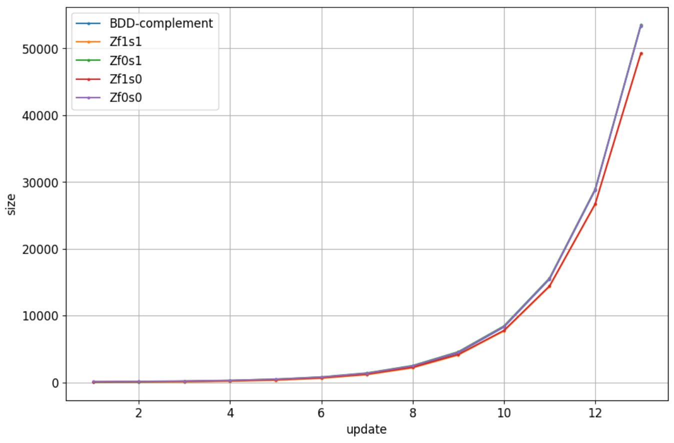


***Figure 2.2**: Dining cryptographers state law sizes per update, where nr. of diners = 13 and nr. of payers = 1. The BDD-complement, f_0s_1 and f_0s_0 lines overlap, the f_1s_1 and f_0s_0 lines overlap.*

To see if these differences are consistent as we vary the number of diners the average sizes are plotted (Figure 2.3), where we see that the f_1 ZDD variants consistently have smaller sizes although all decision diagrams follow a similar exponential curve. At first glance there is no indication that the f_1s_1 variant is outperforming the f_1s_0 variant.

Again we can quantify the differences by plotting the percentage difference relative to the BDD (Figure 2.4), and here we see that for small models the f_1s_1 provides a size improvement over all other decision diagrams - but this quickly converges with the f_1s_0 size difference line and seems to settles at an asymptote around -8%, whereas the differences off the f_0 variants both converge to 0 (equal size as the BDD).

The final state of the model contributes more to the average for a large number of diners, as the state law grows exponentially larger. The proportionally decreasing differences between the s_0 and s_1 ZDD variants show that the XOR-domain (of n variables) proportionally decreases in size compared to the coin flip domain (of $\binom{n}{2}$ variables). The domain with variables representing "*whether* the randomly flipped coin is one or zero" (as we discussed in Sub-subsection 2.2.8) does not discriminate between the s_0 and s_1 variants, and is more compact for the f_1 ZDD variant.

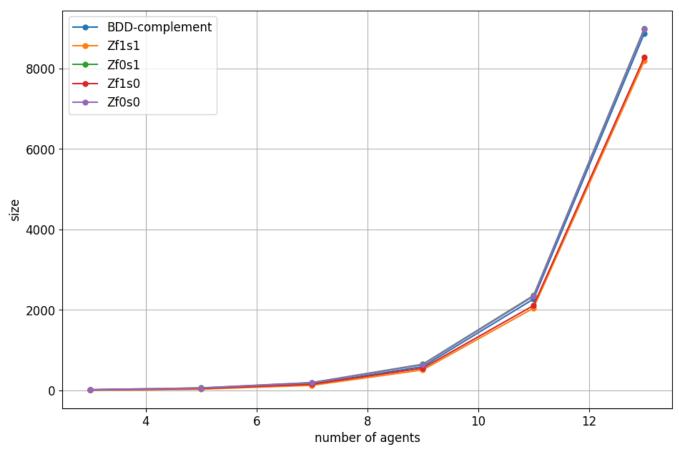


Figure 2.3: *Average sizes per nr. of diners, where nr. of payers = 1. The BDD-complement, f_0s_1 and f_0s_0 lines roughly overlap, the f_1s_1 and f_0s_0 lines roughly overlap.*

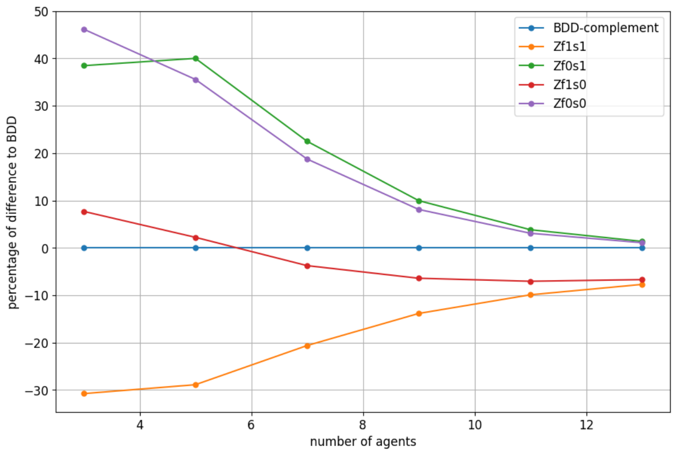


Figure 2.4: *Difference of average sizes in percentage w.r.t. the BDD-complement representation, per nr. of diners, where nr. of payers = 1.*

When we consider only the initial state of the model we can see the influence of the pure XOR-domain; the absolute sizes in Figure 2.5 and the relative size differences in Figure 2.6 show, as expected, the f_1s_1 variant outperforming all other boolean representations, with roughly −45% compared to BDDs.

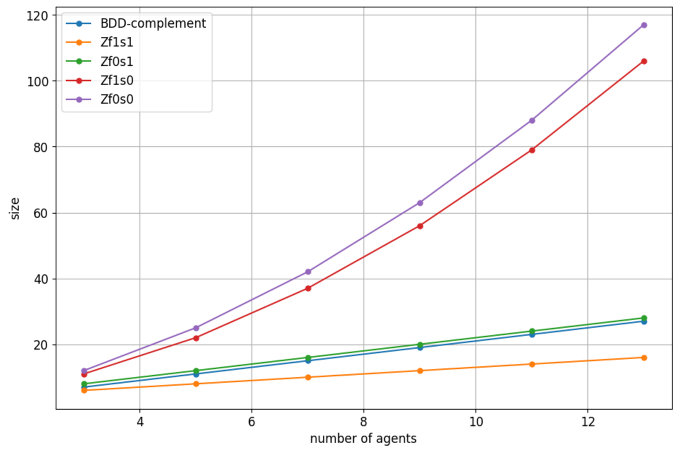


Figure 2.5: *Initial state sizes per nr. of diners, where nr. of payers = 1.*

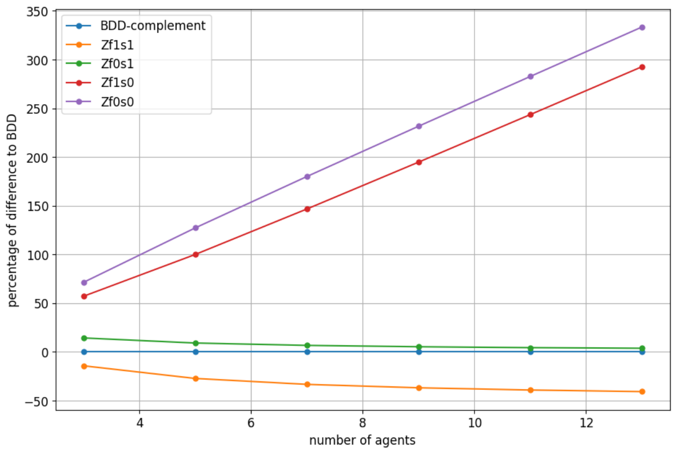


Figure 2.6: *Difference of initial state sizes in percentage w.r.t. the BDD-complement representation, per nr. of diners, where nr. of payers = 1.*

Figure 2.6, where the number of payers is changed and there are 13 diners, shows a sharp normal curve for all boolean representation types. As predicted the s_1 and s_0 variants' lines cross in the middle of the possible number of payers and there is less difference between the f_0s_1 and f_0s_0 variants (due to little eliminable nodes) compared to the difference between f_1s_1 and f_0s_0 variants. If we again quantify the differences in percentages compared to the BDD representation, Figure 2.7 shows that the difference of −8% between the optimal ZDD variant with the BDD for large models is the minimum and can go up to −10% if we choose a number of payers next to 1/3 or 2/3 of the numbers of diners.

The slight dent of the f_0 variants in the middle indicates that, as the number of payers grows, the pattern in the decision diagram where the f_0 variants can eliminate nodes becomes proportionally smaller. This is due to the variable order, patterns at the top and the bottom of the order in the decision tree have fewer copies of themselves in the decision diagram than if the variables of the pattern were placed in the middle.

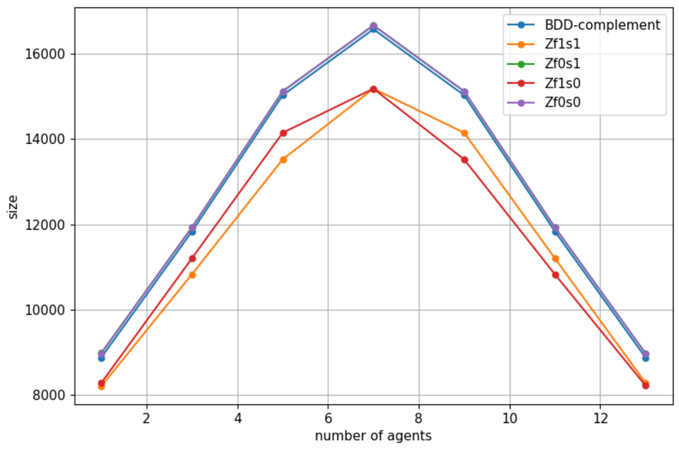


Figure 2.6: *Average sizes per nr. of payers, where nr. of diners = 13.*

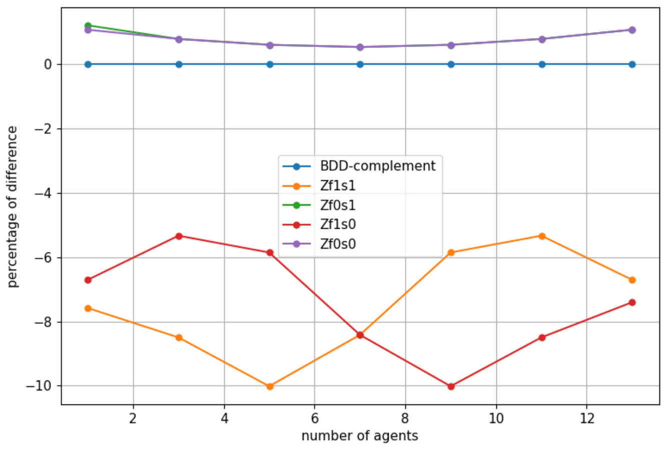


Figure 2.7: *Difference of average sizes in percentage w.r.t. the BDD-complement representation, per nr. of payers, where nr. of diners = 13.*

Sum and Product

Because we have one parameter to vary we only have to consider the plot for the average size as we increase the maximum sum (Figure 3.1) and its "relative difference to BDD" version (Figure 3.2). The plots show that the f_1 variants perform consistently better (38 to 45% smaller for f_1s_1 and 45 to 54% for f_1s_0) whereas the f_0 variants are an insignificant amount larger than the BDDs.

The f_1s_1 and f_1s_0 lines meet up and split up at intervals in the graph, namely when 64, 128 and 256 are taken as maximum sum (highlighted by vertical grey lines). As explained in our expectations, this is due to the boolean (bit) representation used for numbers.

Because the maximum allowed sum is lower than the range, many of the paths containing numbers with this bit set to 1 are eliminated. This gives an advantage for the s_0 variant, diminishing as the maximum sum grows until the current range needs to be extended, which in turn renews the s_0 advantage again.

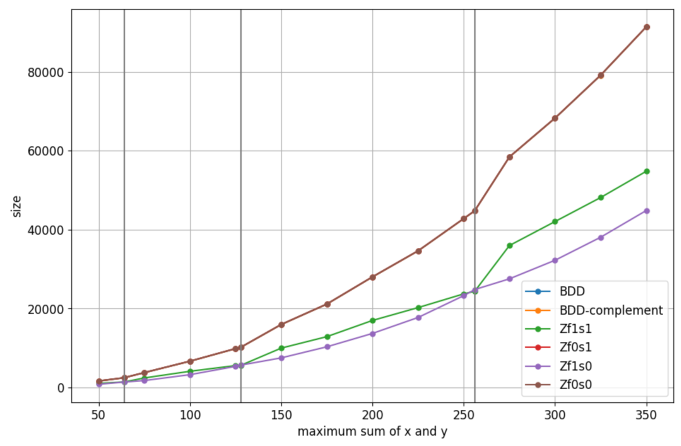


Figure 3.1: *Average sizes per maximum sum of the hidden numbers (x and y), with vertical grey lines at 64, 128 and 256. The f_0 ZDD variants, BDD and BDD-complement lines overlap.*

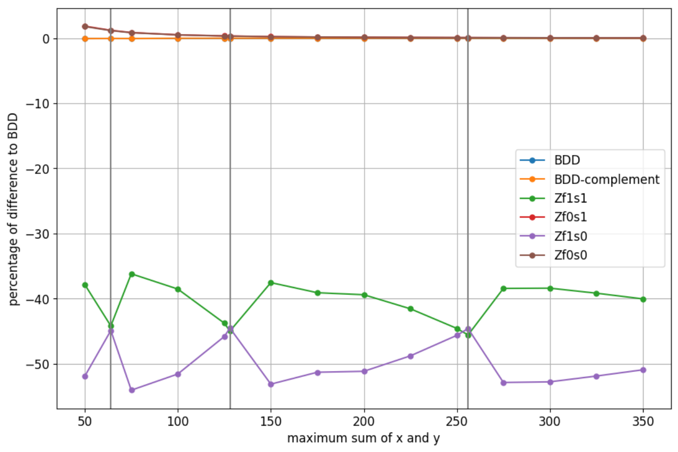


Figure 3.2: *Difference of average sizes in percentage per maximum sum of the hidden numbers (x and y), w.r.t. the BDD representation. The f_0 ZDD variants overlap, and the BDD and BDD-complement lines overlap.*

Higher Order Sally Anne

Figure 4.1 shows the average decision diagram sizes for all model states and Figure 4.2 shows the relative difference of sizes compared to the BDD representation. The plots confirm our expectation that the f_1s_1 is the most compact ZDD representation, although somewhat surprising it also outperforms the BDD representation (with a 25% size reduction at 16 children).

The data shows that, for our chosen parameter values, the belief law sizes are most compact for the f_1s_0 variant, while the f_1s_1 variant is the most compact state law. We therefore included a line (labelled "$f_1s_1 - f_1s_0$") for the most optimal combination of decision diagrams for representing the Sally Anne belief structures.

As we increase the number of children in the model, the belief laws have proportionally less influence on the total size which causes the f_1s_1 line to cross the f_1s_0 line after 14 children. The relative size differences do not seem to settle at an asymptote yet, experiments with more children could further investigate this - especially looking into whether, at some point, the f_1s_1 will be more compact than the "$f_1s_1 - f_1s_0$" belief structures.

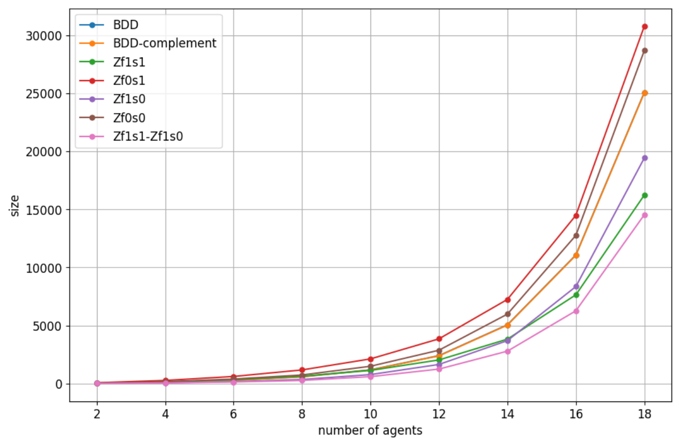


Figure 4.1: *Average sum of all laws sizes per nr. of agents.*

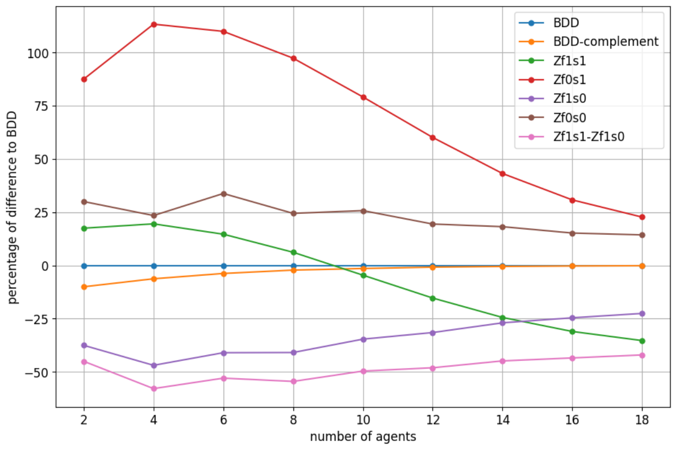


Figure 4.2: *Difference of average sum of all laws sizes in percentage w.r.t. the BDD representation, per nr. of agents.*

Figures 4.3 and 4.4 show a significant reduction in size for the BDDs when using belief indexing (e.i. 30% of 2400 nodes fewer for 12 children). The more compact variants (f_1s_1 and f_1s_0) are reduced less and the less compact (f_0s_1 and f_0s_0) are reduced more when adding belief indexing. The reduction reduces as we increase the number of children because the state law becomes proportionally larger compared to the sum of the belief laws. Because the disjointed and relatively small belief laws, belief indexing is only a significant size reduction in Higher Order Sally Anne for a small number of children.

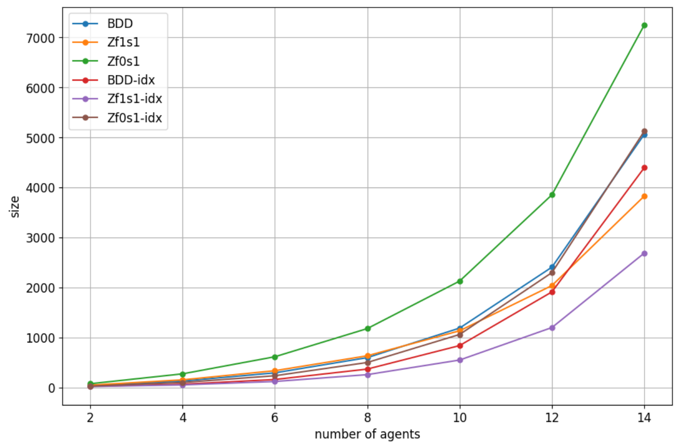


Figure 4.3: *Average (indexed belief laws plus state law) sizes per nr. of agents, where the less interesting f_1s_0, f_0s_0 and complement-BDD sizes are left out to reduce clutter.*

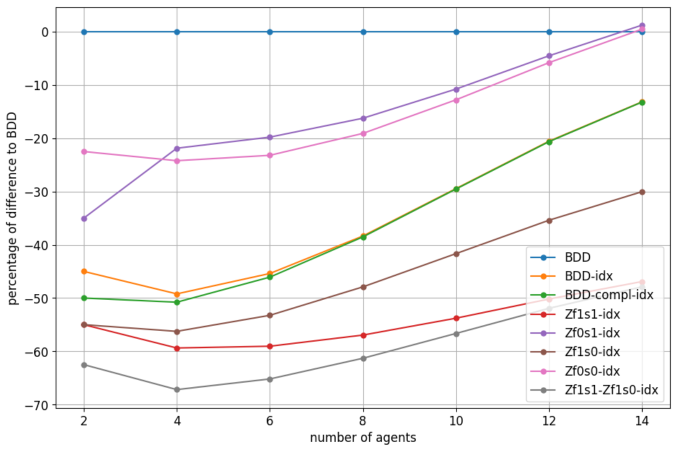


Figure 4.4: *Difference of average (indexed belief laws plus state law) sizes in percentage w.r.t. the not-indexed BDD size, per nr. of agents.*

3.2.2 Conclusions

The results of our experiments substantiate the following answers to our sub-hypotheses:

1. Using ZDDs are promising for common DEL problems. Even when we notice no indicators for ZDD advantage the best variants perform similarly to BDDs (as our muddy children results show, even outperforming BDDs when varying the number of dirty children). This conclusion holds for both the average sizes and worst-case sizes of the state laws. We postulate this is likely due to logic puzzles containing little redundant variables, as each variable serves some purpose to solving the puzzle. This might not be true for multi-agent dynamic epistemic logic applications in the real world.

2. XOR domains are be good predictors for f_1s_1 ZDD variant efficiency, however we cannot trust them completely; we found (from our dining cryptographers results) that an XOR domain can become an insignificant proportion of the decision diagram as we upscale the model such that the difference is determined by other patterns in the symbolic problem statement. Placing the domain in the middle of the variable order can further improve the f_1s_1 advantage.

3. From weak or strong sparsity in the model, we can predict whether the f_0 or f_1 variants, respectively, will be more efficient. The more extreme the sparsity the more significant the size reduction is (as our sum and product results show), although even non-extreme sparsity surprisingly predicts f_0 / f_1 compactness quite well (visible in the specific update sizes and second parameter plots of the muddy children puzzle).

4. Belief indexing can significantly reduce the size, although it is dependent on how proportionally large the belief laws are compared to the state law and on how many nodes can be shared between the belief laws.

This leads to our answer on the main research question:

Can ZDDs be more compact than BDDs for common DEL Model Checking tasks, and if so, can we predict from the task's symbolic representation which ZDD variant is more compact?

Our experiments have shown that there exist common DEL Symbolic Model Checking tasks for which ZDDs will be significantly more compact than BDDs. For these tasks, the most compact ZDD variant can be predicted by considering the combination of XOR-domains present in the symbolic problem statement and information about the sparsity of the set of states represented by the state law. The variant's improvement is proportional to the extremity of the sparsity and the presence of the XOR-domain in the decision diagram. In our experiments the improvements are consistent percentage of size reduction as we upscale the models.

3.2.3 Relevance and further research

Theoretical significance

Chapter 2, Theory, serves as an overview of how we can achieve symbolic model checking for DEL and how BDDs and ZDDs work. On top of this we have provided some novel insights on ZDDs specifically, especially:

- Rewording the elimination rules in decision diagrams to inference rules better explains the differences between BDDs and ZDD variants and in what kind of models they need to keep track of a specified context.
- Changing between ZDD elimination rule variants is equivalent to a combination of negating the children of each node and negating the entire decision diagram, or in symbolic representation: a

combination of negating all atomic propositions and the entire formula.

- The XOR-domain shows a common pattern in decision diagrams maximising ZDD efficiency and our formula calculates the exact node difference of ZDDs with BDDs if all children nodes are unique or the same. These then serve as bounds for all cases in between. The formula provides quantification of the advantage for using ZDDs in tasks containing an XOR-domain within the state law or belief laws.

Practical significance

In this work we have provided the first symbolic model checker for DEL with ZDDs. Adding ZDD functionality is a further development of SMCDEL, further developing it as a teaching tool on symbolic model checking and facilitating further research. It is also another step towards a state of the art knowledge base solver/reasoner for information systems containing multiple agents.

Furthermore, our experiments with the updated SMCDEL shows that using ZDDs is practically worth it for DEL models, and it is possible to predict which variant should be used based on the problem statement. Using the correct variant can reduce memory and time significantly. These results incentivise future research with decision diagrams to not consider only BDDs as the standard option.

Future research

No research is ever quite finished and also here much more can done to elaborate and give further insurance for the conclusions we form. We recommend the following future research directions:

- **Investigation problem domain**
  An overview of (real-world) practical applications of DEL model checking and what type of patterns and sparsity these tasks contain will solidify our conclusions on ZDDs outperforming BDDs.
- **More patterns and logic puzzles**
  We only tested on a hand-picked subset of logic puzzles, based on specific patterns in the formulation. Further research can look into other common patterns in well-known logic puzzles, in order to check whether there are any puzzles that are more compact when using BDDs.
- **Building with ZDDs**
  SMCDEL can be extended with another decision diagram manipulation library such that building with ZDDs can achieved. Optimising building with ZDDs has not been researched yet (aside of a small example given by Minato in [Min01]), and would require keeping the specified context as small as possible (as in case 8 of Example 2.5).
- **Extending DEL to Linear Temporal Logic (LTL) and Computational Tree Logic (CTL)**
  Model checking is often used for formal verification of software, which has concentrated on modeling with LTL [Kam68] and CTL [CES86]. There are BDD model checkers available implementing LTL and CTL ([CG18] gives a clear overview of the methods needed), but comparatively little attention has been given to temporal logics of knowledge, although it is useful in the specifications of protocols for distributed systems. SMCDEL can be extended with operators to also represent such systems, and be the first model checker providing ZDD functionality on this domain.
- **Combining ZDDs and BDDs**
  There is a rapidly growing amount of work on combining elimination rules in the same decision diagram in order to further reduce the structure [Bry18] [Bab+19]. These structures can reduce boolean representations in SMCDEL even further, and patterns indicating a specific elimination rule's advantage (such as our XOR-domain) can help optimise what ZDD or BDD elimination rules should be applied to corresponding sub graphs of the decision diagrams.
- **Dynamic contexts for belief structures**
  A multi-agent knowledge base solver in (real-world) application often deals with a changing number of agents and propositions to reason over. Currently SMCDEL has to recalculate the model

whenever the vocabulary or the list of agents is adjusted in its model, making it unsuitable for such an application. As dynamically adjusting content of our model would require statements over potentially infinite propositions and agents, all inference rules need to be combined to keep the paths finite. This sounds like a fun project to me personally, and it seems like a promising approach for translating Public Inspection (as in Section 5.3 in [Gat18a]) and Temporal Logic operators to methods in our decision diagram based model checker.

Bibliography

[Ake78] Sheldon B. Akers. “Binary decision diagrams”. In: *IEEE Transactions on computers* 6 (1978), pages 509–516. DOI: https://doi.org/10.1109/tc.1978.1675141.

[AMS04] Fadi A Aloul, Igor L Markov, and Karem A Sakallah. “MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation.” In: *J. Univers. Comput. Sci.* 10.12 (2004), pages 1562–1596. URL: http://www.jucs.org/jucs_10_12/mince_a_static_global/Aloul_F_A.pdf.

[Ars+20] Burcu Arslan, Rineke Verbrugge, Niels Taatgen, and Bart Hollebrandse. “Accelerating the Development of Second-Order False Belief Reasoning: A Training Study With Different Feedback Methods”. English. In: *Child Development* 91.1 (Jan. 2020), pages 249–270. ISSN: 0009-3920. DOI: https://doi.org/10.1111/cdev.13186.

[Bab+19] Junaid Babar, Chuan Jiang, Gianfranco Ciardo, and Andrew Miner. “Binary Decision Diagrams with Edge-Specified Reductions”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Edited by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pages 303–318. ISBN: 978-3-030-17465-1.

[BLF85] Simon Baron-Cohen, Alan M. Leslie, and Uta Frith. “Does the autistic child have a “theory of mind” ?” In: *Cognition* 21.1 (1985), pages 37–46. ISSN: 0010-0277. DOI: https://doi.org/10.1016/0010-0277(85)90022-8.

[Bry18] Randal E. Bryant. “Chain Reduction for Binary and Zero-Suppressed Decision Diagrams”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Edited by Dirk Beyer and Marieke Huisman. Cham: Springer International Publishing, 2018, pages 81–98. ISBN: 978-3-319-89960-2. DOI: https://doi.org/10.48550/arXiv.1710.06500.

[Bry86] Randal E Bryant. “Graph-based algorithms for boolean function manipulation”. In: *Computers, IEEE Transactions on* 100.8 (1986), pages 677–691. DOI: https://doi.org/10.1109/TC.1986.1676819.

[Bur+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. “Symbolic model checking: 1020 states and beyond”. In: *Information and computation* 98.2 (1992), pages 142–170. DOI: https://doi.org/10.1016/0890-5401(92)90017-a.

[CE81] Edmund M Clarke and E Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Workshop on Logic of Programs.* Springer. 1981, pages 52–71. DOI: https://doi.org/10.1007/bfb0025774.

[CES86] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pages 244–263. DOI: https://doi.org/10.1145/5397.5399.

[CG18] Sagar Chaki and Arie Gurfinkel. “BDD-Based Symbolic Model Checking”. In: *Handbook of Model Checking.* Edited by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pages 219–245. ISBN: 978-3-319-10575-8. DOI: https://doi.org/10.1007/978-3-319-10575-8_8.

[Cha88] David Chaum. “The dining cryptographers problem: Unconditional sender and recipient untraceability”. In: *Journal of cryptology* 1.1 (1988), pages 65–75. DOI: https://doi.org/10.1007/BF00206326.

[Cim+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. “NuSMV 2: An open-source tool for symbolic model checking”. In: *International Conference on Computer Aided Verification*. Springer. 2002, pages 359–364. DOI: `https://doi.org/10.1007/3-540-45657-0_29`.

[Cla+01a] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. “Bounded model checking using satisfiability solving”. In: *Formal methods in system design* 19.1 (2001), pages 7–34. DOI: `https://doi.org/10.1016/s0065-2458(03)58003-2`.

[Cla+01b] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Progress on the state explosion problem in model checking”. In: *Informatics*. Springer. 2001, pages 176–194. DOI: `https://doi.org/10.1007/3-540-44577-3_12`.

[CMB90] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. “Verifying temporal properties of sequential machines without building their state diagrams”. In: *International Conference on Computer Aided Verification*. Springer. 1990, pages 23–32. DOI: `https://doi.org/10.1007/bfb0023716`.

[Cou97] O. Coudert. “Solving graph optimization problems with ZBDDs”. In: *Proceedings European Design and Test Conference. ED TC 97*. 1997, pages 224–228. DOI: `https://doi.org/10.1109/EDTC.1997.582363`.

[DRV07] H. P. van Ditmarsch, J. Ruan, and R. Verbrugge. “Sum and Product in Dynamic Epistemic Logic”. In: *Journal of Logic and Computation* 18.4 (Dec. 2007), pages 563–588. ISSN: 0955-792X. DOI: `https://doi.org/10.1093/logcom/exm081`.

[DS01] Rolf Drechsler and Detlef Sieling. “Binary decision diagrams in theory and practice”. In: *International Journal on Software Tools for Technology Transfer* 3.2 (2001), pages 112–136. DOI: `https://doi.org/10.1007/s100090100056`.

[EDV09] J van Eijck, H van Ditmarsch, and Rineke Verbrugge. “Publieke werken: Freudenthal’s som-en-productraadsel”. In: *Nieuw archief voor wiskunde. Serie 5* 10.2 (2009), pages 126–131. URL: `https://dspace.library.uu.nl/handle/1874/314090`.

[Eij07] Jan van Eijck. “a demo of epistemic modelling”. In: *Interactive Logic. Selected Papers from the 7th Augustus de Morgan Workshop, London*. Volume 1. 2007, pages 303–362. URL: `https://homepages.cwi.nl/~jve/papers/07/pdfs/DEMO_IL.pdf`.

[Eij14] Jan van Eijck. *DEMO-S5*. Technical report. Tech. rep., CWI, 2014. URL: `https://homepages.cwi.nl/~jve/software/demo_s5`.

[Fag+95] Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. “Reasoning about knowledge, vol. 4”. In: *The MIT Press, Cambridge, MA* 10 (1995), page 208454. DOI: `https://doi.org/10.7551/mitpress/5803.001.0001`.

[Gat18a] Malvin Gattinger. “New directions in model checking dynamic epistemic logic”. PhD thesis. 2018. DOI: `https://doi.org/10.1007/978-3-662-48561-3_30`.

[Gat18b] Malvin Gattinger. “SMCDEL–An Implementation of Symbolic Model Checking for Dynamic Epistemic Logic with Binary Decision Diagrams”. In: 0 (2018), pages 7–92. URL: `https://github.com/jrclogic/SMCDEL`.

[GPS98] Clemens Gröpl, Hans Prömel, and Anand Srivastav. “Size and Structure of Random Ordered Binary Decision Diagrams”. In: volume 1373. Feb. 1998, pages 238–248. ISBN: 978-3-540-64230-5. DOI: `https://doi.org/10.1007/BFb0028565`.

[GV04] Peter Gammie and Ron Van Der Meyden. “MCK: Model checking the logic of knowledge”. In: *International Conference on Computer Aided Verification*. Springer. 2004, pages 479–483. DOI: `https://doi.org/10.1007/978-3-540-27813-9_41`.

[HAA18] Amjad Hawash, Ahmed Awad, and Baker Abdalhaq. “A Comparative Analysis of Binary Decision Diagram Reordering Algorithms for Reversible Circuit Synthesis”. In: Sept. 2018. DOI: `https://doi.org/10.1109/SSCI.2018.8628765`.

[IM13] Hiroaki Iwashita and Shin-ichi Minato. “Efficient top-down ZDD construction techniques using recursive specifications”. In: (2013). DOI: `https://doi.org/10.1.1.646.4753`.

[IMZ07] Haruya Iwasaki, Shin-ichi Minato, and Thomas Zeugmann. “A method of variable ordering for zero-suppressed binary decision diagrams in data mining applications”. In: *2007 IEEE International Workshop on Databases for Next Generation Researchers*. IEEE. 2007, pages 85–90. DOI: `https://doi.org/10.1109/SWOD.2007.353203`.

[Jai+94] J. Jain, J. Bitner, D. Moundanos, J.A. Abraham, and D.S. Fussell. “A new scheme to compute variable orders for binary decision diagrams”. In: *Proceedings of 4th Great Lakes Symposium on VLSI*. 1994, pages 105–108. DOI: `https://doi.org/10.1109/GLSV.1994.289986`.

[Kam68] Hans Kamp. “Tense Logic and the Theory of Linear Order”. Published as Johan Anthony Willem Kamp. PhD thesis. University of California Los Angeles, 1968. URL: `http://www.ims.uni-stuttgart.de/archiv/kamp/files/1968.kamp.thesis.pdf`.

[Kar88] Kevin Karplus. *Using if-then-else DAGs for multi-level logic minimization*. Computer Research Laboratory, University of California, Santa Cruz, 1988. DOI: `https://doi.org/10.1.1.144.6368`.

[Knu11] Donald E Knuth. *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India, 2011.

[Knu64] Donald E Knuth. “Backus normal form vs. backus naur form”. In: *Communications of the ACM* 7.12 (1964), pages 735–736. DOI: `http://dx.doi.org/10.1145/355588.365140`.

[LL92] Heh-Tyan Liaw and Chen-Shang Lin. “On the OBDD-representation of general Boolean functions”. In: *IEEE Transactions on computers* 41.06 (1992), pages 661–664. DOI: `https://doi.org/10.1109/12.144618`.

[LM53] John E Littlewood and A Mathematician’s Miscellany. “Methuen & Co”. In: *Ltd., London* (1953). URL: `https://archive.org/details/mathematiciansmi033496mbp`.

[LQR17] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. “MCMAS: an open-source model checker for the verification of multi-agent systems”. In: *International Journal on Software Tools for Technology Transfer* 19.1 (2017), pages 9–30. DOI: `https://doi.org/10.1007/s10009-015-0378-x`.

[LR06] Alessio Lomuscio and Franco Raimondi. “MCMAS: A model checker for multi-agent systems”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2006, pages 450–454. DOI: `https://doi.org/10.1007/978-3-642-02658-4_55`.

[MB+88] Jean-Christophe Madre, Jean-Paul Billon, et al. “Proving circuit correctness using formal comparison between expected and extracted behaviour”. In: *DAC*. Volume 88. 1988, pages 205–210. DOI: `https://doi.org/10.1109/DAC.1988.14759`.

[McM93] Kenneth L McMillan. “Symbolic model checking”. In: *Symbolic Model Checking*. Springer, 1993, pages 25–60. DOI: `https://doi.org/10.1007/978-1-4615-3190-6_3`.

[Min01] Shin-ichi Minato. “Zero-suppressed BDDs and their applications”. In: *International Journal on Software Tools for Technology Transfer* 3.2 (2001), pages 156–170. DOI: `https://doi.org/10.1007/s100090100038`.

[MIN13] Shin-ichi MINATO. “Techniques of BDD/ZDD: Brief History and Recent Activity”. In: *IEICE Transactions on Information and Systems* E96.D.7 (2013), pages 1419–1429. DOI: `https://doi.org/10.1587/transinf.E96.D.1419`.

[Min93] Shin-ichi Minato. “Zero-suppressed BDDs for set manipulation in combinatorial problems”. In: *Proceedings of the 30th international Design Automation Conference*. 1993, pages 272–277. DOI: `https://doi.org/10.1145/157485.164890`.

[Min94] Shin-ichi Minato. “Calculation of unate cube set algebra using zero-suppressed BDDs”. In: *31st Design Automation Conference*. IEEE. 1994, pages 420–424. DOI: `https://doi.org/10.1145/196244.196446`.

[NV19] Jim Newton and Didier Verna. “A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams”. In: *ACM Transactions on Computational Logic (TOCL)* 20.1 (2019), pages 1–36. DOI: `https://doi-org/10.1145/3274279`.

[OMI98] Hiroshi G. Okuno, Shin-ichi Minato, and Hideki Isozaki. “On the properties of combination set operations”. In: *Information Processing Letters* 66.4 (1998), pages 195–199. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/S0020-0190(98)00067-2`.

[Pix90] Carl Pixley. “A Computational Theory and Implementation of Sequential Hardware Equivalence. 1990”. In: (1990), pages 54–64. DOI: `https://doi.org/10.1007/bfb0023719`.

[Pla07] Jan Plaza. “Logics of public communications”. In: *Synthese* 158.2 (2007), pages 165–179. DOI: `https://doi-org/10.1007/s11229-007-9168-7`.

[PSP94] Shipra Panda, Fabio Somenzi, and Bernard F Plessier. “Symmetry detection and dynamic variable ordering of decision diagrams”. In: *ICCAD*. Citeseer. 1994, pages 628–631. DOI: `https://doi.org/10.5555/191326.191598`.

[QS82] Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *International Symposium on programming*. Springer. 1982, pages 337–351. DOI: `https://doi.org/10.1007/3-540-11494-7_22`.

[Rud93] Richard Rudell. “Dynamic variable ordering for ordered binary decision diagrams”. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE. 1993, pages 42–47. DOI: `https://doi.org/10.1109/ICCAD.1993.580029`.

[SC06] Radu I Siminiceanu and Gianfranco Ciardo. “New metrics for static variable ordering in decision diagrams”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2006, pages 90–104. DOI: `https://doi.org/10.1007/11691372_6`.

[Sha49] Claude E Shannon. “The synthesis of two-terminal switching circuits”. In: *The Bell System Technical Journal* 28.1 (1949), pages 59–98. DOI: `https://doi.org/10.1002/j.1538-7305.1949.tb03624.x`.

[SLZ04] Kaile Su, Guanfeng Lv, and Yan Zhang. “Reasoning about Knowledge by Variable Forgetting.” In: *KR* 4 (2004), pages 576–586. DOI: `https://doi.org/10.1613/jair.2750`.

[VDK07] Hans Van Ditmarsch, Wiebe van Der Hoek, and Barteld Kooi. *Dynamic epistemic logic*. Volume 337. Springer Science & Business Media, 2007. DOI: `https://doi.org/10.1007/978-1-4020-5839-4`.

[VHR13] Hans Van Ditmarsch, Wiebe van der Hoek, and Ji Ruan. “Connecting dynamic epistemic and temporal epistemic logics”. In: *Logic Journal of the IGPL* 21.3 (2013), pages 380–403. DOI: `https://doi.org/10.1093/jigpal/jzr038`.

[VO07] Jan Van Eijck and Simona Orzan. “Epistemic verification of anonymity”. In: *Electronic Notes in Theoretical Computer Science* 168 (2007), pages 159–174. DOI: `https://doi.org/10.1016/j.entcs.2006.08.026`.

[Wal] Adam Walker. *cudd: Bindings to the CUDD binary decision diagrams library.* Version 0.1.0.0 for CUDD 2.5.0. 2015. URL: `https://hackage.haskell.org/package/cudd-0.1.0.0`.

[Weg94] Ingo Wegener. “The size of reduced OBDD’s and optimal read-once branching programs for almost all Boolean functions”. In: *IEEE transactions on computers* 43.11 (1994), pages 1262–1269. DOI: `https://doi.org/10.1109/12.324559`.

[WP83] Heinz Wimmer and Josef Perner. “Beliefs about beliefs: Representation and constraining function of wrong beliefs in young children’s understanding of deception”. In: *Cognition* 13.1 (1983), pages 103–128. ISSN: 0010-0277. DOI: `https://doi.org/10.1016/0010-0277(83)90004-5`.

Chapter 4

Appendix

4.1 Summary program additions

SMCDEL is written in Haskell, a type-safe purely functional programming language. This fits nicely to mathematical style, allowing code that resembles the original notation. This helps with catching mistakes, often before compiling.

Additionally Haskell does not recompute already evaluated expressions and only evaluates expressions in our program when they are needed. Aside of the speed up advantages, it allows for infinite structures, such as the list of natural numbers [0..] or an infinite supply of atomic propositional variables — as long as the program will only use a finite part during its run.

Building on top of the already existing SMCDEL we provide the following additions to the code on a new branch available at `https://github.com/dushiel/SMCDEL` to make our proposed experiments possible. In this section familiarity with the previous version of SMCDEL is assumed, we only discuss the ZDD implementation and we omit duplicate code for multiple variations of similar functions whenever appropriate. The code for knowledge and belief structures with ZDDs reflects the original code supporting BDDs, thus we hope this section still gives an intuitive representation of the workings of SMCDEL as a whole. References to the relevant sections of SMCDELs documentation and of the files are provided for each subsection, where examples and additional information can be found. For further reading on original code not treated in this section, we recommend the sections on explaining the parsing of mathematical expressions to Haskell, type safe variable management, reduction and optimization, and the module overview (from Sections 3.2, 3.5, 3.7, 3.9, respectively, in [Gat18a]).

SMDEL can be used in the interactive compiler ghci and compiled as a library. Additionally there there is a command-line and a web interface for using simple DEL model checking with knowledge structures.

4.1.1 ZDD functionality

Data types

The additions discussed here are in the files `src/SMCDEL/Symbolic/S5_CUDD.hs` *and* `src/SMCDEL/Internal/MyHaskCUDD.hs`*, from which the original code is explained in Section 3.3 from [Gat18a].*

Knowledge structures are treated as a data type in the original code, for which we can provide additional constructors indicating which representation of boolean functions is used. This can be used in pattern matching when needed, but also allows defining functions for general knowledge structures without distinction of what boolean state law representation type is used. When using CUDD we need to keep track of the specific manager being used in the knowledge structure to avoid it remembering variable declarations of previous runs.

```haskell
data KnowStruct =
  KnS Cudd.Cudd.DdManager [Prp] (Dd B) [(Agent,[Prp])]
  | KnSZ Cudd.Cudd.DdManager [Prp] (Dd Z) [(Agent,[Prp])]
  | KnSZs0 Cudd.Cudd.DdManager [Prp] (Dd Z) [(Agent,[Prp])]
  | KnSZf0 Cudd.Cudd.DdManager [Prp] (Dd Z) [(Agent,[Prp])]
  | KnSZf0s0 Cudd.Cudd.DdManager [Prp] (Dd Z) [(Agent,[Prp])]
  deriving (Eq,Show)
type KnState = [Prp]
type KnowScene = (KnowStruct,KnState)
```

Similarly the `Dd` phantom type is used for defining the pure BDD and ZDD representations, such that pattern matching can be used on `Dd B` and `Dd Z` when needed and otherwise the `Dd a` type is used for defining the generalised functions.

```haskell
newtype Dd x = ToDd Cudd.Cudd.DdNode deriving (Eq,Show)
data B
data Z

class DdF a where
  bot :: Cudd.Cudd.DdManager -> Dd a
  top :: Cudd.Cudd.DdManager -> Dd a
  var :: Cudd.Cudd.DdManager -> Int -> Dd a
  neg :: Cudd.Cudd.DdManager -> Dd a -> Dd a
  con :: Cudd.Cudd.DdManager -> Dd a -> Dd a -> Dd a
  dis :: Cudd.Cudd.DdManager -> Dd a -> Dd a -> Dd a
  xor :: Cudd.Cudd.DdManager -> Dd a -> Dd a -> Dd a
  equ :: Cudd.Cudd.DdManager -> Dd a -> Dd a -> Dd a
  imp :: Cudd.Cudd.DdManager -> Dd a -> Dd a -> Dd a
  impf0 :: Cudd.Cudd.DdManager -> Dd a -> Dd a -> Dd a
  exists :: Cudd.Cudd.DdManager -> Int -> Dd a -> Dd a
  forall :: Cudd.Cudd.DdManager -> Int -> Dd a -> Dd a
  existsSet :: Cudd.Cudd.DdManager -> [Int] -> Dd a -> Dd a
  forallSet :: Cudd.Cudd.DdManager -> [Int] -> Dd a -> Dd a
  conSet :: Cudd.Cudd.DdManager -> [Dd a] -> Dd a
  disSet :: Cudd.Cudd.DdManager -> [Dd a] -> Dd a
  xorSet :: Cudd.Cudd.DdManager -> [Dd a] -> Dd a
  ifthenelse :: Cudd.Cudd.DdManager -> Dd a -> Dd a -> Dd a -> Dd a
  restrict :: Cudd.Cudd.DdManager -> Dd a -> (Int,Bool) -> Dd a
  restrictSet :: Cudd.Cudd.DdManager -> Dd a -> [(Int,Bool)] -> Dd a
  restrictQ :: Cudd.Cudd.DdManager -> Dd a -> [Prp] -> (Int,Bool) -> Dd a
  restrictSetQ :: Cudd.Cudd.DdManager -> Dd a -> [Prp] -> [(Int,Bool)] -> Dd a
  restrictQs0 :: Cudd.Cudd.DdManager -> Dd a -> [Prp] -> (Int,Bool) -> Dd a
  restrictSetQs0 :: Cudd.Cudd.DdManager -> Dd a -> [Prp] -> [(Int,Bool)] -> Dd a
  gfp :: Cudd.Cudd.DdManager -> (Dd a -> Dd a) -> Dd a
  gfpf0 :: Cudd.Cudd.DdManager -> (Dd a -> Dd a) -> Dd a
```

As CUDD does not support ZDD representations for the variants other than the originally proposed one (f_1s_1) we instead change the interpretation of the given logic formulas and ZDDs differently according to the translations given in Definition 2.35, for which code is shown in the second to next sub-subsection.

Haskell bindings for ZDD manipulation with CUDD

The additions discussed here are in the files `Cudd/C.hs`, *and* `Cudd/Cudd.hs` , which further builds on the Haskell bindings for CUDD initially developed in [Wal] and extended by Gattinger for use in SMCDEL, available at `https://github.com/dushiel/cudd`.

The `Dd` manipulation functions are called from another module which in turn imports the functions from the CUDD library written in C. Following the previous structure of importing CUDD functions we start by importing them from *cudd/cudd.c* to a Haskell module in *Cudd/C.hs* (with a capital C).

```haskell
foreign import ccall safe "cudd.h Cudd_zddReadOne_withRef_s"
    c_cuddZddReadOneWithRef :: Ptr CDdManager -> IO (Ptr CDdNode)
```

These are imported to a module (in *Cudd/Cudd.hs*) where the foreign pointers, needed referencing and de-referencing is handled. We also use `UnsafePerformIO` to convince the GHC (Haskell compiler) that these are pure functions. For most used functions the CUDD manual - or accompanying comments in the code - tell us whether they have side effects or not. An online version of the 2.4.1 version can be accessed via `http://web.mit.edu/sage/export/tmp/y/usr/share/doc/polybori/cudd/cuddIntro.html`.

```haskell
cuddZddReadOne :: DdManager -> DdNode
cuddZddReadOne (DdManager d) = DdNode $ unsafePerformIO $ do
    node <- c_cuddZddReadOneWithRef d
    newForeignPtrEnv derefZ d node
```

We can then call these functions in the MyHaskCudd file which serves as a module for our `Dd` manipulation functions, such that we can use them freely when interpreting formulas.

```haskell
instance DdF Z where
  top mgr = ToDd (Cudd.Cudd.cuddZddReadOne mgr)
  bot mgr = ToDd (Cudd.Cudd.cuddZddReadZero mgr)
  var mgr n = ToDd (Cudd.Cudd.cuddZddIthVar mgr n)
  neg mgr z = ifthenelse mgr z (bot mgr :: Dd Z) (top mgr :: Dd Z)
  con mgr (ToDd z1) (ToDd z2) = ToDd (Cudd.Cudd.cuddZddIntersect mgr z1 z2)
  dis mgr (ToDd z1) (ToDd z2) = ToDd (Cudd.Cudd.cuddZddUnion mgr z1 z2)
  -- ... etc
```

Building ZDDs

The additions discussed here are in the file `src/SMCDEL/Symbolic/S5_CUDD.hs` .

Even though CUDD does not support universal or existential abstraction we can implement building ZDDs from boolean logic formulas (without the abstraction operators). Public announcements are still possible because they do not use any knowledge operators. Using another library we can implement the other functions easily. Only the code for the f_0s_0 variant of ZDD representation is given here, but it closely resembles the BDD and other ZDD variants' code. The `f0` indicates that all non-true information is represented, thus the given logic formula interpreted as its negation. The `s0` indicates that all positive literals are inferred (whenever possible), and interprets all literals as their negation from the given logic formula. This reflects the methods from Definition 2.35.

```haskell
--ZDD in f0s0 form

zddOf (KnSZf0s0 mgr _ _ _)   Top           = bot mgr :: Dd Z
zddOf (KnSZf0s0 mgr _ _ _)   Bot           = top mgr :: Dd Z
zddOf (KnSZf0s0 mgr _ _ _)   (PrpF (P n))  = neg mgr (var mgr n :: Dd Z)
zddOf (KnSZf0s0 mgr _ _ _)   (Neg (PrpF (P n)))    = var mgr n :: Dd Z
zddOf kns@(KnSZf0s0 mgr _ _ _) (Neg form) = neg mgr (zddOf kns form)
zddOf kns@(KnSZf0s0 mgr _ _ _) (Conj forms)  = disSet mgr $ map (zddOf kns) forms
zddOf kns@(KnSZf0s0 mgr _ _ _) (Disj forms)  = conSet mgr $ map (zddOf kns) forms
zddOf kns@(KnSZf0s0 mgr _ _ _) (Xor  forms)  = xorSet mgr $ map (zddOf kns) forms

zddOf kns@(KnSZf0s0 mgr _ _ _) (Impl f g)    = impf0 mgr (zddOf kns f) (zddOf kns g)
zddOf kns@(KnSZf0s0 mgr _ _ _) (Equi f g)    = equ mgr (zddOf kns f) (zddOf kns g)

zddOf (KnSZf0s0 _ _ _ _) (Forall _ _) = error "forall not implemented with ZDD"
zddOf (KnSZf0s0 _ _ _ _) (Exists _ _) = error "exists not implemented with ZDD"

zddOf KnSZf0s0{} K {} = error "knowledge operators not implemented with ZDD"
zddOf KnSZf0s0{} Kw {} = error "knowledge operators not implemented with ZDD"
zddOf KnSZf0s0{} Ck {} = error "knowledge operators not implemented with ZDD"
zddOf KnSZf0s0{} Ckw {} = error "knowledge operators not implemented with ZDD"
zddOf KnSZf0s0{} Announce {} = error "announce operator not implemented with ZDD"
zddOf KnSZf0s0{} AnnounceW {} = error "announce operator not implemented with ZDD"

zddOf kns@(KnSZf0s0 mgr  _ _ _) (PubAnnounce form1 form2) = impf0 mgr (zddOf kns form1) newform2 where
    newform2 = zddOf (pubAnnounce kns form1) form2

zddOf kns@(KnSZf0s0 mgr  _ _ _) (PubAnnounceW form1 form2) =
  ifthenelse mgr (zddOf kns form1) newform2b newform2a where
    newform2a = zddOf (pubAnnounce kns form1) form2
    newform2b = zddOf (pubAnnounce kns (Neg form1)) form2

zddOf _ (Dia _ _) = error "Dynamic operators are not implemented for CUDD."
zddOf _ _ = error "zzddOf with a wrong kns type"
```

Converting to different variants

The additions discussed here are in the files `src/SMCDEL/Symbolic/S5_CUDD.hs` *and* `src/SMCDEL/Internal/MyHaskCUDD.hs`.

For conversion between the different boolean representations we label the knowledge structure accordingly and change the law following the conversion methods introduced in Definition 2.35.

```haskell
toKnsZf0s0 :: KnowStruct -> KnowStruct
toKnsZf0s0 (KnS mgr vocab law obs) = KnSZf0s0 mgr vocab (swapChildNodes mgr (neg mgr $ createZddFromBdd
↪  mgr law) (map fromEnum vocab)) obs
toKnsZf0s0 (KnSZ mgr vocab law obs) = KnSZf0s0 mgr vocab (swapChildNodes mgr (neg mgr law) (map
↪  fromEnum vocab)) obs
toKnsZf0s0 (KnSZs0 mgr vocab law obs) = KnSZf0s0 mgr vocab (neg mgr law) obs
toKnsZf0s0 (KnSZf0 mgr vocab law obs) = KnSZf0s0 mgr vocab (swapChildNodes mgr law (map fromEnum
↪  vocab)) obs
toKnsZf0s0 kns@(KnSZf0s0 _ _ _ _) = kns
```

Where `swapChildNodes` complements all literals by swapping the edges of all nodes in the graph with the provided `cuddZddChange` function. This is only available for ZDD variants as it does not change the size for the BDD type. It takes as second argument the context of the function in integers which we can get from the vocabulary of the knowledge structure.

```haskell
swapChildNodes Dd Z -> [Int] -> Dd Z
swapChildNodes mgr (ToDd z) [n] = ToDd $ Cudd.Cudd.cuddZddChange mgr z n
swapChildNodes mgr (ToDd z) (n:ns) = swapChildNodes mgr (ToDd $ Cudd.Cudd.cuddZddChange mgr z n) ns
swapChildNodes _   z        [] = z

swapChildNodes Dd B -> [Int] -> Dd B
swapChildNodes mgr b [n] = con mgr (imp mgr (neg mgr (var mgr n)) (restrict mgr b (n,True))) (imp mgr
↪  (var mgr n) (restrict mgr b (n,False)))
swapChildNodes mgr b (n:ns) = con mgr (con mgr (imp mgr (neg mgr (var mgr n)) (restrict mgr b
↪  (n,True))) (con mgr (var mgr n) (restrict mgr b (n,False)))) (swapChildNodes mgr b ns)
swapChildNodes _ b [] = b
```

Testing on arbitrary formulas

The additions discussed here are in the files `test/CUDD_S5.hs`, `src/SMCDEL/Symbolic/S5_CUDD.hs` *and* `src/SMCDEL/Internal/MyHaskCUDD.hs`. *The original code is discussed in Section 3.10 of [Gat18a].*

It is good practice in modern software engineering to test implementations against specifications, especially for a model checker which itself is meant to check specifications.

For randomized property-based testing, we use the QuickCheck library [CH00] to test the conversion functions used in our experiments. A SimplifiedForm is an arbitrarily produced formula containing operators and propositions at random, while removing some redundancy. The code for the construction of these formulas can be found in `src/SMCDEL/Language.hs`.

```haskell
conversionDdTest :: SimplifiedForm -> [Bool]
conversionDdTest (SF f) =
  [ Cudd.evalViaBdd (myKnS, myDefaultState) f `debug` ("for f : " ++ show f ++ "\n\n")
    , fst b `debug` ("bdd: " ++ show (fst b) ++ ", size = " ++ show(snd b))
    , fst z `debug` ("zdd: " ++ show (fst z) ++ ", size = " ++ show(snd z))
    , fst zf0 `debug` ("zddf0: " ++ show (fst zf0) ++ ", size = " ++ show(snd zf0))
    , fst zs0 `debug` ("zdds0: " ++ show (fst zs0) ++ ", size = " ++ show(snd zs0))
    , fst zf0s0 `debug` ("zddf0s0: " ++ show (fst zf0s0) ++
            ", size = " ++ show(snd zf0s0) ++ "\n \n")
  ] where
```

```haskell
    b = Cudd.evalViaDd (myKnS, myDefaultState) f
    z = Cudd.evalViaConvertedZDD (myKnS, myDefaultState) f
    zf0 = Cudd.evalViaConvertedZDDf0 (myKnS, myDefaultState) f
    zs0 = Cudd.evalViaConvertedZDDs0 (myKnS, myDefaultState) f
    zf0s0 = Cudd.evalViaConvertedZDDf0s0 (myKnS, myDefaultState) f
```

We test whether we get the same (boolean) evaluation, and for debugging the current code also prints the sizes of each constructed variant. The evaluation functions are similar to the conversion functions thus we again only show the most complicated variant $f_0 s_0$ here:

```haskell
evalViaConvertedZDDf0s0 :: KnowScene -> Form -> (Bool,Int)
evalViaConvertedZDDf0s0 (kns@(KnS mgr allprops _ _),s) f = (bool, nodeCount) where
  zf0s0 = swapChildNodes mgr (createZddFromBdd mgr (neg mgr $ bddOf kns f)) (map fromEnum allprops)
  nodeCount = size mgr zf0s0
  bool | b== (bot mgr :: Dd Z) = True
       | b== (top mgr :: Dd Z) = False
       | otherwise = error ("evalViaDd failed: ZDDs0 leftover:\n" ++ texDdZ mgr b)
  b    = restrictSetQs0 mgr zf0s0 allprops list
  list = [ (n, P n `elem` s) | (P n) <- allprops ]
evalViaConvertedZDDf0s0 _ _ = error "wrong KnS type\n"
```

restrictSetQ (as opposed to restrictSet - without a Q) takes a specified fixed context as argument as missing variables in ZDD paths matter for the result, unlike with BDDs. The s0 version interprets every variable as its complement.

```haskell
restrictQ mgr zdd u (n,bit) = productZ mgr (if bit then sub1 mgr zdd n else sub0 mgr zdd n)
↪  (exceptVarZContext mgr u n)
restrictQs0 mgr zdd u (n,bit) = productZ mgr (if bit then sub0 mgr zdd n else sub1 mgr zdd n)
↪  (exceptVarZContext mgr u n)


restrictSetQ _ _ zdd [] = zdd
restrictSetQ mgr u zdd n = foldl (restrictQ mgr u) zdd n
restrictSetQs0 _ _ zdd [] = zdd
restrictSetQs0 mgr u zdd n = foldl (restrictQs0 mgr u) zdd n
```

The functions sub1 and sub0 implement the $\boldsymbol{restrict}(f, p, 1)$ and $\boldsymbol{restrict}(f, p, 0)$ methods from Definition 2.27 and therefor removes the variable p from the ZDD context of function f. Since there is no specified context to keep track of, we have to use some trickery on top of a CUDD provided function wrapped as productZ in order to add the removed variable back in the ZDD as a dont-care node (both positive and negative literal).

Visualising decision diagrams for CUDD

The additions discussed here are in the file src/SMCDEL/Symbolic/S5_CUDD.hs.

The original SMCDEL only supports visualising decision diagrams for BDDs with the CacBDD library. In order to view the decision diagrams represented by CUDD we use its dumpDot function (wrapped as returnDot to get it as a String type), which generates a Dot file for a given DD, then we adjust the content to fit with the user given names for the variables and transform it to TikzPicture format in order to include it into latex.

```haskell
texDdZWith mgr d vocab = unsafePerformIO $ do
  let xDotText = B.pack $ returnDot mgr d
  let myShow = formatDotCUDD vocab
  let xDotGraph = parseDotGraphLiberally xDotText :: DotGen.DotGraph String
  let renamedXDotGraph = renameMyGraph xDotGraph myShow

  (i,o,_,_) <- runInteractiveCommand "dot2tex --figpreamble=\"\\huge\" --figonly -traw"
  hPutStr i (B.unpack (renderDot $ toDot renamedXDotGraph) ++ "\n")
  hClose i
  result <- hGetContents o
  return $ dropWhileEnd isSpace $ dropWhile isSpace result
```

Here `renameMyGraph` replaces all node labels with their proposition names (e.g. node labels $\{1,2\}$ becomes $\{P_1,P_2\}$). This is currently set up in SMCDEL to be only numbers, but can easily be changed, and `formatDotCUDD` formats the resulting dot file to more closely resemble the CacBDD version.

CUDD support for general K models with belief indexing

The additions discussed here are in the files `src/SMCDEL/Symbolic/K_CUDD.hs`, `src/SMCDEL/Symbolic/Ki_CUDD.hs`, `src/SMCDEL/Symbolic/Ki.hs` *and* `src/SMCDEL/Internal/MyHaskCUDD.hs`.

The code for CUDD belief structures closely resembles the code for CacBDD belief structures (`src/SMCDEL/Symbolic/K.hs`), therefore we immediately consider the code for model checking belief structures with agent belief indexing for brevity sake.

The belief laws have the double vocabulary $(V \cup V')$ as context, thus functions are implemented that transform the propositions to propositions in the double vocabulary. For belief indexing we do not translate to the double vocabulary by simply multiplying and dividing by 2, but we have to add and subtract the agent index propositions accordingly (in the code we use m to for the number of agents).

```haskell
mvP, cpP :: Int -> Prp -> Prp
mvP m (P n) = P  ((2*n) + m)      -- represent p  in the double vocabulary
cpP m (P n) = P  ((2*n) + 1 + m) -- represent p' in the double vocabulary

unmvcpP :: Int -> Prp -> Prp
unmvcpP m (P n) | even (n-m)    = P $ (n-m) `div` 2
                | otherwise = P $ (n-1-m) `div` 2

mv, cp :: Int -> [Prp] -> [Prp]
mv m = map (mvP m)
cp m = map (cpP m)

unmv, uncp :: Int -> [Prp] -> [Prp]
-- | Go from p in double vocabulary to p in single vocabulary.
unmv m = map f where
  f (P n) | odd (n-m)    = error "unmv failed: Number is odd!"
          | otherwise = P $ (n-m) `div` 2
-- | Go from p' in double vocabulary to p in single vocabulary.
uncp m = map f where
  f (P n) | even (n-m)    = error "uncp failed: Number is even!"
          | otherwise = P $ (n-m-1) `div` 2
```

Boolean representations with the double vocabulary as context are given a tagged type, such that we utilise Haskell's type-safety. For these RelBDDs and RelZDDs we also have to implement functions that translate back and forward with their boolean representations in the single vocabulary context.

```haskell
data Dubbel
type RelBDD = Tagged Dubbel (Dd B)
type RelZDD = Tagged Dubbel (Dd Z)

totalRelBdd, emptyRelBdd :: Cudd.Cudd.DdManager -> RelBDD
totalRelBdd mgr = pure $ boolBddOf mgr Top
emptyRelBdd mgr = pure $ boolBddOf mgr Bot

cpBdd :: Cudd.Cudd.DdManager -> Int -> Dd B -> RelBDD
cpBdd mgr m b = Tagged $ relabelFun mgr (\n -> (2*n) + 1 + m ) b

mvBdd :: Cudd.Cudd.DdManager -> Int -> Dd B -> RelBDD
mvBdd mgr m b = Tagged $ relabelFun mgr (\n -> (2 * n) + m) b

-- etc
```

CacBDD has `relabelFun` implemented in its library already, while for CUDD we still had to define the `relabelFun` function:

```haskell
relabelFun :: Cudd.Cudd.DdManager -> (Int -> Int) -> Dd B -> Dd B -- assumes no int mapping to itself in
↪  rf
relabelFun mgr rF bdd = loop bdd disjointListOfNodeLists
```

```haskell
  where

  --get indexes of "overlapping" integers positions (integers in l2 that also occur in L1)
  --and use that to return the corresponding elements in a tuple
  getOverlap l1 l2 = (map ((\x -> l1 !! x) . fromJust) (indexesOverlap l1 l2),mutuals l1 l2)
  indexesOverlap l1 l2 = map (\x -> elemIndex x l2) (mutuals l1 l2)
  indexesNotOverlap l1 l2 = map (\x -> elemIndex x l1) (mutuals l1 l2)

  -- swap the (overlapping l1 ints with the corresponding l1 ints)
  -- in the not overlapping l1 values we look for the overlapping l2 values

  newOverlap l1 l2 = (fst $ getOverlap l1 l2, map ((\x -> l1 !! x) . fromJust) (indexesNotOverlap l1
  ↪  l2))

  -- get a list of tuples containing 2 equal length, disjointed lists of vars to be swapped
  -- by removing the "overlapping" variables and performing a recursive call with them
  splitCompare [] [] = []
  splitCompare [] _ = error "varlists used for relabeling do not have equal length."
  splitCompare _ [] = error "varlists used for relabeling do not have equal length."
  splitCompare l1 l2 = (l1\\fst (getOverlap l1 l2),l2\\snd (getOverlap l1 l2)) : splitCompare (fst $
  ↪  newOverlap l1 l2) (snd $ newOverlap l1 l2)

  -- apply the function above to the support variable integers and the resulting integers from applying
  ↪  rf (the Remapping Function)
  -- and turn the integers into BDD nodes (as bddSwapVars requires this)
  disjointListOfLists = splitCompare support (map rF support)
  disjointListOfNodeLists = map (Data.Bifunctor.bimap (map (var mgr)) (map (var mgr)))
  ↪  disjointListOfLists :: [([Dd B],[Dd B])]
  support = getSupport mgr bdd

  -- loop and uncurry bddSwapVars so that it can be applied to the disjointListOfNodeLists
  loop b (n:ns) = loop (uncurry (bddSwapVars mgr b) n) ns
  loop b [] = b
```

The believe structures also get their data own type, where they can contain their laws in multiple types of boolean representations.

```haskell
data BelStruct = BlS Cudd.Cudd.DdManager  -- Cudd manager for removing/reseting variables
                     [Prp]                -- vocabulary
                     (Dd B)               -- state law
                     (M.Map Agent Int,RelBDD) -- observation laws
                | BlSZ Cudd.Cudd.DdManager [Prp] (Dd Z) (M.Map Agent Int,RelZDD)
                | BlSZf0 Cudd.Cudd.DdManager [Prp] (Dd Z) (M.Map Agent Int,RelZDD)
                | BlSZs0 Cudd.Cudd.DdManager [Prp] (Dd Z) (M.Map Agent Int,RelZDD)
                | BlSZf0s0 Cudd.Cudd.DdManager [Prp] (Dd Z) (M.Map Agent Int,RelZDD)
                -- mixed BDDs and ZDDs, to avoid typing all combinations i define only 3:
                | BlSunsafeZB Cudd.Cudd.DdManager [Prp] (Dd Z) (M.Map Agent Int,RelBDD)
                | BlSunsafeBZ Cudd.Cudd.DdManager [Prp] (Dd B) (M.Map Agent Int,RelZDD)
                | BlSunsafeZZ Cudd.Cudd.DdManager [Prp] (Dd Z) (M.Map Agent Int,RelZDD)
                  deriving (Eq,Show)

instance Pointed BelStruct KnState
type BelScene = (BelStruct,KnState)

instance Pointed BelStruct (Dd B)
type MultipointedBelScene = (BelStruct, Dd B)
```

Building is again only supported for BDDs, for the same reasons as mentioned before. The `bddOf` function for belief structures with agent belief indexing mainly has changed knowledge operators compared to the knowledge structures' `bddOf`; to manipulate the agent beliefs, the methods manipulating the BDD have to be lifted to work on Tagged BDDs and they have to restrict the BDD to the subgraphs where specific agent propositions hold.

```haskell
bddOf :: BelStruct -> Form -> Dd B
bddOf (BlS mgr _ _ _)   Top           = top mgr
bddOf (BlS mgr _ _ _)   Bot           = bot mgr
bddOf (BlS mgr _ _ _)   (PrpF (P n))  = var mgr n
bddOf bls@(BlS mgr _ _ _) (Neg form)    = neg mgr $ bddOf bls form
-- etc
```

```haskell

bddOf bls@(BlS mgr allprops lawbdd (ag, obdds)) (K i form) = unmvBdd mgr (M.size ag) result
  where
  result = forallSet mgr ps' <$> (imp mgr <$> cpBdd mgr (M.size ag) lawbdd <*> (imp mgr <$> omegai <*>
  ↪  cpBdd mgr (M.size ag) (bddOf bls form)))
  ps'    = map fromEnum $ cp (M.size ag) allprops
  omegai = Tagged $ restrict mgr (untag obdds) (ag ! i, True)

bddOf bls@(BlS mgr voc lawbdd (ag, obdds)) (Ck ags form) = lfp lambda (top mgr)  where
  ps' = map fromEnum $ cp (M.size ag) voc
  lambda :: Dd B -> Dd B
  lambda z = unmvBdd mgr (M.size ag) $
    forallSet mgr ps' <$>
      (imp mgr <$> cpBdd mgr (M.size ag) lawbdd <*>
        (imp mgr <$> (disSet mgr <$> sequence [Tagged $ restrict mgr (untag obdds) (ag ! i, True) | i
        ↪  <- ags]) <*>
          cpBdd mgr (M.size ag) (con mgr (bddOf bls form) z)))
-- etc
```

For conversion from the different types to each other also the belief laws have to be converted. Again we only show the most complicated f_0s_0 variant's conversion function:

```haskell
toBlsZf0s0 :: BelStruct -> BelStruct
toBlsZf0s0 (BlS mgr vocab law (ags, obs)) = BlSZf0s0 mgr vocab
    (swapChildNodes mgr (neg mgr $ createZddFromBdd mgr law) (map fromEnum vocab))
    (ags, (Tagged . neg mgr . createZddFromBdd mgr . swapChildNodes2 mgr (map fromEnum vocab) . untag)
    ↪  obs)
toBlsZf0s0 (BlSZ mgr vocab law (ags, obs)) = BlSZf0s0 mgr vocab
    (swapChildNodes mgr (neg mgr law) (map fromEnum vocab))
    (ags, (Tagged . neg mgr . swapChildNodes2 mgr (map fromEnum vocab) . untag) obs)
toBlsZf0s0 (BlSZs0 mgr vocab law (ags, obs)) = BlSZf0s0 mgr vocab (neg mgr law)
    (ags, (Tagged . neg mgr . untag) obs)
toBlsZf0s0 (BlSZf0 mgr vocab law (ags, obs)) = BlSZf0s0 mgr vocab
    (swapChildNodes mgr law (map fromEnum vocab))
    (ags, (Tagged . swapChildNodes2 mgr (map fromEnum vocab) . untag) obs)
toBlsZf0s0 bls@(BlSZf0s0 _ _ _ _) = bls
toBlsZf0s0 _ = error "conversion not defined for unsafe BlS types"
```

4.1.2 Puzzles and collecting data

We get the size data per logic puzzle by looping over each combination of the parameters (if the combination is valid) and build the knowledge/belief structure for each update with those parameters. CUDD's `cuddDagSize` and CacBDD's `sizeOf` methods are then used for gathering the size of the state (and belief) laws. Because CUDD treats the Bot-ZDD as an empty ZDD the function breaks when presented with the Bot ZDD, thus 0 is returned for that specific case. CacBDD's `sizeOf` does not count the leaf nodes thus we add these manually.

```haskell
size :: Bdd -> Int
size b = if sizeOf b == 0 then sizeOf b + 1 else sizeOf b + 2
```

```haskell
size :: Cudd.Cudd.DdManager -> Dd a -> Int
size mgr (ToDd dd)
  | ToDd dd == (bot mgr :: Dd Z) = 0
  | otherwise = Cudd.Cudd.cuddDagSize dd
```

There are already implementations in place for Muddy Children, Dining Cryptographers and Sum and Product, but these are for the single parameter cases. We add a second parameter to each logic model checking task according to our description given in the experimental set up section. The general version of Sally-Anne is implemented from scratch. Building BDDs with the CacBDD library (already implemented) gives us a non-complement edge version of the BDD representation because CUDD does not have a direct method for disabling the complement edge optimisation. The CacBDD library does not support ZDDs. The CUDD ZDDs do not have complement edge optimisation.

2-parameter muddy children

The additions discussed here are in the file `sizeExperiments/muddychildren.hs`*, where functions are used from* `src/SCMDEL/Examples/MuddyChildren.hs`*. The original code is discussed in Section 4.1 of [Gat18a].*

Initialising muddy children is the only part that changes when adding the second parameter, changing the number of dirty children each child can observe.

The functions `muddySizeCAC` and `muddySizeCUDD` take the two parameters as arguments and returns the sizes of the decision diagrams representing the state law for each update.

```haskell
muddySizeCAC :: Int -> Int -> [Int]
muddySizeCAC n m = loop 0 cuddMudScnInit  where
  cuddMudScnInit = S5_CAC.KnS (mudPs n) (S5_CAC.boolBddOf (father n)) [ (show x,delete (P x) (mudPs n))
  ↪  | x <- [1..n] ]
  loop i kns
    | i == 0 = info kns : loop (i+1) kns
    | i < m = info (kns `unsafeUpdate` nobodyknows n) : loop (i+1) (kns `unsafeUpdate` nobodyknows n)
    | i == m = []
    | otherwise = error ("something went wrong with loop: " ++ show i) where

      info (S5_CAC.KnS _ lawb _) = S5_CAC.size lawb


muddySizeCUDD :: (S5_CUDD.KnowStruct -> S5_CUDD.KnowStruct) -> Int -> Int -> IO [Int]
muddySizeCUDD convertfunc n m = do
  start@(S5_CUDD.KnS mgr _ _ _) <- genMuddyKnStructCudd n -- also creates the manager!
  MyHaskCUDD.initZddVars mgr [0..n]
  return (loop 0 start)  where
    loop i kns

      | i == 0 = info (convertfunc kns) : loop (i+1) (convertfunc kns)
      | i < m = info (kns `S5_CUDD.unsafeUpdate` nobodyknows n) : loop (i+1) (kns
      ↪  `S5_CUDD.unsafeUpdate` nobodyknows n)
      | i == m = []
      | otherwise = error ("something went wrong with loop: " ++ show i) where

        info (S5_CUDD.KnS mgr _ lawb _) = MyHaskCUDD.size mgr lawb
        info (S5_CUDD.KnSZ mgr _ lawz _) = MyHaskCUDD.size mgr lawz
        info (S5_CUDD.KnSZf0 mgr _ lawz _) = MyHaskCUDD.size mgr lawz
        info (S5_CUDD.KnSZs0 mgr _ lawz _) = MyHaskCUDD.size mgr lawz
        info (S5_CUDD.KnSZf0s0 mgr _ lawz _) = MyHaskCUDD.size mgr lawz
```

`findNumberWithSize` then takes the initial state (`mudScnInit`) and the evaluation function, which are both dependent on what decision diagram we are testing, and returns the sizes and number of updates needed for the final state. The number of updates needed should equal the number of dirty children, m, minus one.

```haskell
genMuddyKnStructCudd :: Int -> IO S5_CUDD.KnowStruct
genMuddyKnStructCudd n = do
  mgr <- MyHaskCUDD.makeManager
  let law = S5_CUDD.boolBddOf mgr (father n)
  let obs = [ (show x,delete (P x) (mudPs n)) | x <- [1..n] ]
  return $ S5_CUDD.KnS mgr (mudPs n) law obs
```

The `checkform` function provides the formula for the update step.

```haskell
checkForm :: Int -> Int -> Form
checkForm n 0 = nobodyknows n
checkForm n k = PubAnnounce (nobodyknows n) (checkForm n (k-1))
```

2-parameter Dining-Cryptographers

The additions discussed here are in the file `sizeExperiments/dyningcrypto.hs` *, where functions are used from* `src/SCMDEL/Examples/DyningCrypto.hs`.

For the two parameter version of the Dining-Cryptographers the initial state and the final check are different compared to the original implementation. We start with considering the functions for each decision diagram which return true or false depending on whether all diners know whether the NSA payed (among the m number of payers) and return the state law sizes for each update step. We do not have to perform the last update step as it always returns the sizes for Top no matter the parameter variations.

```haskell
genDcValidWithSparCudd :: Int -> Int -> IO (Bool, [Double])
genDcValidWithSparCudd n m = do
  startKns@(S5_CUDD.KnS mgr _ _ _) <- genDcKnsInitCudd n m
  return (S5_CUDD.validViaBdd startKns (genDcCheckForm n), loop 0 startKns (S5_CUDD.boolBddOf mgr Top))
  where
  loop i startKns@(S5_CUDD.KnS mgr allprops lawb _) current_law
    | i == 0 = MyHaskCUDD.sparsity mgr lawb (maximum (map fromEnum allprops)) : loop (i+1) startKns lawb `debug` "start compl"
    | i < n = MyHaskCUDD.sparsity mgr (MyHaskCUDD.con mgr current_law updateDc) (maximum (map fromEnum allprops)) : loop (i+1) startKns (MyHaskCUDD.con mgr current_law updateDc) --)`debug` "during"
    | i == n = []--[MyHaskCUDD.size (current_law `MyHaskCUDD.con` finalCheck)]`debug` "end"
    | otherwise = error "something went wrong with loop"
    where
      updateDc = S5_CUDD.bddOf startKns (genDcReveal n i)
      --finalCheck = S5_CUDD.bddOf startKns (Conj [ genDcEveryoneKnowsWhetherNSApaid n, genDcNobodyknowsWhoPaid n ])
  loop _ _ _ = error "initial structyre should be normal BDD"

genDcValidWithSizeCudd :: Int -> Int -> IO (Bool, [Int])
genDcValidWithSizeCudd n m = do
  startKns@(S5_CUDD.KnS mgr _ _ _) <- genDcKnsInitCudd n m
  return (S5_CUDD.validViaBdd startKns (genDcCheckForm n), loop 0 startKns (S5_CUDD.boolBddOf mgr Top) )
  where
  loop i startKns@(S5_CUDD.KnS mgr _ lawb _) current_law
    | i == 0 = MyHaskCUDD.size mgr lawb : loop (i+1) startKns lawb
    | i < n  = MyHaskCUDD.size mgr (MyHaskCUDD.con mgr current_law updateDc) : loop (i+1) startKns (MyHaskCUDD.con mgr current_law updateDc)
    | i == n = []
    | otherwise = error "something went wrong with loop"
    where
      updateDc = S5_CUDD.bddOf startKns (genDcReveal n i)
      --finalCheck = S5_CUDD.bddOf startKns (Conj [ genDcEveryoneKnowsWhetherNSApaid n, genDcNobodyknowsWhoPaid n ])
  loop _ _ _ = error "initial structure should be normal BDD"

genDcValidWithSizeCuddZ :: Int -> Int -> IO (Bool, [Int])
genDcValidWithSizeCuddZ n m = do
  startKns@(S5_CUDD.KnS mgr _ _ _) <- genDcKnsInitCudd n m
  return (fst $ S5_CUDD.validViaConvertedZDD (S5_CUDD.toKnsZ startKns) (genDcCheckForm n), loop 0 startKns (S5_CUDD.boolBddOf mgr Top))
  where
  loop i startKns@(S5_CUDD.KnS mgr _ lawb _) current_law
    | i == 0 = MyHaskCUDD.size mgr (createZddFromBdd mgr lawb) : loop (i+1) startKns lawb
    | i < n  = MyHaskCUDD.size mgr (createZddFromBdd mgr $ MyHaskCUDD.con mgr current_law updateDc) : loop (i+1) startKns (MyHaskCUDD.con mgr current_law updateDc)--
    | i == n = []
    | otherwise = error "something went wrong with loop"
    where
      updateDc =  S5_CUDD.bddOf startKns (genDcReveal n i)
      --finalCheck = S5_CUDD.bddOf startKns (Conj [ genDcEveryoneKnowsWhetherNSApaid n, genDcNobodyknowsWhoPaid n ])
  loop _ _ _ = error "initial structure should be normal BDD"
```

The initial knowledge structure now uses the `genXor` function (see below), similar to our $\oplus!$ operator (as in Definition 2.36), to generate a formula for the number of payers possible, e.g. when we say there are three payers the state law says that 3 and only 3 variables should be positive for all variables representing n payers in all branches of the decision diagram. The CUDD knowledge structure can be converted to facilitate all decision diagram types by using the conversion functions mentioned in the previous subsection.

```haskell
genDcKnsInitCudd :: Int -> Int -> IO S5_CUDD.KnowStruct
genDcKnsInitCudd n m = makeManager >>= \mgr -> do
  let sharedbitLabels = [ [k,l] | k <- [1..n], l <- [1..n], k<l ] -- n(n-1)/2 shared bits
  let sharedbitRel = zip sharedbitLabels [ (P $ n+1) .. ]
  let sharedbits =  map snd sharedbitRel
  let props = [ P 0 ] -- The NSA paid
           ++ [ (P 1) .. (P n) ] -- agent i paid
           ++ sharedbits
  let law = S5_CUDD.boolBddOf mgr $ genXorM n m
  let obsfor i =  P i : map snd (filter (\(label,_) -> i `elem` label) sharedbitRel)
  let obs = [ (show i, obsfor i) | i<-[1..n] ]
  return $ S5_CUDD.KnS mgr props law obs

genDcKnsInitCac :: Int -> Int -> S5_CAC.KnowStruct
genDcKnsInitCac n m = S5_CAC.KnS props law obs where
  props = [ P 0 ] -- The NSA paid
    ++ [ (P 1) .. (P n) ] -- agent i paid
    ++ sharedbits
  law = S5_CAC.boolBddOf $ genXorM n m
  obs = [ (show i, obsfor i) | i<-[1..n] ]
  sharedbitLabels = [ [k,l] | k <- [1..n], l <- [1..n], k<l ] -- n(n-1)/2 shared bits
  sharedbitRel = zip sharedbitLabels [ (P $ n+1) .. ]
  sharedbits =  map snd sharedbitRel
  obsfor i =  P i : map snd (filter (\(label,_) -> i `elem` label) sharedbitRel)
```

We use the following functions for producing the formulas describing the initial state and updates as in Subsection 3.1.2.

```haskell
genDcEveryoneKnowsWhetherNSApaid :: Int -> Form
genDcEveryoneKnowsWhetherNSApaid n = Conj [ Kw (show i) (PrpF $ P 0) | i <- [1..n] ]

-- XOR between shared secret bits for agent i
genDcReveal :: Int -> Int -> Form
genDcReveal n i = Xor (map PrpF ps) where
  (S5_CAC.KnS _ _ obs) = genDcKnsInit n
  (Just ps)     = lookup (show i) obs

genDcNobodyknowsWhoPaid :: Int -> Form
genDcNobodyknowsWhoPaid n =
  Conj [ Impl (PrpF (P i)) (Conj [Neg $ K (show k) (PrpF $ P i) | k <- delete i [1..n] ]) | i <- [1..n] ]

genDcCheckForm :: Int -> Form
genDcCheckForm n =
  pubAnnounceWhetherStack [ genDcReveal n i | i<-[1..n] ] $
    Conj [ genDcEveryoneKnowsWhetherNSApaid n, genDcNobodyknowsWhoPaid n ]

--Creates a disjoint form where each term has n propositions in conjunction of which m are positive and n-m are negative. All terms add up to all possible combinations of n choose m, giving a general XOR where m propositions have to be positive.
genXorM :: Int -> Int -> Form
genXorM n m = Disj
  [ Conj
    ([PrpF (P p) | p <- x]
      ++ [Neg (PrpF (P i)) | i <- y])
  | x <- select
  , let y = [0 .. n] \\ x
  ]
  where
    select = combinations m [0 .. n]
```

1-parameter Sum and Product

The additions discussed here are in the file `sizeExperiments/sumandproduct.hs`.

Currently we only have the maximum sum of the two numbers as parameter for the sum and product, but the original code has a fixed maximum of 100, thus we still need to adjust the code.

```haskell
genSapWhereWithSizeCac :: Int -> ([State], [Int])
genSapWhereWithSizeCac n = (S5_CAC.whereViaBdd (genSapKnStruct n) (genSapProtocol n), getSizePerUpdate)
↪  where
  info (S5_CAC.KnS _ lawb _) = S5_CAC.size lawb
  getSizePerUpdate = map info
    [ genSapKnStruct n
    , genSapKnStruct n `update` genSapForm1 n
    , genSapKnStruct n `update` genSapForm1 n `update` genSapForm2 n
    , genSapKnStruct n `update` genSapForm1 n `update` genSapForm2 n `update` genSapForm3 n ]

infoCudd :: S5_CUDD.KnowStruct -> Int
infoCudd (S5_CUDD.KnS mgr _ lawb _) = MyHaskCUDD.size mgr lawb
infoCudd (S5_CUDD.KnSZ mgr _ lawz _) = MyHaskCUDD.size mgr lawz
infoCudd (S5_CUDD.KnSZf0 mgr _ lawz _) = MyHaskCUDD.size mgr lawz
infoCudd (S5_CUDD.KnSZs0 mgr _ lawz _) = MyHaskCUDD.size mgr lawz
infoCudd (S5_CUDD.KnSZf0s0 mgr _ lawz _) = MyHaskCUDD.size mgr lawz

genSapWhereWithSizeCudd :: (S5_CUDD.KnowStruct -> S5_CUDD.KnowStruct) -> Int -> IO ([S5_CUDD.KnState],
↪  [Int])
genSapWhereWithSizeCudd convertFunc n = do
  start <- genSapKnStructCudd n -- also creates manager!
  let resultsPerUpdate = map infoCudd
        [ convertFunc start
        , convertFunc start `S5_CUDD.unsafeUpdate` genSapForm1 n
        , convertFunc start `S5_CUDD.unsafeUpdate` genSapForm1 n `S5_CUDD.unsafeUpdate` genSapForm2 n
        , convertFunc start `S5_CUDD.unsafeUpdate` genSapForm1 n `S5_CUDD.unsafeUpdate` genSapForm2 n
        ↪  `S5_CUDD.unsafeUpdate` genSapForm3 n ]
  let checkAnswer = S5_CUDD.toKns $ convertFunc start `S5_CUDD.unsafeUpdate` genSapForm1 n
  ↪  `S5_CUDD.unsafeUpdate` genSapForm2 n `S5_CUDD.unsafeUpdate` genSapForm3 n
  return (S5_CUDD.whereViaBdd checkAnswer Top, resultsPerUpdate)
```

Only BDDs are used for checking the answer as the `whereViaBDD` function uses `allSat`, a wrapper for a CUDD function only available for BDDs. This does not matter as we can still gather the size after each update and are not particularly interested in the boolean result as long as it is correct.

```haskell
whereViaBdd :: KnowStruct -> Form -> [KnState]
whereViaBdd kns@(KnS props lawbdd _) f =
 map (sort . map (toEnum . fst) . filter snd) $
   allSatsWith (map fromEnum props) $ con lawbdd (bddOf kns f)
whereViaBdd _ _ = error "whereViaBdd with wrong kns type"
```

```haskell
-- | Get all satisfying assignments, inspired by the CacBDD version. These will be partial, i.e. only
-- contain (a subset of) the variables that actually occur in the BDD.
allSats :: Dd B -> [Assignment]
allSats (ToDd b) = concatMap bitsToAss (Cudd.Cudd.cuddAllSat manager b)

-- | Get the lexicographically smallest satisfying assignment, if there is any.
anySat :: Dd B -> Maybe Assignment
anySat (ToDd b) = fmap (head . bitsToAss) (Cudd.Cudd.cuddOneSat manager b)

-- | Get all complete assignments, given a set of all variables.
-- In particular this will include variables not in the BDD.
allSatsWith :: [Int] -> Dd B -> [Assignment]
allSatsWith allvars b = concatMap (completeAss allvars) (allSats b)
```

Initialisation of the starting knowledge structure closely resembles the original code.

```haskell
-- possible pairs 1<x<y, x+y<=n, Alica knows the sum of the 2 numbers, Bob knows the product.
-- for all possible pairs of numbers imply the product and sum,
-- which can be written as a disjuction of conjunction terms, consisting of x, y, s and p.
genSapKnStruct :: Int -> KnowStruct
genSapKnStruct n = KnS (genSapAllProps n) law obs where
  law = boolBddOf $ Disj [ Conj [ genxyAre (x,y) n, genSIs (x+y) n, genPIs (x*y) n ] | (x,y) <-
  ↪  genPairs n ]
  obs = [ (alice, genSProps n), (bob, genPProps n) ]
```

```haskell

genSapKnStructCudd :: Int -> IO S5_CUDD.KnowStruct
genSapKnStructCudd n = do
  mgr <- MyHaskCUDD.makeManager
  let law = S5_CUDD.boolBddOf mgr $ Disj [ Conj [ genxyAre (x,y) n, genSIs (x+y) n, genPIs (x*y) n ] |
  ↪  (x,y) <- genPairs n ]
  let obs = [ (alice, genSProps n), (bob, genPProps n) ]
  return $ S5_CUDD.KnS mgr (genSapAllProps n) law obs
```

The only change is the parameter for the maximum value a sum can have, thus now the variables are initialised depending on the given parameter value. The maximum number of variables needed to represent a range of numbers in binary representation, from 0 to n, can be calculated by taking the ceiling of the log base 2 of n. The maximum range needed for representing all possible product numbers is $\lceil (n/2)^2 \rceil$

```haskell
-- from max sum to max product
maxProduct :: Int -> Int
maxProduct n = ceiling ((fromIntegral n / 2) * (fromIntegral n / 2))

--from max to binary representation max
maxBinary :: Int -> Int
maxBinary = ceiling . logBase 2.0 . fromIntegral

-- possible pairs 1<x<y, x+y<=n
genPairs :: Int -> [(Int, Int)]
genPairs n = [(x,y) | x<-[2..100], y<-[2..100], x<y, x+y<=n]

-- e.g. 7 propositions to label [2..100], because 2^6 = 64 < 100 < 128 = 2^7
genXProps, genYProps, genSProps, genPProps :: Int -> [Prp]
genXProps n = [(P  1)..(P $ maxBinary n)]
genYProps n = [(P $ maxBinary n + 1)..(P $ maxBinary n * 2)]
genSProps n = [(P $ maxBinary n * 2 + 1)..(P $ maxBinary n * 3)]
-- then 12 propositions for the product, because 2^11 = 2048 < 2500 < 4096 = 2^12
genPProps n = [(P $ maxBinary n * 3)..(P $ maxBinary n * 3 + (maxBinary $ maxProduct n))]

genSapAllProps :: Int -> [Prp]
genSapAllProps n = sort $ genXProps n ++ genYProps n ++ genSProps n ++ genPProps n

genXIs, genYIs, genSIs, genPIs :: Int -> Int -> Form
genXIs n m = booloutofForm (powerset (genXProps m) !! n) (genXProps m)
genYIs n m = booloutofForm (powerset (genYProps m) !! n) (genYProps m)
genSIs n m = booloutofForm (powerset (genSProps m) !! n) (genSProps m)
genPIs n m = booloutofForm (powerset (genPProps m) !! n) (genPProps m)
```

The following functions show how logic formulas are produced for the initial knowledge structure and updates.

```haskell
genxyAre :: (Int,Int) -> Int ->  Form
genxyAre (n,m) o = Conj [ genXIs n o, genYIs m o ]

genSapKnows :: Agent -> Int -> Form
genSapKnows i n = Disj [ K i (genxyAre p n) | p <- genPairs n]

genSapForm1, genSapForm2, genSapForm3 :: Int -> Form
genSapForm1 n = K alice $ Neg (genSapKnows bob n) -- Sum: I knew that you didn't know the numbers.
genSapForm2 = genSapKnows bob  -- Product: Now I know the two numbers
genSapForm3 = genSapKnows alice -- Sum: Now I know the two numbers too
```

1-parameter Higher Order Sally-Ann

The parameter n gives the number of additional hiding spots. Further we implemented a function for the initial belief structure and the transformer as defined in Subsection 3.1.2.

```haskell
--- higher order Sally Anne version

hidingSpots :: Int -> [Prp]
hidingSpots n = map P [0 .. n]

initHigherOrderSA :: Int -> IO K_CUDD.BelScene
initHigherOrderSA n = do
  mgr <- HC.makeManager
  let law    = S5_CUDD.boolBddOf mgr $ Conj $ PrpF (P 0) : [Neg (PrpF $ P x) | x <- [1 .. n]]
  -- it is common knowledge where the chocolate is placed
  let obs    = fromList $ [ ("child " ++ show i, K_CUDD.totalRelBdd mgr) | i<-[0..n] ]
  let actual = [P 0]
  return (K_CUDD.BlS mgr (hidingSpots n) law obs, actual)

iSecretlySeesTheMovingChocolate :: K_CUDD.BelScene -> Int -> Int -> K_CUDD.Event
iSecretlySeesTheMovingChocolate (K_CUDD.BlS mgr props _ _, _) n i = (K_CUDD.Trf mgr addprops addlaw (fromList changedPropsLaws) (fromList agentBeliefs), knstate) where
  addprops = [a,b]
  -- (b ^ a) := child i beliefs that no-one sees her/him secretly observing child (i+1) moving the chocolate,
  -- (b ^ neg a) := children > i believe they are the only ones that know that child (i+1) has moved the chocolate,
  -- (neg b ^ neg a) := children < i are not witnessing the event,
  -- (neg b ^ a) := not possible.
  -- note that the information of who moved the chocolate is not modeled/stored.
  addlaw = Neg $ Conj [ PrpF a, Neg $ PrpF b] --`debug` (show a ++ show b) --(neg (neg b ^ a))
  changedPropsLaws = [(P i, S5_CUDD.boolBddOf mgr (Conj [ Impl (Neg $ PrpF b) (PrpF $ P i), Impl (PrpF b) Bot])), (P (i+1), S5_CUDD.boolBddOf mgr (Conj [ Impl (Neg $ PrpF b) (PrpF $ P (i+1)), Impl (PrpF b) Top])) ]
  -- moving the chocolate from position i-1 to i for all states where b has happened
  agentBeliefs =
    [("child " ++ show i, relBDD mgr (Disj [Conj [PrpF aOut, PrpF bOut, PrpF aIn, PrpF bIn], Conj [Neg $ PrpF aOut, PrpF bOut, Neg $ PrpF aIn, Neg $ PrpF bIn], Conj [Neg $ PrpF aOut,Neg $ PrpF bOut,Neg $ PrpF aIn,Neg $ PrpF bIn]]) )]  ++ -- Child i believing (b ^ a)
     (if i < n then [("child " ++ show after, relBDD mgr (Disj [Conj [PrpF aOut, PrpF bOut, Neg $ PrpF aIn, PrpF bIn], Conj [Neg $ PrpF aOut, PrpF bOut, Neg $ PrpF aIn, PrpF bIn], Conj [Neg $ PrpF aOut, Neg $ PrpF bOut, Neg $ PrpF aIn, Neg $ PrpF bIn]] )) | after <-[(i+1)..n]] else []) ++ -- Children after i believing (neg a ^ b)
     (if i > 0 then [ ("child " ++ show before, relBDD mgr (Disj [Conj [PrpF aOut, PrpF bOut, Neg $ PrpF aIn, Neg $ PrpF bIn], Conj [Neg $ PrpF aOut, Neg $ PrpF bOut, Neg $ PrpF aIn, Neg $ PrpF bIn], Conj [Neg $ PrpF aOut,PrpF bOut,Neg $ PrpF aIn,Neg $ PrpF bIn]])) | before <-[0..(i-1)]] else []) -- all other childs before i are not aware of the event happening and thus still believe (neg a ^ neg b)
  knstate = [a,b] -- in actuality the atomic event (a^b) has taken place

  a = freshp props
  b = freshp $ props ++ [a]
  aOut = K_CUDD.mvP a -- i rename mvp and cpp to outward and incoming ..
  aIn = K_CUDD.cpP a -- .. purely for my own understanding
  bOut = K_CUDD.mvP b
  bIn = K_CUDD.cpP b
almostSecretMovingChocolate _ _ _ = error "other bls types not supported"

relBDD :: HC.Manager -> Form -> K_CUDD.RelBDD
relBDD mgr f = pure (S5_CUDD.boolBddOf mgr f) :: K_CUDD.RelBDD
```

4.2 Random boolean function sizes and proportional XOR-domain advantage

Before running the experiments I had the idea to make a brief exploratory analysis on what size we can expect our BDDs and ZDDs to become and so determine the expected proportional advantage for the XOR-pattern in ZDD representation. For this we could look at the expected size of random boolean functions and/or the worst-case size, however, this is not fair as our logic puzzles (and most models) are highly structured and thus are comparatively much smaller. Since the analysis, although useless with respect to our experiments and conclusions, does contain interesting content we have decided to include it here in the Appendix.

Worst-case sizes

The maximum size of the last row in an unreduced decision diagram is the size of the powerset of the context, $2^{|V|}$ in our case. Including all nodes of the rows beforehand gives us $2^{2^{|V|}} + 2^{|V|}$, where the second term counts the leaf nodes. But as the merge rule is applied (some symmetry is unavoidable) with elimination rules on top of that, BDDs and ZDDs have fewe nodes. The literature [NV19] [Weg94] [LL92] tells us that the maximum/worst size of a BDD can be calculated by:

$$\boldsymbol{max\_size}(\boldsymbol{BDD}(\varphi)) = 2 + \textstyle\sum_{i=1}^{n} \boldsymbol{min}\{2^{i-1}, 2^{2^{n-i+1}} - 2^{2^{n-i}}\}$$

where n is the size of the context, in our case $|V|$.

To our knowledge there is no literature on the precise maximum size for ZDDs, mainly because it will not differ much from BDDs since by definition it would contain as little eliminable nodes as possible.

Random boolean function sizes

It is known that, for *most* given values of context size, n, almost all random boolean functions are equal to the worst size (usually described as the strong Shannon effect), while for the leftover *few* values of n almost all functions have the same BDD size but with lower than worst size (coined as weak Shannon effect) [Weg94] [GPS98]; The fraction of boolean functions whose reduced BDD size with respect to the variable ordering $x_1, ..., x_n$ differs more than $O(2^{2n/3})$ from $\boldsymbol{S}(n)$ is bounded by $O(2^{-n/3})$, where O indicates the big O notation and $\boldsymbol{S}(n)$ is the expected SDD (or QROBDD as named in other literature) size for a random boolean function with context of size n:

$$\boldsymbol{S}(n) := \textstyle\sum_{0 \leq i \leq n-1} 2^{2^{n-i}}(1 - (1 - 2^{-2^{n-i}})^{2i})$$

Proportional advantage of an XOR-domain

To find the bounds of the ZDD advantage for an XOR-domain, we use the equation from Definition 2.37, but replace our variable for $|\mathcal{D}|$ with k to avoid multiple definitions of n in this subsection:

$$(k-1) \quad \leq \quad \boldsymbol{size}(\boldsymbol{BDD}^{\mathcal{D}\oplus !}(\varphi)) - \boldsymbol{size}(\boldsymbol{ZDD}^{\mathcal{D}\oplus !}(\varphi)) \quad \leq \quad c \cdot (\tbinom{k+2}{m+1} - \tbinom{k}{m} - \tbinom{k+1}{m})$$

Comparing our tasks to expected size of random boolean functions is not fair; there is a strong structure present in our problems as we make use of few update rules, even as we up-scale the problems, thus we can expect the graph sizes for our belief structures to be smaller. Furthermore, the libraries used for representing the graphs can have complement edges implemented, such that the graphs are reduced even more. This means that the proportional XOR-domain advantage is much higher for our experiments than if we were to calculate it when assuming random state and belief laws. Still, if we would want to calculate this advantage we could do it

In knowledge structures (for S5) θ is the only present boolean function, where $n = |V|$. The belief structure (for K) with belief indexing has two boolean functions, θ and Ω, where Ω's $n = |V \cup V'| + |agents|$. If we place the domain on top of the graph, then $c = 1$ in our equation for the XOR domain advantage. Knowing n, k, m and c, we can consider a smaller, thus more accurate, upper bound on boolean functions that contain an XOR domain and otherwise are worst case size:

$$\text{for BDDs: } (\tbinom{k+2}{m+1} - 1 - \tbinom{k}{m}) + 2 + \textstyle\sum_{i=k}^{n} \boldsymbol{min}\{\tbinom{k}{m} \cdot 2^{i-k-1}, 2^{2^{n-i+1}} - 2^{2^{n-i}} m\}.$$
$$\text{for ZDDs: } (\tbinom{k+1}{m} - 1) + 2 + \textstyle\sum_{i=k}^{n} \boldsymbol{min}\{\tbinom{k}{m} \cdot 2^{i-k-1}, 2^{2^{n-i+1}} - 2^{2^{n-i}} m\}.$$

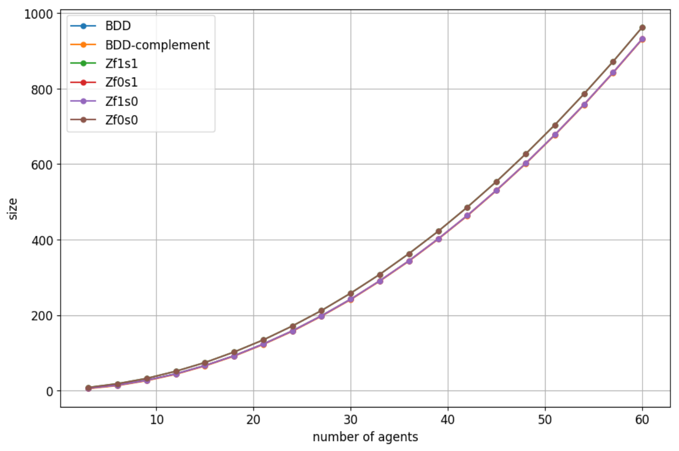
where the first term describes the size of the XOR-domain, the "+2" in middle represent the leaf nodes, and the last term describes the worst case subgraph, starting at the end from XOR-domain for $\binom{k}{m}$ different copies. The number of copies does not have an influence on on which level the merge rule starts being applied, thus only the first argument of the $\boldsymbol{min}$ is multiplied with $\binom{k}{m}$. Keep in mind that this is a very rough estimation and therefore not included in the main body of the thesis.

4.3 Additional results

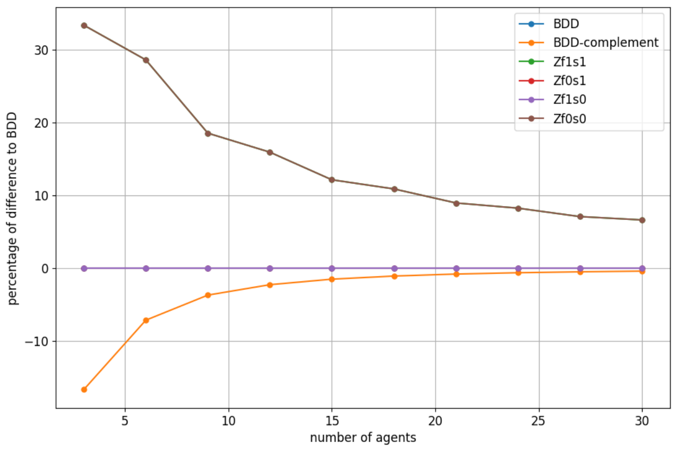
Here we provide the plots for the worst state law sizes that we left out of our main results as they roughly show the same patterns as the average state law sizes. Also the larger sized 3D plots and some additional second parameter investigation plots can be found here.

4.3.1 Muddy children

Worst state law sizes

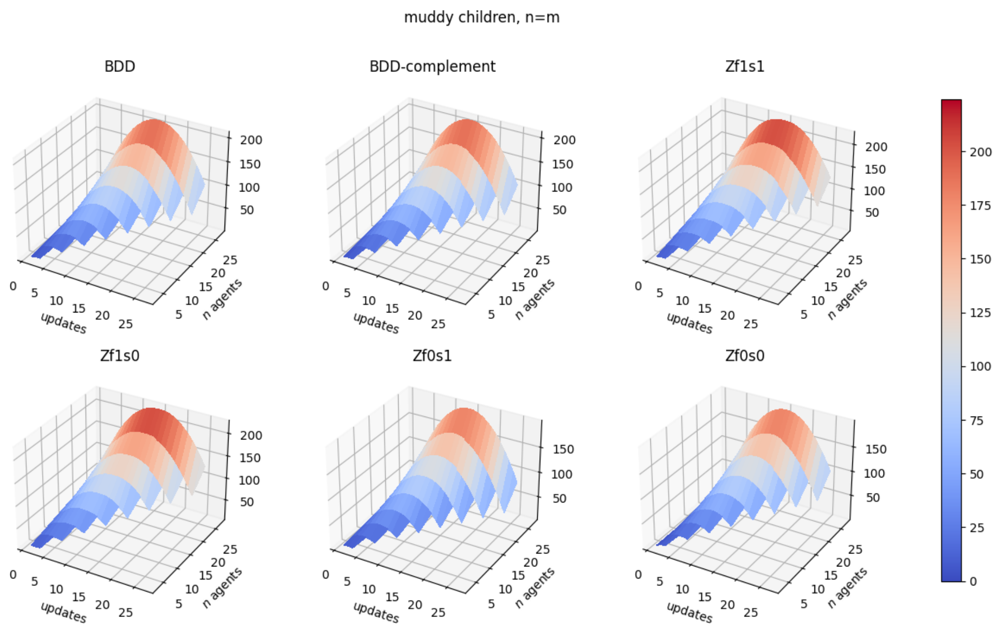


Worst state law sizes per nr. of children, where nr. of muddy = nr. of children. The BDD, BDD-complement, f_1s_0 and f_0s_1 lines are overlapping, and the f_1s_1 and f_0s_0 lines are overlapping.

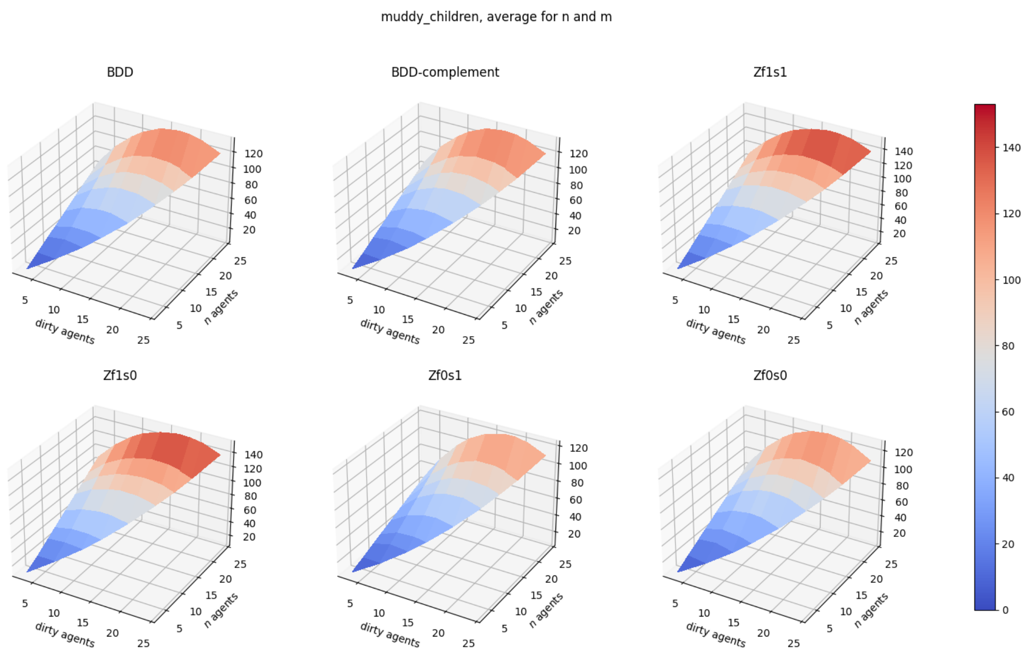


Difference of worst state law sizes in percentage w.r.t. the BDD representation, per nr. of children, where nr. of muddy = nr. of children. The BDD-complement, f_1s_0 and f_0s_1 lines are overlapping, and the f_1s_1 and f_0s_0 lines are overlapping.

3D plots



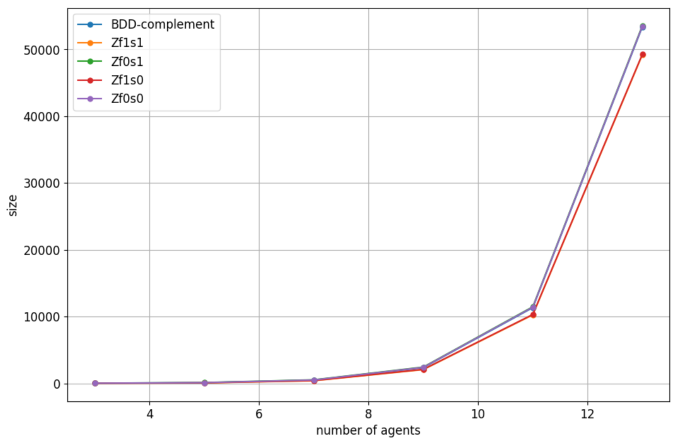
Dining cryptographers state law size per update per nr. of agents, where nr. of dirty = nr. of agents.



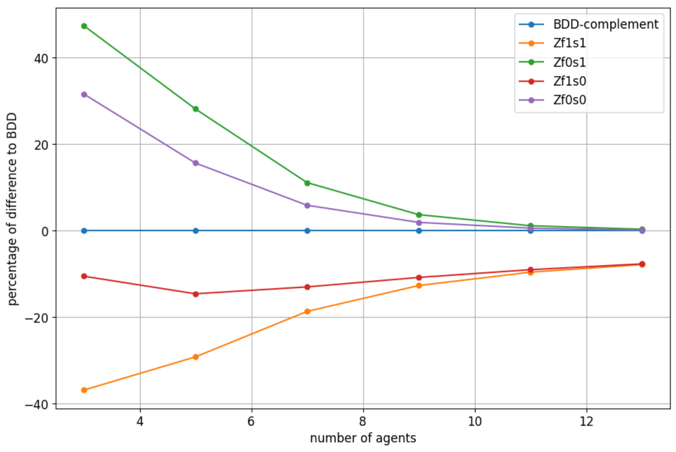
Muddy children average state law sizes per nr. of dirty, per nr. of agents.

4.3.2 Dining cryptographers

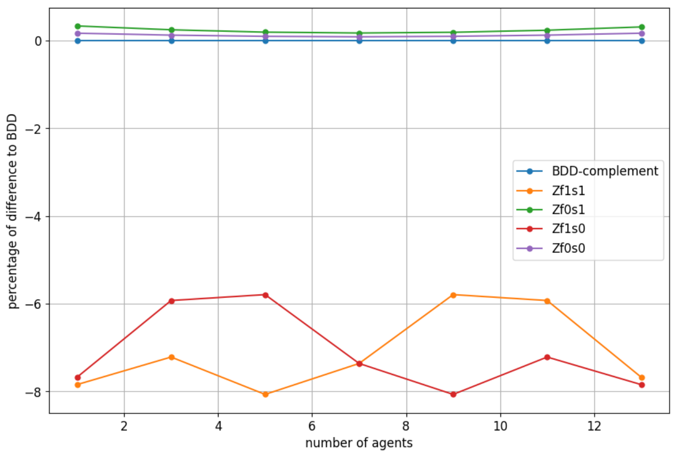
Worst state law sizes



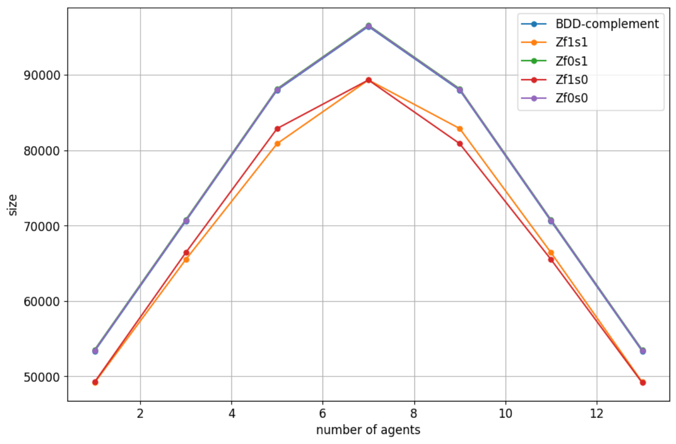
Dining cryptographers average state law sizes, per nr. of diners, where nr. of payers = 1. The BDD-complement, f_0s_1 and f_0s_0 lines overlap, and the f_1s_1 and f_0s_0 lines overlap.



Dining cryptographers average state law sizes, per nr. of diners, where nr. of payers = 1, w.r.t. the BDD-complement representation.

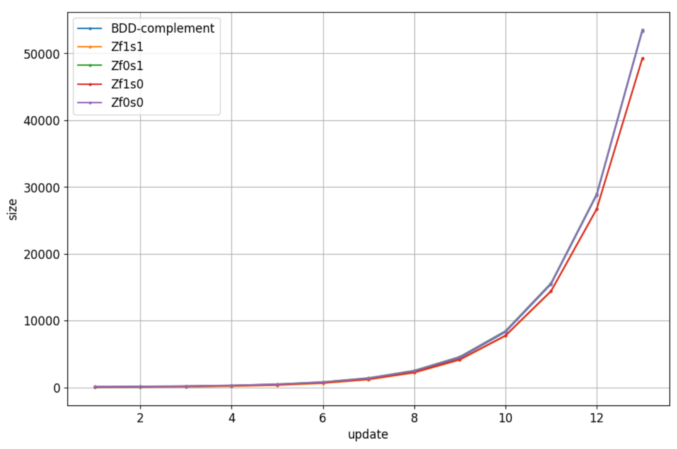


Dining cryptographers average state law sizes per nr. of payers, where nr. of diners = 13.

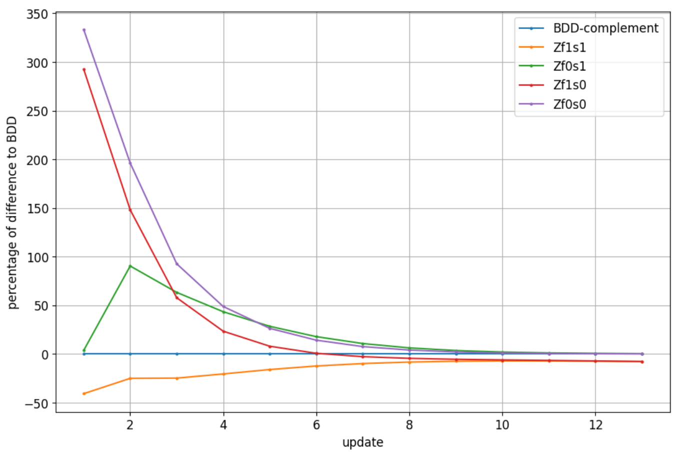


Dining cryptographers average state law sizes per nr. of payers, w.r.t. the BDD-complement sizes, where nr. of diners = 13.

Per update



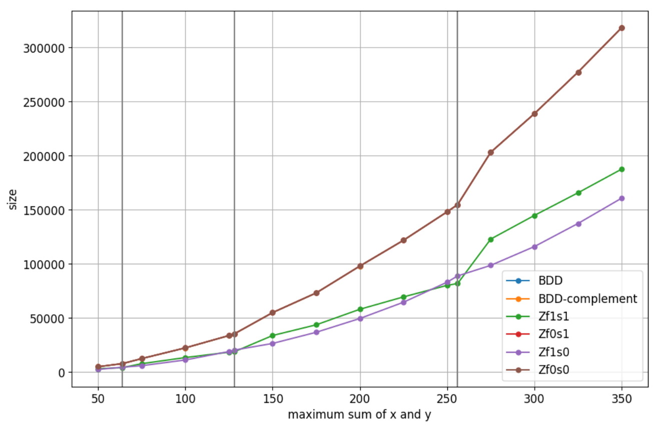
Dining cryptographers state law sizes per update, where nr. of diners = 13 and nr. of payers = 1. The BDD-complement, f_0s_1 and f_0s_0 lines overlap, the f_1s_1 and f_0s_0 lines overlap.



Dining cryptographers state law sizes per update, where nr. of diners = 13 and nr. of payers = 1.

4.3.3 Sum and Product

Worst state law sizes



Worst state law per maximum sum of the hidden numbers (x and y).

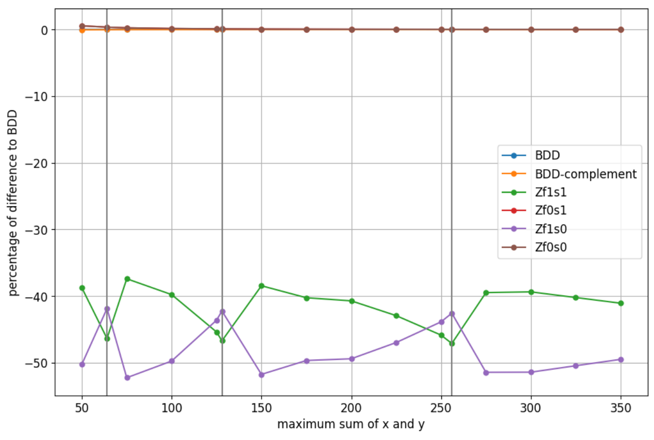
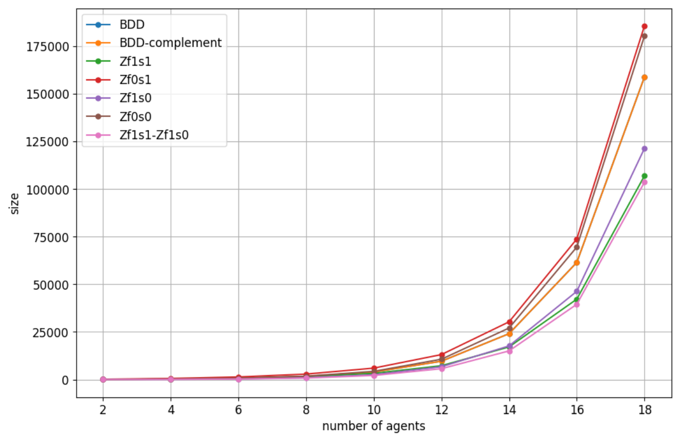


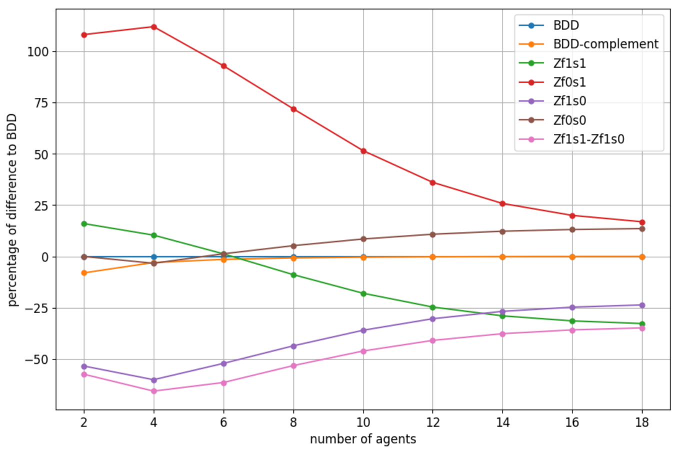
***Figure 3.4**: Difference of worst state law sizes in percentage per maximum sum of the hidden numbers (x and y), w.r.t. the BDD representation.*

4.3.4 Higher Order Sally-Anne

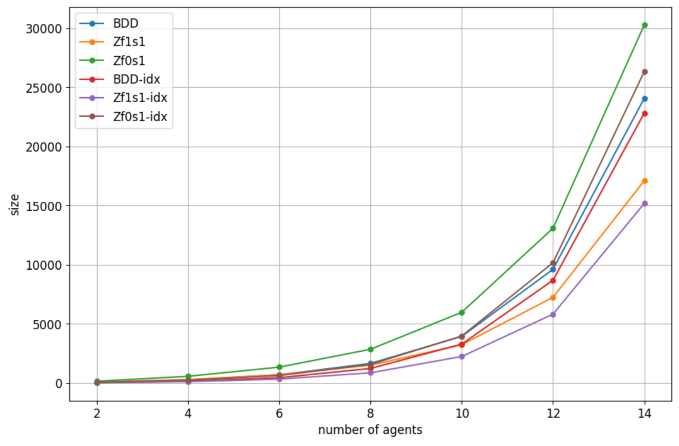
Worst state sum of laws



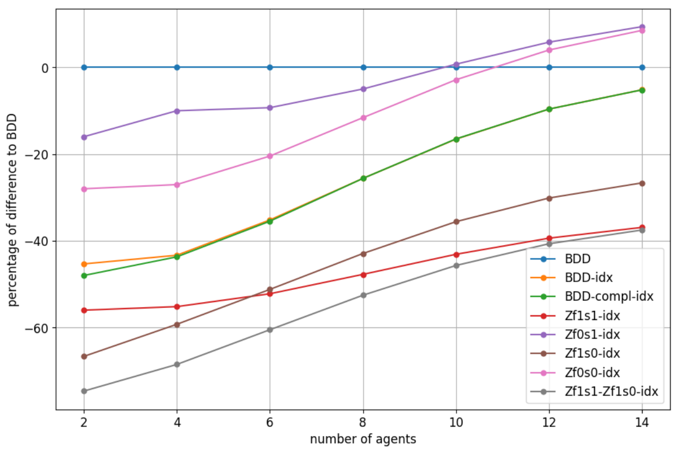
Worst state sum of laws (belief laws plus state law) sizes per nr. of agents, where the less interesting f_1s_0, f_0s_0 and complement-BDD sizes are left out to reduce clutter.



Difference of worst state sum of laws (belief laws plus state law) sizes in percentage w.r.t. the BDD size, per nr. of agents.

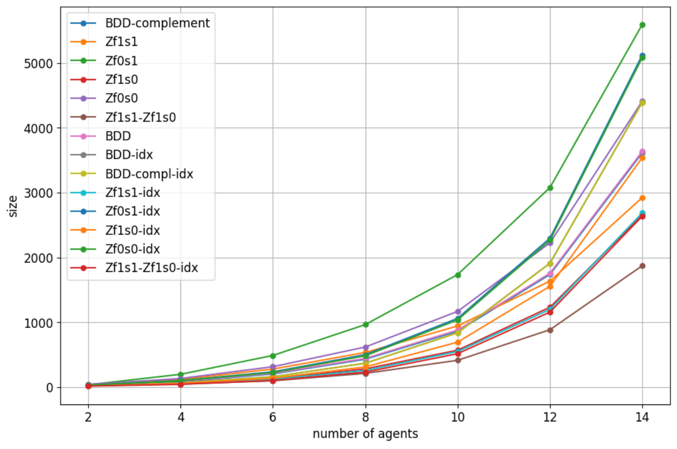


Worst state sum of laws (indexed belief laws plus state law) sizes per nr. of agents, where the less interesting f_1s_0, f_0s_0 and complement-BDD sizes are left out to reduce clutter.



Difference of worst state sum of laws (indexed belief laws plus state law) sizes in percentage, per nr. of agents, w.r.t. the not-indexed BDD size.

All lines included



Average sum of laws (belief laws plus state law) sizes per nr. of agents.