Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Controlling SAT/SMT solvers with decision diagrams to support abstraction-based model checking

Scientific Students' Association Report

Author:

Nóra Almási

Advisor:

dr.Vince Molnár

2020

Contents

Kivonat

Kritikus rendszerek tervezésekor a helyes működés kulcsfontosságú: az esetleges tervezési hibák ilyenkor hatalmas anyagi kárt, vagy akár személyi sérüléssel járól baleseteket is okozhatnak. A hibamentesség a hagyományos tesztelési módszerekkel nem bizonyítható, ennek belátásához (vagy legalábbis közelítéséhez) a lehetséges viselkedések kimerítő vizsgálata szükséges. Ehhez adnak támogatást a különböző formális verifikációs módszerek, ahol a rendszereknek megfeleltetett tervezési modellek helyessége a matematikai precizitással belátható.

Az egyik ilyen módszer, a logikaalapú szimbolikus modellellenőrzés vezető technikája az ellenpélda-vezérelt absztrakció finomítás (Counterexample-Guided Abstraction Refinement, CEGAR). A módszer egyre finomabb absztrakciókon iterálva vizsgálja a modellt, ezzel elkerülve az irreleváns részek vizsgálatát. Predikátumabsztrakció használatakor az állapottér-felderítés során megoldandó az AllSAT-probléma, ahol a feladat egy adott logikai kifejezésre az összes őt igazzá tevő kiértékelés megtalálása. Ennek megoldására már állnak rendelkezésre AllSAT-megoldók, a kutatás célja az eddigieknél rugalmasabb, az alkalmazási terület igényei szerint testreszabható megoldó tervezése, amit sikerrel lehetne alkalmazni a különböző modellellenőrzési módszereken belül is.

Dolgozatomban egy olyan megoldást javaslok, ahol az AllSAT megoldóknál sokkal kiforrottabb és szélesebb körben elérhető SAT/SMT megoldók felhasználásával a megoldásokat döntési diagram strukturába szervezem, és ez a struktúra nem csak a kompakt tárolást, hanem a megoldások lekérdezésének vezérlését is végzi. Várható, hogy a megoldás a modellellenőrzőkben jelenleg használt módszerhez képest versenyképes lesz, emellett a funkciókör bővítése szélesebb körű alkalmazhatósághoz vezethet. Fontos kiterjesztés, hogy a módszer nem csak SAT, de SMT megoldókkal is működik, tehát a változók nem csak bináris értékűek lehetnek. Dolgozatomban kitérek az emiatt felmerülő új kihívásokra, és ezekre a különböző felhasználási területek igényeihez igazodó megoldásokat javaslok. A módszert implementálom egy konfigurálható modellellenőrző keretrendszerben, integrálom már létező algoritmusokkal, és teljesítményét mérésekkel vizsgálom.

Abstract

When designing critical systems, correct operation is crucial: design errors may lead to serious economical consequences or even accidents with personal injuries. Freedom from errors cannot be proved with traditional testing methods; to show (or at least approximate) correctness, an exhaustive examination of all possible behaviors is required. Various formal verification techniques facilitate this process, where the correctness of the design models corresponding to the systems can be demonstrated with mathematical precision.

One such technique is logic-based symbolic model checking, where the leading approach is called Counterexample-Guided Abstraction Refinement (CEGAR). The approach analyzes the model through a series of increasingly finer abstractions, thus avoiding the examination of irrelevant details. When using predicate abstraction, one of the arising problems which should be solved during the state space search is the AllSAT problem, where the task is to find all the evaluations that satisfy a given logical expression. There are available AllSAT solvers to tackle this problem. The aim of this research is to create a more flexible solution that can be customized according to the needs of the application area, so that it can be successfully applied in different model checking methods.

In my work I propose a solution using SAT and SMT solvers, which are generally more advanced and available than AllSAT solvers. I store the solutions in a decision diagram structure, which not only offers compact storage, but also controls the enumeration of solutions. We expect this approach to be competitive compared to the current practices used in model checking, and the extended functionality compared to traditional AllSAT solvers may lead to wider applicability. A major improvement is that our method works not only with SAT but also with SMT solvers, which means that variables are allowed to have non-binary discrete domains. In my report I discuss the challenges coming from this extension of the variable domain, and propose solutions adjusted to the needs of different application areas. I implement the new method in a configurable model checking framework, integrate it with existing algorithms, and examine its performance with measurements.

List of Abbreviations

Abbreviation	Description
ARG	Abstract Reachability Graph
BDD	Binary Decision Diagram
CFA	Control Flow Automata
CEGAR	Counterexample-Guided Abstraction Refinement
DFS	Depth-First Search
MDD	Multi-valued Decision Diagram
SAT	Boolean Satisfiability Problem
SMT	Satisfiability Modulo Theories
XSTS	Extended Symbolic Transition System

Chapter 1

Introduction

When designing safety-critical systems, there is an increasing demand for ensuring functional correctness. The main cause is that system failures may have far-reaching consequences like large economic losses or even threatening human lives. But how can we prove the absence of errors?

The simplest way to find faults is testing, where a failing test also shows the incorrect behaviour. However, the lack of errors cannot be shown this way. A system functions properly if and only if all of the potential behaviours are problem-free. Unfortunately an exhaustive examination is not easy. With the growing complexity of the tested system, the number of possible behaviours that should be checked increases exponentially. The phenomenon is known as the state space explosion problem. Due to the exponential nature, checking all possible behaviours one by one cannot be efficient.

Formal verification deals with proving correctness in a mathematically precise way. Model checking is a technique within formal verification, which answers the question of correctness by checking the possible states' state space exhaustively. In the case of symbolic transition systems, the model's states and transitions are described with mathematical (logic) formulae. Thus, the question of reachability can be transformed into the Boolean satisfiability problem (SAT), i.e. the problem of determining if a satisfying interpretation for the given Boolean formula exists. A more general form of the problem is also present in model checking – the AllSAT problem deals with determining all satisfying assignments for a given logical formula. The main focus of our research is this problem, for which we propose a novel approach that combines different, well-proven solutions used in model checking, designed to deal with many specific challenges of this application domain (unlike general AllSAT solvers).

There are quite advanced heuristic algorithms available for the solution of the SAT problem. There are available SAT solvers too, which decide the satisfiability of the given formula efficiently, and return a proper variable assignment if there is any. A naive but widely used approach for the AllSAT problem is to use multiple calls to SAT solvers. The satisfying interpretations can be obtained one by one by negating and concatenating the calculated result to the end of the expression and asking the SAT solver for a satisfying assignment again. However, in practice, this approach is often not scalable because the expression can grow too large to handle when there are too many solutions.

The proposed approach solves this problem by using a decision diagram structure. Decision diagram is a popular data structure in the area of model checking, allowing a compact storage of (state) vectors. We introduce *substitution diagram*, which improves the original construction by annotating the diagram nodes with the additional information of the

logical expression. Therefore the identity of nodes are not determined by their child nodes, but by the assigned expression, which is known even *before* calculating its children. This approach provides a way to compute a decision diagram representation of the solution space directly.

Besides the storage size advantages, the approach can be tailored to a wide area of applications in model checking. We have identified four use cases, three of which have been implemented in the Theta Model Checking Framework [14], which is a generic, modular and configurable framework developed at the Critical Systems Research Group of the Budapest University of Technology and Economics. The correctness and efficiency of the realized solution are investigated with measurements.

This report is structured as follows. Chapter 2 provides basic knowledge about model checking, decision diagrams and the AllSAT problem. It also presents the targeted application areas and related approaches. Chapter 3 details the proposed idea, and chapter 4 presents the implementation details. Chapter 5 gives information about the measurements and benchmark results. Finally, Chapter 6 concludes the work.

Chapter 2

Preliminaries

This chapter summarizes the basic knowledge and theoretical background needed for understanding the research idea and the meaning of the results. Firstly, some details of the concept of model checking are given. We present symbolic transition systems and the connection between model correctness examination and mathematical logic. Afterwards, we present the SAT and AllSAT problems. Finally, we show application areas of these problems in model checking.

2.1 Model checking

Model checking [3] is a formal verification method used for testing properties of finite state systems. The technique analyzes all possible behaviours of a model automatically, investigating, whether it meets a given specification or not. Formally, the problem of model checking is deciding if for a given model M and a formal requirement φ, the following holds: $M \models \varphi$, i.e. the model is indeed a model of the requirement. There are various formalisms for formal models and requirements. This work assumes that models are given as Kripke structures (Def. 1), specified by symbolic transition systems.

2.1.1 Modeling formalisms

Formal models can be categorized into low-level or high-level formalisms. Low-level models describe the system with mathematical precision. They are usually built on mathematical constructs like sets, relations and functions. They are usually represented with labelled edges and nodes. A common example of low-level formalisms is a Kripke-structure. Low-level models are mostly easy to handle algorithmically, but they are not easy to handle for humans. High-level formalisms are easier to comprehend, as they are much closer to the modelled system. Common examples of higher-level formalisms are Petri-nets or statecharts. As this work focuses on algorithms on low-level formalisms, only Kripke structures and symbolic transition systems are explained here.

2.1.2 Kripke structure

Kripke structure[6] is an example of low-level modelling formalism. Its structure is a directed graph. The nodes represent states of the modelled systems, and edges symbolize

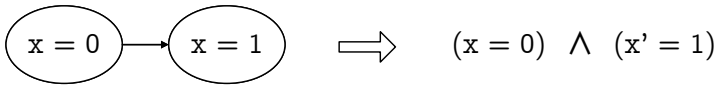

Figure 2.1: Encoding state transitions in symbolic transition systems

the state transitions. The states may be labelled. Labels denote properties, which hold for the current state. Thus, paths stand for the possible behaviours of the system.

Definition 1 (Kripke structure). Given a set of atomic propositions P a Kripke structure is a 4-tuple $M = (S, I, N, L)$, where:

- S is the set of states;
- $I \subseteq S$ is the set of initial states;
- $N \subseteq S \times S$ is the transition relation consisting of state pairs;
- $L : S \to 2^P$ is the labeling function mapping a set of atomic propositions to each state. ▪

2.1.3 Symbolic transition system

Symbolic transition systems describe Kripke structures implicitly by preserving state variables of a high-level model and using logical expressions to specify initial states and transitions. An example can be seen in Fig. 2.1.

Definition 2 (Symbolic transition system). Given a set of atomic variables V a symbolic transition system is a pair $STS = (\phi_I, \phi_N)$, where:

- ϕ_I is a Boolean mathematical expression over variables in V which evaluates to true iff the valuation describes an initial state of the system;
- ϕ_N is a Boolean mathematical expression over variables in V and successor variables in V' which evaluates to true iff the valuations of variables in V and V' describe source and target states of a transition, respectively. ▪

Computing if a specific valuation has a corresponding state in the Kripke structure defined by a symbolic transition system is called the reachability problem, which, in this case, can be mapped to the satisfiability of Boolean mathematical expressions.

2.2 Satisfiability problems

Satisfiability problems deal with the decision of whether there exists a valuation for which a given Boolean mathematical expression evaluates to true.

Depending on the variables used in the expression, we can speak about the SAT problem (for Boolean variables) or the SMT problem (for various other domains) [7]. It is interesting to note that a SAT problem always has a finite number of solutions (i.e. satisfying valuations), whereas SMT problems may have infinitely many solutions.

2.2.1 Calculating every solution

The problem of finding all solutions of a SAT problem is called the AllSAT problem. The same problem for SMT formulae is not well-defined because of the potentially infinite solutions, but we can formalize the problem for cases with finite solutions or in terms of bound k (i.e. we need at most k solutions) [9].

2.2.2 Solvers

The research of effective heuristic algorithms for the solution of SAT and SMT problems has a long history, therefore there are lots of available solver tools with high maturity. Most SAT/SMT solvers are capable of returning a satisfying valuation as proof of satisfiability. Most of the solvers are also equipped with functions to push and pop constraints as a way to solve multiple problems incrementally. This work will build on these functionalities to compute every solution using SAT/SMT solvers.

The number of available AllSAT solvers is much lower, while AllSMT solvers are almost non-existing [9]. This is the main motivation for building on SAT/SMT solvers instead.

2.3 Decision diagrams

The other crucial question by solving the AllSAT problem besides the algorithm speed is the limited storage space. Listing all the solutions one by one is often not possible when there is a large number of solutions. For this problem, the decision diagram structure offers a better alternative.

2.3.1 Binary decision tree

A binary decision tree is specified by a Boolean logic formula and a predefined variable sequence, the value substitution order. An example of the variable order a, b, c and the expression $(a \vee \neg b) \wedge (b \vee c)$ explains the concept in Fig. 2.2. It is a binary directed tree with labelled nodes and edges. Internal nodes are labelled with the corresponding variable (that is evaluated in that node). The edge labels show the substituted value. Terminal nodes (leaves) show the evaluation result after substituting the values along the paths leading to the node, i.e. *true* or *false*. The individual solutions are the individual paths from the root node to the terminal nodes containing a *true* value.

2.3.2 Binary decision diagram

There may be identical subtrees in a binary decision tree where we could save space if we merge them into one. This results in a binary decision diagram, carrying the same information in a more compact structure.

The resulting binary decision diagram after the node merges in the previous example can be seen in Fig. 2.3. When decision diagrams are discussed, *false* terminal nodes – and routes ending in *false* terminal nodes – are usually omitted. The reason is that only the *true*-evaluating substitutions are of importance.

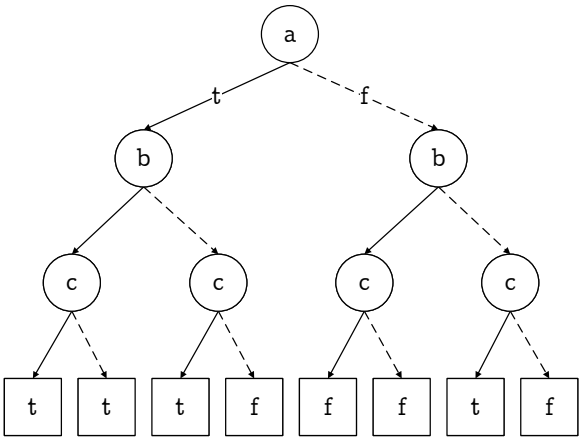

Figure 2.2: Binary decision tree for $(a \vee \neg b) \wedge (b \vee c)$ expression with a, b, c substitution order

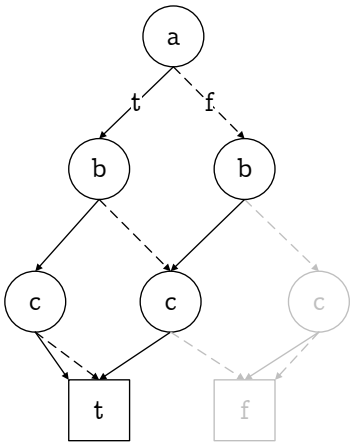

Figure 2.3: Binary decision diagram for $(a \vee \neg b) \wedge (b \vee c)$ expression with a, b, c substitution order

2.3.3 Multi-value decision diagram

The extension of the previous definitions happens analogously for discrete domains like integers. Edges, nodes and labels have the same meaning as by the binary versions. The only difference is that nodes may have several, possibly infinite outgoing edges. This is the consequence of that the substituted value can be any value from the appropriate discrete domain. An instance of MDD is shown in Fig. 2.4. The variable order is a, b and the expression is

$$(a < 3 \wedge a \geq 0 \wedge (b = 1 \vee b = 2)).$$

2.4 Logic-based model checking techniques

In this section, we present the occurrences of the satisfiability problems during the model checking procedure.

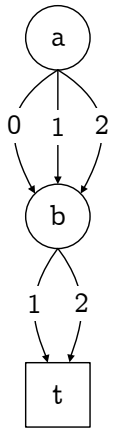

Figure 2.4: Multi-value decision diagram for $(a < 3 \wedge a \geq 0 \wedge (b = 1 \vee b = 2))$ expression with a, b substitution order

2.4.1 Path-based techniques

A class of model checking techniques approaches the problem by describing paths of finite length in the state space. The correctness of the system is established based on the existence of these paths.

2.4.1.1 Bounded model checking

Bounded model checking [1] is a symbolic model checking technique using SAT procedures. Its basic idea is to search for counterexamples of length k by creating a propositional formula that is satisfiable if and only if there exists a counterexample. The name *bounded* stands for the limit k, which ensures that a counterexample with a possibly small size will be found, but does not guarantee completeness (i.e. it will not find paths longer than k). The process of finding a counterexample is quite fast due to the nature of SAT solvers.

2.4.1.2 k-induction

A bounded method that is capable of proving correctness is k-induction [11], which aims to prove the safety of a system in terms of a given invariant. The basic idea is to find a bound k, for which all states not further than k steps from the initial states are safe and the following inductive condition is also true: When every path of length k contains only safe states, then any path of length $k + 1$ will also contain only safe states. This proves inductively that every reachable state is safe.

2.4.2 Abstraction-based approaches

Non-bounded techniques have to explore the complete state space to reason about states and behaviours. The state space can often be too large to handle, or even infinite. To counter this problem, a class of model checking techniques use abstraction. This can be done easily on symbolic transition systems, in which case the state space exploration problem contains multiple instances of AllSAT/AllSMT problems: whenever we compute the successors of a given state.

2.4.2.1 CEGAR

Counterexample-Guided Abstraction Refinement [5] is a generally known technique for automated verification of both hardware and software systems. The basic functionality is based on the CEGAR loop (see Fig. 2.5), which is iteratively constructing and refining abstractions until a proper precision is reached. Initially, a coarse abstraction of the system is computed (the abstract reachability graph, ARG), which will be refined later. When an erroneous behaviour can be found in the abstract state space, it is checked whether the abstract counterexample is reproducible in the original system. When it is, the feasible counterexample proves that the original system is unsafe. Otherwise, the counterexample is spurious, and the abstraction must be refined. This procedure is repeated until either one concrete counterexample is found or the abstract state space is safe. CEGAR is a general method, which may be specified by choosing different abstraction domains and refinement methods. My work is applied to both predicate and explicit-value abstraction, which concepts are detailed in the following sections.

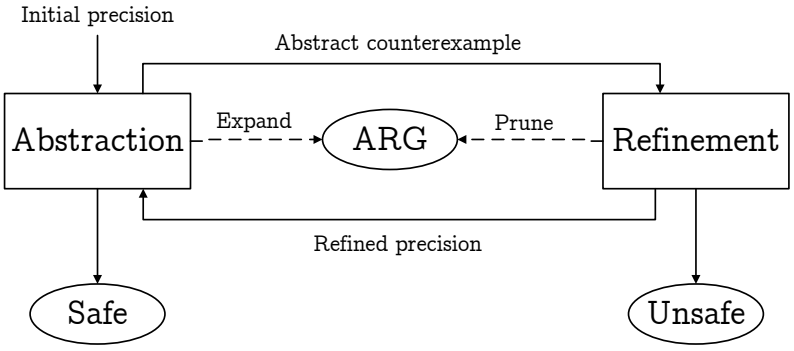

Figure 2.5: CEGAR loop

Definition 3 (Abstraction). Given a concrete state space (i.e. Kripke structure) an abstraction is a function mapping concrete states to abstract states based on a *precision* π which defines the abstract domain D, the way to transform a concrete state to an abstract state, and an abstract transfer function T that contains a pair of abstract states if and only if there is a (concrete) transition between any two concrete states mapped to the corresponding abstract states. ▪

2.4.2.2 Predicate abstraction

In Boolean predicate abstraction, a Boolean combination of first-order logic formulas represents abstract states [5]. The precision π stores the currently tracked predicates of the algorithm. The transfer function $T(s, op, \pi)$ results in a predicate combination that is entailed by the source state and the operation. The idea behind this method is to assign a new propositional variable to each predicate and find all assignments for the propositional variables that evaluate to true. For each assignment, the conjunction of predicates is created, where predicates are present in the suitable (positive or negated) form.

2.4.2.3 Explicit-value abstraction

In explicit-value abstraction, an abstract state is an abstract variable assignment that maps each variable to an element from its domain extended with a *true* or $false$ value

[5]. A precision π is the set of currently tracked variables, which set is extended in each iteration. The transfer function determines the successor states such that the variables not included in the precision (i.e. the variables not being tracked) are omitted.

2.5 Related approaches in AllSAT solvers

As mentioned before, there exist available solvers for both SAT and AllSAT problem. In general, the range of SAT-solvers is way wider, though there are a few AllSAT solvers too. In the following, an AllSAT solver is detailed, which is based on some ideas that may also be useful in our future work. The details can be read in [13].

The article examines the functionalities of AllSAT solvers, including BDD-based methods. However, it is important to mention that the article investigates the internal operation of the solvers, while this work finds solutions from the outside, using existing solvers.

Section 2.3.2 presented the concept of binary decision diagrams, which was the result of merging the equivalent nodes of the binary decision tree (i.e. the identical subtrees). Building a binary decision diagram has a lot of intermediate steps so the decision diagram-based solution in the article completes the structure with some additional information. Based on this information it is possible to build the final decision diagram directly. which says during the construction phase of the diagram whether the actual node is already present in the structure, without traversing the underlying subtree, To understand the proposed solution, we must first get familiar with the concepts of cutsets.

Definition 4 (Cutset). For a given CNF (conjunctive normal form) formula, the i-th cutset is the set of clauses, which contains variables with both non-greater and greater indices than i. ▪

Example 1. *Fig. 2.6 gives an example for the both definitions. In the figure the filled circles mean poned, the empty ones mean negated variables, horizontal lines are for the clauses. With this symbolism the clauses in the example are:*

$$
\begin{array}{ll}
C5: & x5 \wedge \neg x6 \\
C4: & x4 \wedge \neg x5 \wedge x6 \\
C3: & \neg x1 \wedge \neg x3 \wedge x4 \\
C2: & x2 \wedge x3 \wedge x5 \\
C1: & x1 \wedge \neg x3
\end{array}
$$

As an example of $i = 3$, the 3rd cutset is the $\{C2, C3\}$ set of clauses, while the 3rd separator is the $\{x_1, x_2, x_3\}$ set of variables.

The key idea described in the article is the observation that investigating two nodes on level $i + 1$, they are equivalent if and only if clauses in cutset i have the same fulfilment along the paths leading to the nodes. It follows that it is enough to store the true or false values of these clauses to decide the equivalence, without traversing the underlying subtrees.

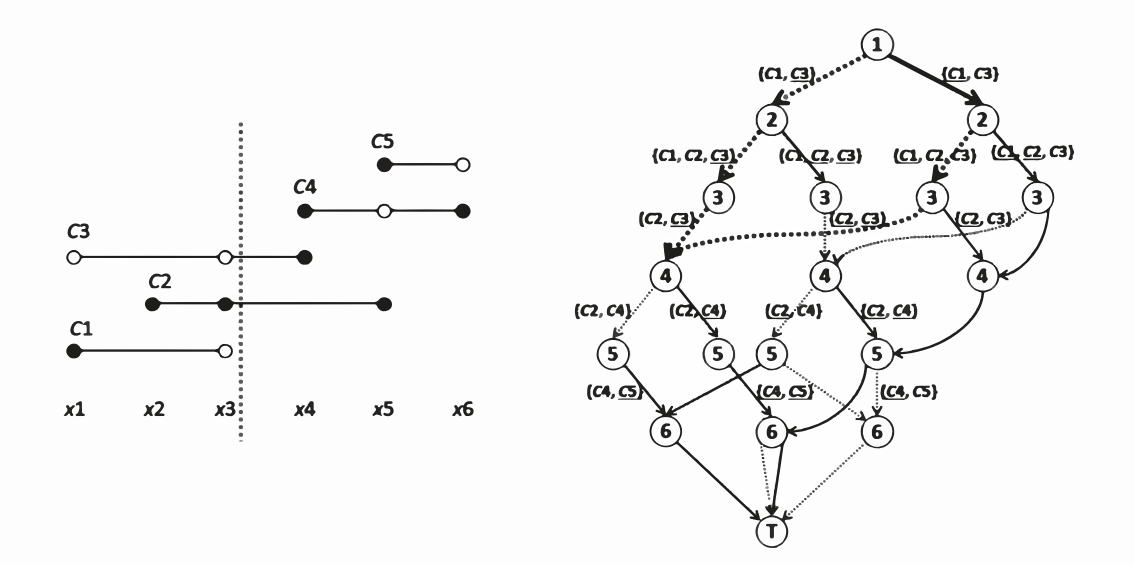

Figure 2.6: Cutset example from [13]

Chapter 3

Flexible Computation of Multiple Solutions of SMT Formulas

As we have seen, the AllSAT/AllSMT problem comes up again and again in the area of model checking. Currently, most model checking tools employ a naive solution building on SAT/SMT solvers to compute the successor states during state space exploration. This approach may suffer from scalability issues when the number of successor states is large or infinite. The main motivation behind our work is to handle this issue by proposing a novel decision diagram-based data structure: the *substitution diagram*. The proposed approach attempts to combine the advantages of decision diagram-based techniques with abstraction-based techniques as they generally perform well in other areas of model checking. We expect our solution to be competitive with the naive approach, but much more flexible, which we will illustrate by outlining different use cases where our solution is more appropriate.

The chapter is structured as follows. Firstly, Section 3.1 presents a simple and widespread solution for the AllSAT/AllSMT problem. Thereafter Section 3.2 presents the definition of substitution diagrams. Finally, Section 3.3 details the possible applications of the described solution.

3.1 The loop solution

The naive solution builds on existing SMT solvers by calling them in a loop. The loop algorithm is shown in Fig. 3.1. The algorithm iterates over the solutions as follows. In the first step, an evaluation is queried from the solver. After the negation of this solution is conjoined to the input expression. The result is the new input expression that will be read by the solver in the next operation. Thus the solver is told to return an assignment that is not among the already found ones. This iteration is repeated until the unsatisfiable result is returned, i.e. until all possible assignments are found.

This algorithm provides a theoretically correct solution for the AllSAT/AllSMT problem, although it often cannot be used in practice. In the case of large state spaces, the one by one listing of solutions is not feasible. As there are numerous assignments, the negations of these cannot be concatenated to the end of the original expression due to memory restraints. This solution has been used as a baseline to evaluate the efficiency of our proposed approach.

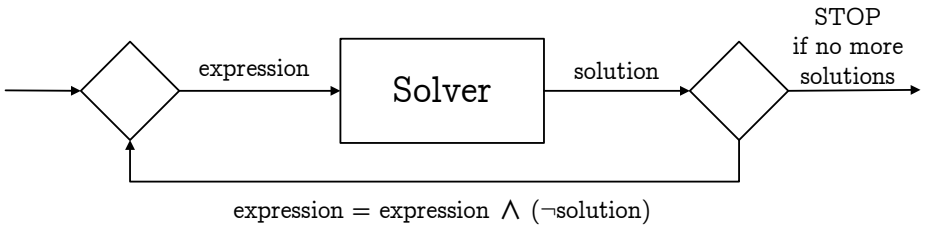

Figure 3.1: Loop solution for the AllSAT/AllSMT problem

3.2 Substitution diagram

This section introduces the solution diagram, a decision diagram completed with additional information. The substitution diagram gives a more flexible solution to the AllSAT/AllSMT problem. The decision diagram-based data structure has the advantage of a more compact representation. It also offers a configurable strategy to sample infinite solution spaces and custom traversal strategies.

Definition 5 (Substitution diagram). A substitution diagram over K variables with domains embeddable into the set of natural numbers is a tuple (N, L_N, R, E, V, L_E) such that:

- V is the set of nodes, containing the terminal *true* and *false* nodes (**1** and **0**, respectively);
- $L_N : N \to \mathit{Exprs}$ is the node, labeling function mapping an expression to each node, with $L_N(\mathbf{1}) = \mathit{true}$ and $L_N(\mathbf{0}) = \mathit{false}$;
- $r \in N$ is the root node;
- $E \subseteq (N \setminus \{\mathbf{1}, \mathbf{0}\}) \times N$ is the child relation consisting of node pairs (edges of the diagram);
- V is an ordered set of variables, where $|V| = K$. Nodes in $(N \setminus \{\mathbf{1}, \mathbf{0}\})$ are assigned to variables in V (denoted by $\mathit{var}(n)$ for $n \in N \setminus \{\mathbf{1}, \mathbf{0}\}$) and we require that edges lead to nodes lower in the order determined by V;
- $L_E : E \to \mathbb{N}$ is the edge labeling function which assigns a value from the corresponding variable domain to the edges. ▪

Similarly to multi-value decision diagrams, a substitution diagram can be represented by a directed acyclic graph (see Fig. 3.2). Nodes are labelled and identified by the corresponding expression added to each node. The nodes are grouped into levels, and a variable is assigned to each level, which is the next to be substituted. The levels below each other are in the order of the predefined variable order. Each edge corresponds to a variable substitution. As the variable substitutions happen in a fixed order, the edges go between neighbouring levels. Each edge is labelled with the substituted value for the current variable. The edge starts from the expression for which the substitution happens while it ends in the node with the resulting expression. The resulting expression is obtained by substituting the value and optionally reducing and/or canonizing the expression.

However, in contrast with MDDs, a substitution diagram is not required to merge every identical subdiagram. The relaxed reduction rule for substitution diagrams will build on the expressions used to label the nodes: there can be no two nodes belonging to the same variable with the same label.

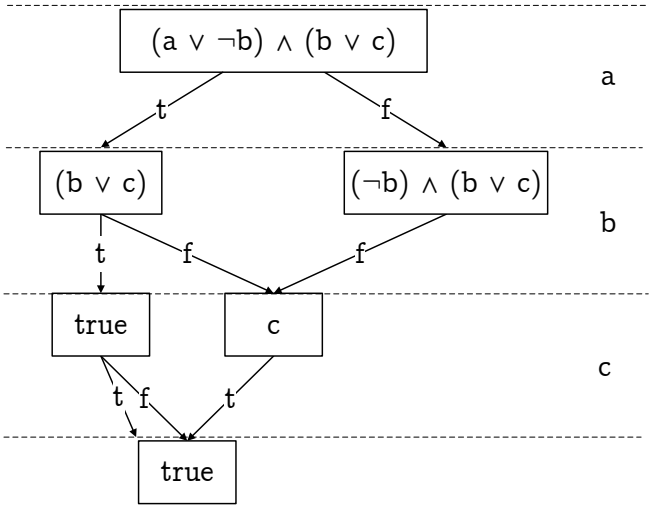

Figure 3.2: Substitution diagram for expression $(a \vee \neg b) \wedge (b \vee c)$ and variable order (a, b, c)

Definition 6 (Well-formed substitution diagram). A substitution diagram is well-formed if:

- $\nexists n_1, n_2 \mid n_1 \in N \wedge n_2 \in N \wedge L_N(n_1) = L_N(n_2)$;
- $\forall n_1, n_2 \mid n_1 \in N \wedge n_2 \in N \wedge ((n_1, n_2) \in E \implies [\langle L_N(n_1) \rangle \wedge \langle var(n_1) \rangle = \langle L_E(n_1, n_2) \rangle \Leftrightarrow \langle L_N(n_2) \rangle])$, i.e. for every two node connected by an edge, the label of the source node as a logical expression in conjunction with the expression binding the variable corresponding to the node to the value assigned to the edge is equivalent to the label of the target node as a logical expression. ▪

The relaxed reduction rule can be regarded as a kind of syntactic equivalence, whereas the reduction rules of MDDs operate with semantic equivalence. Our observation is that syntactic equivalence implies semantic equivalence, so substitution diagrams do not lose information – the only thing we may lose is the compactness of the representation. However, this will be offset by the cheaper comparison of expressions and the ability to directly construct substitution diagrams from mathematical expressions without building intermediate diagrams.

Theorem 1. The set of all satisfying substitutions is equivalent to the set of valuations along the paths leading from the root node to the terminal *true* node. ▪

The theorem follows from the second part of Def. 6 by transitively applying the equivalence along each path.

3.2.1 Constructing the diagram

This section describes the construction method of the diagram. The starting point is a given expression and a predefined variable order, the aim is to construct the corresponding substitution diagram. The construction happens with the help of an SMT solver. The solver gets an expression as an input and returns a satisfying solution as an output when possible. The initial diagram consists of only one node that stores the expression. To expand this node, the solver gets the expression as an input and returns an output. If

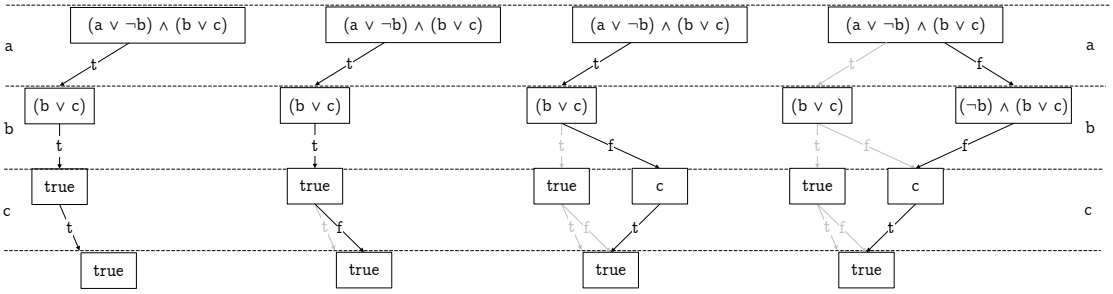

Figure 3.3: Substitution diagram construction

the result is *unsatisfiable*, then the current node is replaced with the terminal *false* node. Otherwise, the result is saved by iterating over the variables, substituting the offered values and creating new expression nodes along the path, reusing already existing nodes whenever possible. New nodes are created by performing the substitution and simplifying/canonizing the resulting expression. If the immediate child of the current node is a new node, then we expand it recursively. Next, the value offered for the current variable is used to append a new constraint to the solver, excluding solutions which would yield the same value for this variable. We repeat this until the solver returns *unsatisfiable*, in which case we mark the node as final and finish the expansion. The substitution diagram when the root node is expanded. Notice that whenever a node already exists, it has to be final, and we will set it as a child and will not expand it again.

Example 2. *As an example, the construction of the previous diagram is presented in Fig. 3.3.*

In the beginning the expression

$$(a \vee \neg b) \wedge (b \vee c)$$

and the variable order (a, b, c) *are available. The root node is created with this expression, and then a Z3 solver is asked for a satisfying substitution. The* $(true, true, true)$ *values are proposed so we add it to the diagram. As the construction method is a DFS-based method, DFS steps follow. Backtracking starts from the resulting true node, and each node is searched for further new solutions. To do this, by each step, extra predicates are added to or popped from the solver expression. In this concrete example, the first backtracking step leads to the true expression on level c. The solver is asked here, whether the expression*

$$(a \vee \neg b) \wedge (b \vee c) \wedge (a = true) \wedge (b = true) \wedge (c \neq true)$$

is satisfiable i.e. whether a new edge can be drawn from the current node. The solver returns the $(true, true, false)$ *solution so the edge with label* $false$ *is added to the diagram. The next solver expression is*

$$(a \vee \neg b) \wedge (b \vee c) \wedge (a = true) \wedge (b = true) \wedge (c \neq true) \wedge (c \neq false)$$

for which the solver returns unsatisfiable, so no more edges can be added to the current node. After this result, the backtracking step follows, the predicates with c are removed from the solver expression, and the search is continued from the next node.

3.2.2 Default next nodes

In some cases, the resulting expression is independent of the substituted variable value. Therefore we introduced a new special edge label. The label *default* means that the resulting expression node is the endpoint of the edge regardless of the substituted variable, i.e. any value of the variable domain is suitable (even if the domain is infinite). This improvement helps saving storage capacity as the one by one listing of substituted values can be avoided. The simplest case for the application of *default* edges is when the expression does not contain the level variable. Our solution also uses default edges for this case. There may be other cases when *default* labels could be used. They are not so easy to recognize though. Optimizing the usage of default edges poses theoretical challenges, but could lead to significant improvements in the future.

3.2.3 Comparison to the loop solution

Also confirmed by profiling results, the most costly operations of computing every solution are the solver calls. As other operations take a negligible time amount compared to solver handling methods, the comparison will happen based on this.

In the loop-based method, each solution of the expression requires a solver call (plus one extra in the end, which results in *unsatisfiable*). The amount of solver calls is, therefore, $SOL + 1$, where SOL is the number of solutions.

Substitution diagram also saves each solution after a solver call. It has a greater overhead however, as each node most get an *unsatisfiable* result before backtracking. So this method demands $SOL + |N|$ solver calls, where $|N|$ is the number of diagram nodes. As the decision diagram offers a compact structure, this overhead is not expected to be critical. Furthermore, solver expressions generated by this approach should be smaller.

The main disadvantage of the loop-based solution comes from memory demand. The storage needed by the looped algorithm is proportional to the number of solutions, i.e. the previous solutions cannot be cached or reused. The other disadvantage is that by later solver calls the solver expressions may get unmanageably large. The explanation is that it lists all previous solutions one by one to exclude them.

The advantage of the substitution diagram is the compact structure that reuses previous solutions. Thereby previous solutions (or previous expression nodes) may be reused. The solver expression also gets smaller as the common parts of the previous solutions are merged owing to the decision diagram virtue.

It must be mentioned that in this case, the diagram sizes are strongly varying and are not proportional to the solution space size. We also have to note that compactness will not be relevant in explicit-state model checking, because for each solution the model checking algorithm will create a new state.

Example 3. *Fig. 3.4 shows an example. The expression* $(a \vee \neg b) \wedge (b \vee c)$ *with the variable order* (a, b, c) *has* 4 *solutions and the diagram has* 6 *nodes. On the other hand the* $(a \vee \neg b)$ *expression has* 8 *solutions for the same variable order, the diagram has however only* 4 *nodes.*

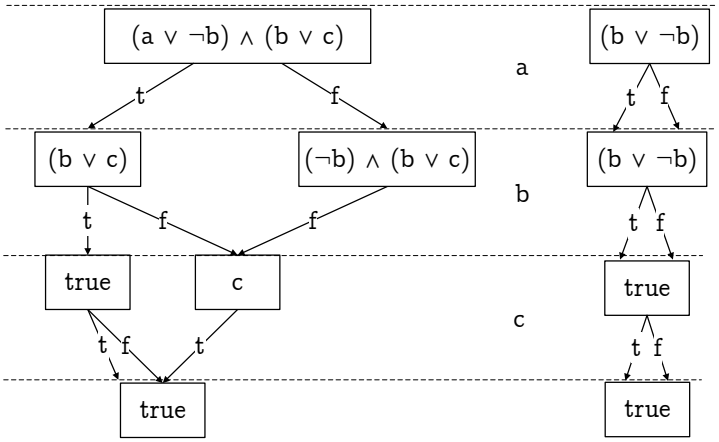

Figure 3.4: An example showing that solution space size is not proportional to the diagram size

3.3 Use cases

As mentioned before, the substitution diagram offers an alternative for existing AllSAT/AllSMT solvers and the loop-based solution. The main advantage compared to an AllSAT/AllSMT solver is that the data structure is available for manipulation, which provides flexibility in traversal strategies. In the following subsections, we investigate the possibilities of this flexibility through three use cases that we have also implemented (see Chapter 4).

There also is a fourth promising direction to continue the research, relating to the external control of solution space traversal. Binding certain variables on the fly would allow exploring any subset of the solution space. This feature can be useful in decision diagram-based symbolic model checking algorithms such as saturation [2].

3.3.1 Retrieving all solutions

The first case is the solving of the original AllSAT/AllSMT problem. In this case, the calculation of all satisfying variable assignments is desired. For this, Section 3.1 gives an alternative solution method. As discussed before, substitution diagram may offer a better solution, as the one-by-one listing of solutions requires too much memory. The diagram provides a more compact structure, from which the individual solutions can be read through a DFS traversal. However, this problem can only be solved on either Boolean variable domains or integer domains with finite solutions.

3.3.2 Retrieving a specific number of solutions

This use case emerges when dealing with the general AllSMT problem. Extending the variable domains from Boolean to discrete domains like integers or enumerations raises new problems. After the expansion, infinite domains are also allowed, which makes it impossible to list all solutions. With the solution diagram structure, the domain expansion means simply changing the possible label value set. Retrieving only a given number of solutions can be done without calculating all solutions, as the diagram construction may

be stopped any time. It can also be modified later, i.e. first storing only a few solutions, and then later complementing it with additional solution paths.

Retrieving only k solutions may be useful methods with numerous or infinite solutions. As the number of possible assignments may be infinite, a limit of k may be set so that after finding k solutions, we assume it to have an infinite number. This use case is still covered by the loop-based solution too, which may provide better performance when k is low but will fail when k is too large.

3.3.3 Only k values of each variable are relevant

This case is related to the previous one. When searching for a specific number of solutions for a given formula, a variable with infinite possible values may lead to detecting very similar results. By allowing only k different values for each variable, the almost identical solutions can be omitted. Thus by assuring the wider diversity of solutions leads to a more even sampling of the solution space. By limited investigations, this feature provides greater coverage, which may lead to a larger chance to uncover faults during model checking.

Chapter 4

Implementation

The theoretical approach detailed in Chapter 3 has been implemented and integrated to the Theta model checking framework. This chapter gives information about software design decisions and architecture details. Section 4.1 investigates the tool as a whole. Section 4.2 follows with the class structure and the used design patterns. Finally section 4.3 describes the realized application areas.

4.1 Design and architecture of the software library

4.1.1 Theta framework

Theta [14] is an open-source model checking framework developed at the Critical Systems Research Group of Budapest University of Technology and Economics. Being an extensible and configurable framework, Theta offers a wide range of model checking algorithms for numerous model types like programs and statecharts. There are multiple options for formalisms like control flow automatons (CFA) or (extended) symbolic transition systems (XSTS) as well as for model checking approaches like predicate or explicit-value abstraction. During the model checking procedure, various SMT solvers may be used, including the SMT solver Z3 by Microsoft [4]. Among the many Theta components, we used the following.

- Core: This module offers several tools for handling expressions, statements, literals and valuations.
- Solver: A component that makes the solver functions available.
- Analysis: The project containing analysis algorithms and their components.
- Cfa-cli: A project which offers a tool for running CEGAR-based analyses on CFAs.
- Xsts-cli: This project offers a tool for running CEGAR-based analyses on XSTSs.

4.1.2 The AllSolutionSolver solver module

The substitution diagram approach described in the previous chapter is realized as a new component of the Theta framework. The two major blocks realized in the project are the classes for describing substitution diagrams and the proposed `AllSolutionSolver` interface, which allows the usage of both loop or substitution algorithms without their deeper understanding. The general structure of the module can be seen in Fig. 4.1.

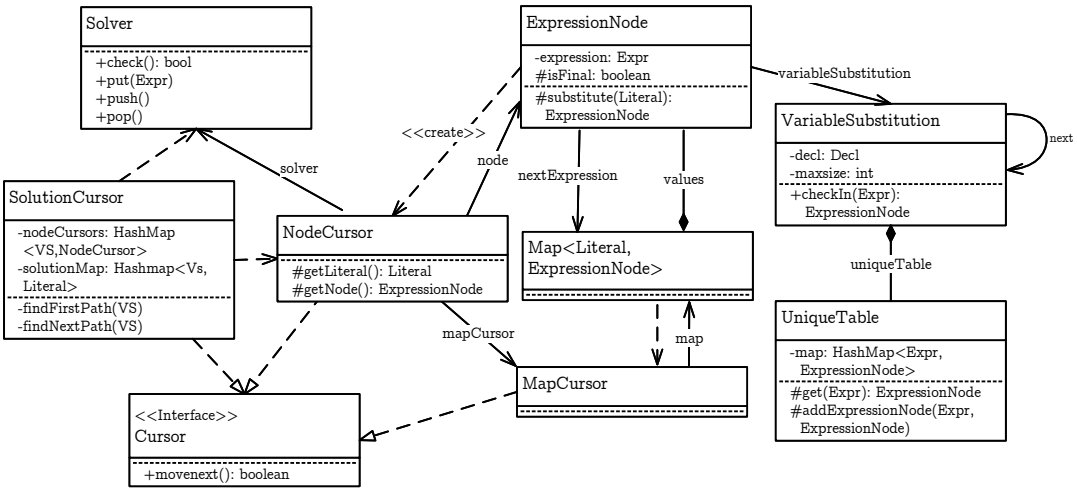

Figure 4.1: The basic architecture of the AllSolutionSolver module

4.2 Implementation details

4.2.1 All solution solvers

The `AllSolutionSolver` interface (see Fig. 4.2) facilitates the usage of the substitution diagram structure as a black box. It extends an `Iterator<Valuation>` interface which allows the iteration through the satisfying substitutions for a given expression.

There are two classes implementing the interface: `LoopAllSolutionSolver` and `MddAllSolutionSolver`. The former class realizes an AllSAT/AllSMT solver running the naive loop algorithm. Each time when the iterator's `next()` function is called, a new satisfying substitution is returned, and its negation is added to the current solver instance. On the other hand, the latter class realizes the substitution diagram-based AllSAT/AllSMT solver functionality. The main advantage is that the `next()` function call finds a new solution and adds the corresponding nodes and edges to the structure. The usage of cursors on different levels facilitates a continuous DFS on the diagram. It is advantageous as in each iteration only the new nodes and edges are added to the structure, so the complete construction of the diagram is not necessary to list some solutions.

4.2.2 Factories

The factory design pattern is responsible for the comfortable usage of `AllSolutionSolver` instances. The `AllSolutionSolverFactory` interface is responsible for creating an AllSAT/AllSMT solver instance. Two factory classes implement the interface: `LoopAllSolutionSolverFactory` and `MddAllSolutionSolverFactory` (as in Fig. 4.3). Both classes are singleton classes, i.e. they have only one instance, which may be got with a `getInstance()` function call. The two factory types are responsible for creating and returning a new AllSAT/AllSMT solver of the corresponding type.

The Factory pattern is a popular software design pattern as it facilitates simple extensibility. It also contributes to the principle of least knowledge software design guideline, as the usage of the abstraction (loop or decision diagram-based solution) is decided at a

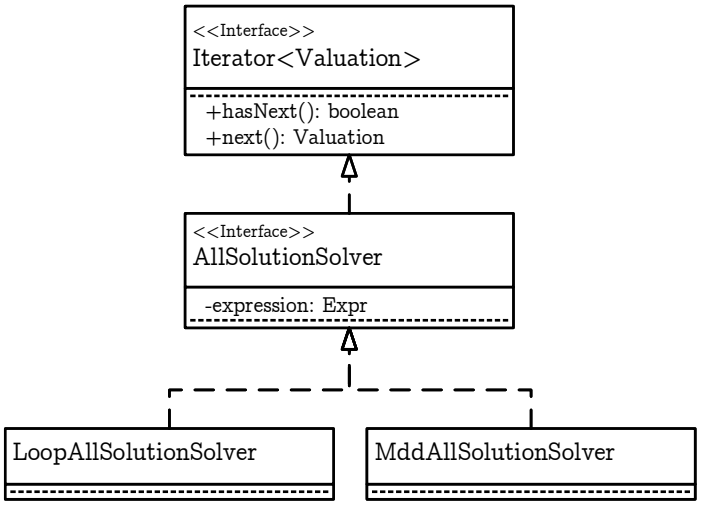

Figure 4.2: The `AllSolutionSolver` interface

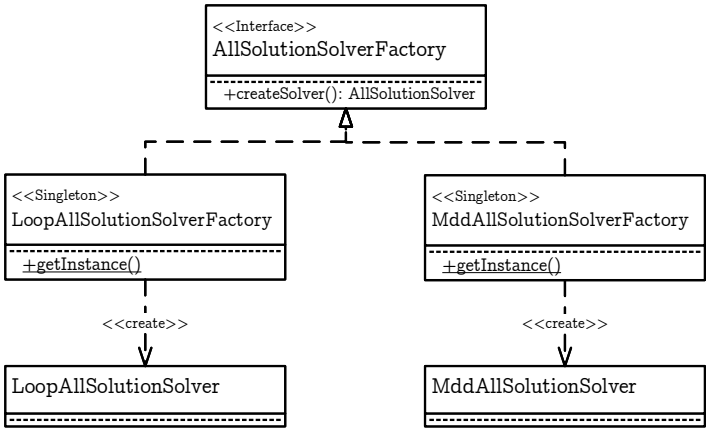

Figure 4.3: The `AllSolutionSolverFactory` interface

high level, where the lower level modules do not depend on the decision. The original code contained the loop-based solution inlined, which has been refactored to enable the minimal modification of existing model checking code (such modifications would risk the correctness of already established implementations).

4.2.3 Variable substitutions

The `VariableSubstitution` class corresponds to the variables being present in the given variable order, i.e. each variable has a corresponding class instance. The major attribute of an instance is an `UniqueTable` object which is a map assigning an expression to the diagram nodes. The role of this table is to store all expressions being present at the current level. Before the construction of a new node, its expression is looked up in the table. It ensures uniqueness by returning the existing node when it is already present, thus avoiding unwanted duplicates. Otherwise, a new node is created and added to the table.

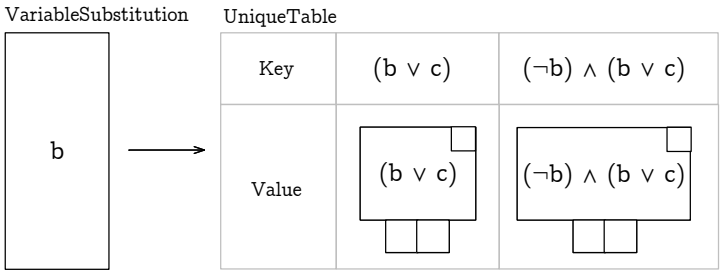

Figure 4.4: Variable substitution for variable b and its unique table from example of Fig. 3.2

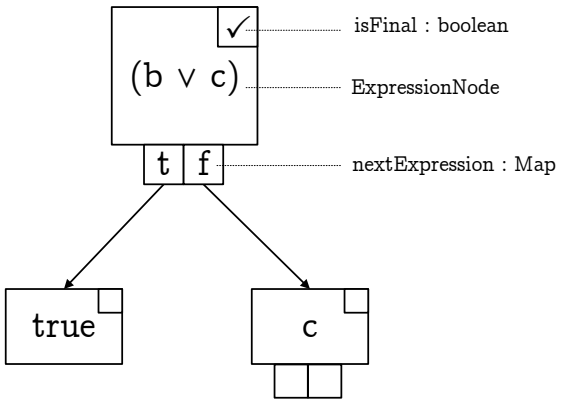

Figure 4.5: Expression node for expression $(b \vee c)$ from example of Fig. 3.2

As the variable order is predefined, it is stored as a linked list of `VariableSubstitution` instances. This way every variable substitution stores a reference to the next one, which identifies the level below. See a visualization of the described classes in Fig. 4.4.

4.2.4 Expression nodes

The class `ExpressionNode` is for nodes of the substitution diagram. Each node stores an expression and a variable substitution which identifies its level. The `nextExpression` map stores the edges, mapping the substituted literal to the resulting expression node. New edges can be added to the diagram by adding new entries to the maps of the suitable expression nodes. The node to be added is the result of the literal substitution function. It is essential to note that new nodes can only be created through this function – this way the library can guarantee that reduction rules are respected. Besides there is a Boolean `isFinal` flag for each node for which a true value means that the investigation of the node is over, i.e. the underlying subtree is completed. See an example in Fig. 4.5.

4.2.5 Cursors

After describing the implementation of the structure of the substitution diagram, we now describe the details of finding the satisfying substitutions. To achieve an interruptable depth-first search, we use multiple layers of the cursor design pattern. The three kinds of cursors are detailed in the following.

4.2.5.1 MapCursor

`MapCursor` is the lowest-level cursor. As mentioned before, each node has a map `nextExpression` mapping the substituted literals to the resulting nodes, i.e. the outgoing edge labels to their endpoints. The `MapCursor` is the built-in cursor for this map.

4.2.5.2 NodeCursor

`NodeCursor` is one level above the map cursor. An instance belongs to an expression node, and the iteration happens through the possible substitutions for the node's variable. The search begins through the already computed `nextExpression` map entries (if any). After this, if the node is not marked as final, the solver asked for a new solution. New solutions are cached into the `nextExpression` map. When the node cursor ends its iteration, the node is marked as final, i.e. no more literals can be substituted for the current variable, the underlying subtree is complete.

4.2.5.3 SolutionCursor

`SolutionCursor` is the highest-level cursor. It also owns the same solver instance as node cursors, as well as a map of node cursors for each variable substitutions and an expression node. The main purpose is to find solutions without making redundant steps for graph traversals. This is done by saving the current node cursor states in the map so that after returning a satisfying solution the search for the next one may be retrieved from where it left off. This method also facilitates a continuous diagram construction, i.e. the acquirement of only some solutions without building up the whole diagram. The usage of the solution cursor from the outside is completely convenient as a new solution can be obtained with a simple `moveNext()` function call.

4.2.6 Solver

The most costly fragments of the algorithm are the SMT solver calls. Therefore the aim is to keep the number of these calls possibly low. The generic Theta solver interface was used in the implementation, which offers push and pop functions besides the basic expression adding and satisfiability checking features. This feature is useful during the DFS of the solution space, as this way a backtracking step means a solver pop instead of a reinitialization (this is why the solution cursor also needs a reference to the solver). Thereby valuable runtime may be saved, which is a constant bottleneck in model checking.

4.3 Integration into Theta

The solution described in the previous section was applied in the Theta model checking process. The framework realizes the CEGAR algorithm in a highly configurable way, supporting for example predicate and explicit-value abstraction, various refinement strategies and multiple input formalisms.

4.3.1 Predicate abstraction

In predicate abstraction, the abstraction is determined by a set of tracked predicates. The transition function is responsible for finding the successor states. As the states and transitions are described with formulae, the successor state creator function meets an AllSAT problem instance – this is where the original loop solution can be switched for the decision diagram-based method. Determining the possible next states happens as follows. Firstly a Boolean value, called *activation literal* is assigned to each predicate. Then the transfer function formula is built by extending the original formula with these literals. Even though this formula contains the original variables too, only the values of Boolean activation literals need to be computed. This is allowed by the implementation, because expression may contain variables that are not present in the variable order.

4.3.2 Explicit-value abstraction

The other main application area is explicit-value abstraction. This variant tracks variables instead of predicates. During an iteration, a set of variables is investigated, the other variables are omitted. Refinement means extending the tracked variable set. It also applies here that states and transitions are described with formulae. Finding the next states means solving an AllSMT instance to the transfer function. In contrast to predicate abstraction, there may be both finite and infinite domain variables (i.e. Booleans and integers) in this case, and untracked variables are free to assume any value. The consequence is that when the solution set is too large (or infinite), the solutions cannot be listed. Thus except when it is not sure that all domains are finite, a limit k is determined so that only k solutions are interesting. When the limit is reached, the solution set is stated to be infinite, and the search for solutions is stopped. This way the AllSMT (or k-SMT) problem can be solved with the help of the substitution diagram.

Even though the library supports it, the "k values per variable" mode was not integrated due to conflicts with existing configurations (to be solved later).

4.3.3 Supported configurations

The command-line parameters of Theta have been extended with the settings `-allsat LOOP / MDD` and `-domain PRED / EXPL` to control the activation of the substitution diagram-based approach. We built our solution in both *cfa-cli* and *xsts-cli* modules, i.e. the new solution is supported in checking both CFA and XSTS models.

Chapter 5

Evaluation

The implemented solution has been evaluated in multiple configurations. The parameters varied during the benchmarking include the following:

- Solution: naive (loop) or solution diagram (MDD-based); type
- Model type: CFA or XSTS;
- Domain: predicate or explicit-value abstraction;
- MaxNum(k): the preset limit of solutions (by EXPL)-

5.1 Benchmark models

Control flow automata The measurements were done on a selection of 40 different CFA models. Most of the models come from the 2019 Competition on Software Verification (SV-Comp [12]) reachability tasks. The CFA models were generated from C codes with the Eclipse-based C/C++ Development Tools front end for Theta [10].

Extended symbolic transition systems The XSTS measurements were run on 53 models. The model set is a diverse set of Gamma Statechart Composition Framework models [8]. The following real-life model types are included:

- COID: The modelled system is composed of the antivalence filter and signalling components of the railway safety equipment.
- PIL: These systems are also composed of two railway safety equipment components: switch and signalling. They are larger and more complex than the previous ones and aim to help to model and to verify track reservation protocols inside a safety equipment algorithm.
- INPE: These models describe a communication protocol in a nanosatellite.

5.2 Benchmark results

We ran several experiments on various model types and configurations. The results are detailed in the following. Each configuration has been executed twice, once with the original loop algorithm and once with the substitution diagram solution. The measurements were run on a virtual computer with the following resources:

- 8GB RAM
- 4 CPU cores
- Ubuntu 18.04 OS
- Java 11

5.2.1 CFA results

There were 8-8 configurations for LOOP and MDD methods that traditionally work well with CFA models, including explicit-value and predicate abstraction types. No configurations yielded an incorrect result.

The number of models verified within the time limit of 90 seconds can be seen in a heat map representation in Fig. 5.1 for each configuration and model type. The interpretation of configuration codes is the following:

- the starting letter E/PB is for explicit-value or predicate abstraction method;
- the following number is the *maxEnum* value which is the maximal number of desired solutions. 0 means infinite (i.e. all solutions are desired);
- the last L or M letter means a LOOP or MDD algorithm;
- the rest of the letters denote configurations that are orthogonal to the use of substitution diagrams (e.g. refinement strategy).

With this interpretation, two consecutive lines compare the results of each configuration with MDD and LOOP methods. The predicate abstraction method was tested for 2 configurations (first two rows) and showed that the overhead of using substitution diagrams could not be compensated by the low number of solutions that are inherent in predicate abstraction (as discussed in 3.2.3).

The explicit-value abstraction results carry more information. It can be seen that the MDD solution is as good as the loop-based solution – there was no configuration where the LOOP solution verified more models. They performed the same in most cases, but there was one case where the MDD could verify a model which the original solution could not.

The distribution of success, time limit, memory limit and exception can be seen in Fig. 5.2. The LOOP and MDD method perform alike on the model set in terms of almost all aspects – the only difference is the extra verified model of the substitution diagram method.

Fig. 5.3 shows the best 10 configurations based on the measurement results. Although the success counter of the solution diagram algorithm keeps up with the original one, the table shows that the new method ran slower on the models. Nevertheless, it is not a significant overhead, so the new approach may offer an alternative to the old one even when the increased flexibility is not exploited.

5.2.2 XSTS results

The XSTS measurements were run with 7-7 configurations for both LOOP and MDD approaches which traditionally work well with XSTS models, including explicit-value and

predicate abstraction types. Experiments on XSTS models also yielded solely correct results.

The results are shown in a heat map in Fig. 5.4 for the individual configurations and model types. The configuration code interpretation is analogous to the codes used with CFA measurements. Again, two consecutive rows compare the same configuration differing only in the enumeration approaches. Results with predicate abstraction show the same patters as with CFA models. With explicit-value abstraction, the number of verified models is again similar for the two approaches, although the loop-based method performed slightly better this time. The reason for this is subject to future investigation, but it might be related to a different trade-off in executions time vs. memory.

Fig. 5.5 compares the results according to the termination causes of success, time limit or memory limit. These results seem to support our assumption that the substitution diagram implementation sacrifices execution time for a smaller memory footprint. The best configurations in this setup can be seen in Fig. 5.6.

5.2.3 Overall evaluation

As we have examined only two of the four identified use cases, both of which are supported by the loop-based solution, our expectations were not so high – an ideal outcome was measuring a low overhead. Fortunately, this expectation has been fulfilled by the results, and we can say that increased flexibility does not come at a high price. This confirms that the direction we took with this approach is worthy of a more thorough investigation and further developments, and may provide real gains when used in setups that were not realizable/efficient before.

Figure 5.1: CFA benchmark results

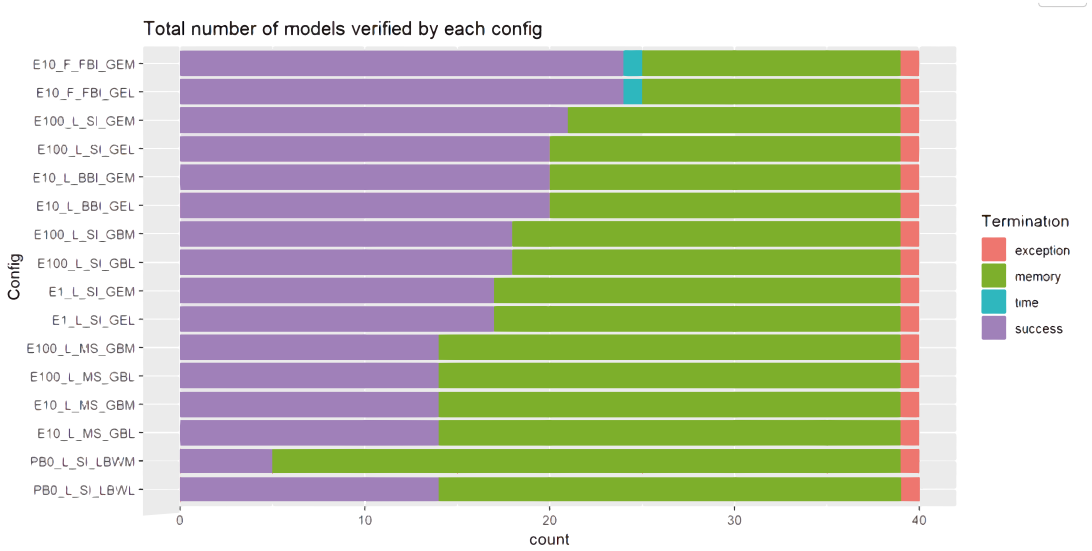

Figure 5.2: CFA benchmark results

Config	Domain	Encoding	Refinement	PrecGranularity	PredSplit	Search	MaxEnum	InitPrec	PruneStrategy	AllSat	SuccCount	Unique	TimeSum	TimeMax
E10_F_FBI_GEL	EXPL	LBE	FW_BIN_ITP	GLOBAL		ERR	10	EMPTY	FULL	LOOP	24	0	100.27841	11.207119
E10_F_FBI_GEM	EXPL	LBE	FW_BIN_ITP	GLOBAL		ERR	10	EMPTY	FULL	MDD	24	0	127.63092	15.491634
E100_L_SI_GEM	EXPL	LBE	SEQ_ITP	GLOBAL		ERR	100	EMPTY	LAZY	MDD	21	0	102.78908	19.867665
E10_L_BBI_GEL	EXPL	LBE	BW_BIN_ITP	GLOBAL		ERR	10	EMPTY	LAZY	LOOP	20	0	55.20328	8.192864
E100_L_SI_GEL	EXPL	LBE	SEQ_ITP	GLOBAL		ERR	100	EMPTY	LAZY	LOOP	20	0	60.20067	8.284473
E10_L_BBI_GEM	EXPL	LBE	BW_BIN_ITP	GLOBAL		ERR	10	EMPTY	LAZY	MDD	20	0	75.02731	15.160927
E100_L_SI_GBL	EXPL	LBE	SEQ_ITP	GLOBAL		BFS	100	EMPTY	LAZY	LOOP	18	0	50.00222	5.743061
E100_L_SI_GBM	EXPL	LBE	SEQ_ITP	GLOBAL		BFS	100	EMPTY	LAZY	MDD	18	0	69.23426	8.776530
E1_L_SI_GEL	EXPL	LBE	SEQ_ITP	GLOBAL		ERR	1	EMPTY	LAZY	LOOP	17	0	44.70907	7.476285
E1_L_SI_GEM	EXPL	LBE	SEQ_ITP	GLOBAL		ERR	1	EMPTY	LAZY	MDD	17	0	53.55874	8.939708

Figure 5.3: CFA benchmark best configurations

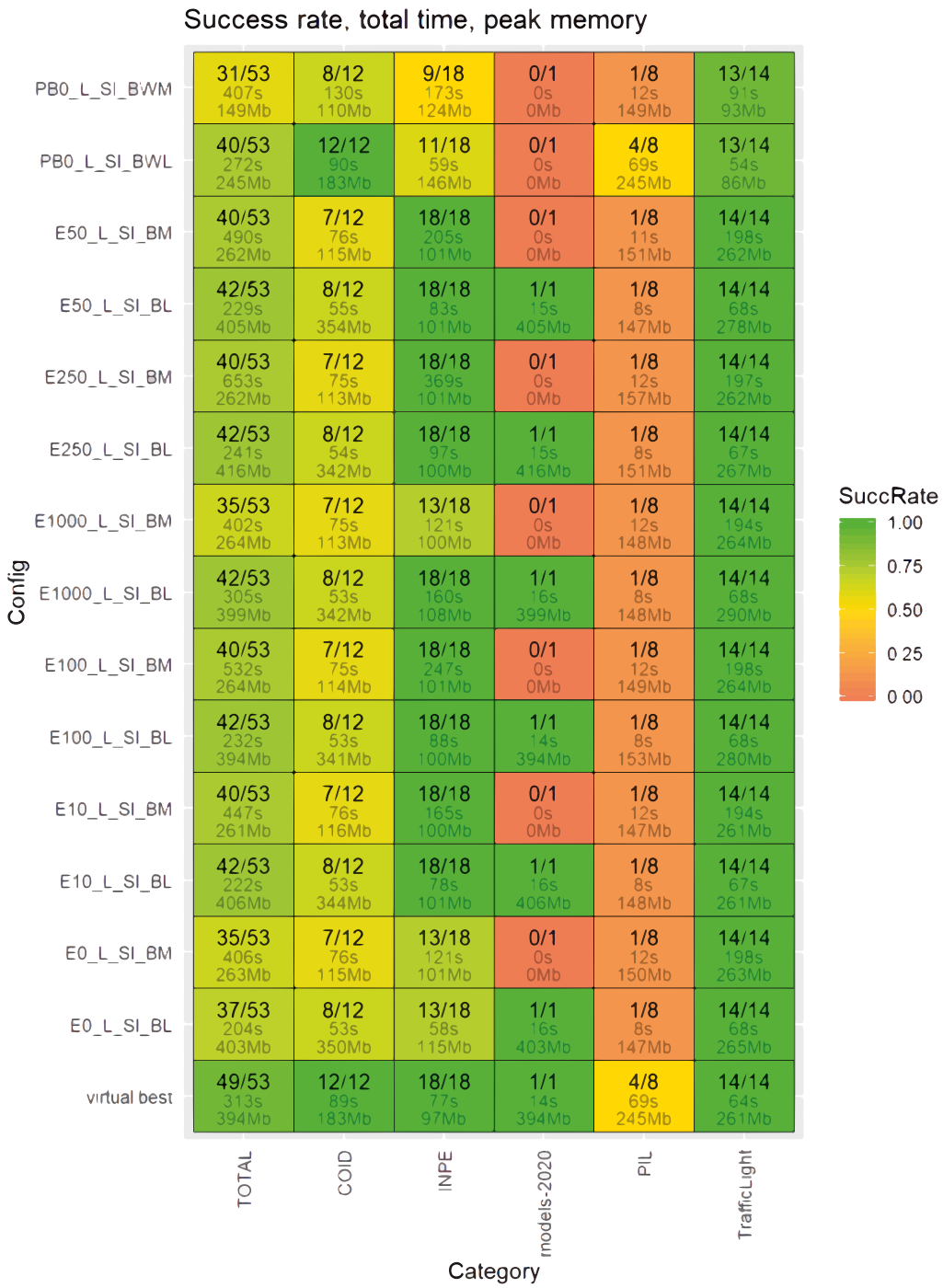

Figure 5.4: XSTS benchmark results

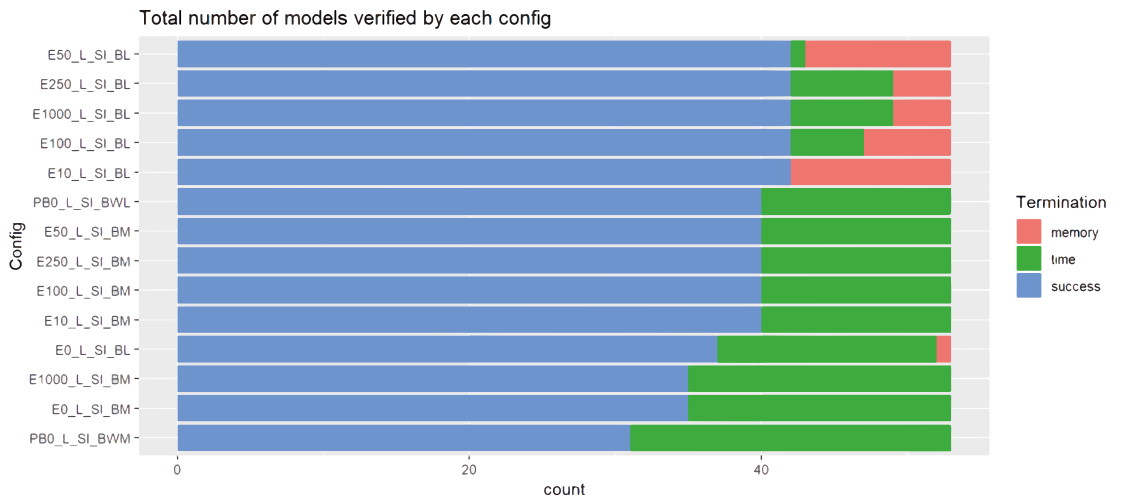

Figure 5.5: XSTS benchmark results

Config	Domain	Refinement	Search	PredSplit	MaxEnum	InitPrec	PruneStrategy	AllSat	SuccCount	Unique	TimeSum	TimeMax
E10_L_SI_BL	EXPL	SEQ_ITP	BFS		10	EMPTY	LAZY	LOOP	42	0	222.3027	15.67595
E50_L_SI_BL	EXPL	SEQ_ITP	BFS		50	EMPTY	LAZY	LOOP	42	0	229.2802	17.12963
E100_L_SI_BL	EXPL	SEQ_ITP	BFS		100	EMPTY	LAZY	LOOP	42	0	232.1800	15.01045
E250_L_SI_BL	EXPL	SEQ_ITP	BFS		250	EMPTY	LAZY	LOOP	42	0	241.3725	16.23604
E1000_L_SI_BL	EXPL	SEQ_ITP	BFS		1000	EMPTY	LAZY	LOOP	42	0	305.0867	21.24581
PB0_L_SI_BWL	PRED_BOOL	SEQ_ITP	BFS	WHOLE	0	EMPTY	LAZY	LOOP	40	7	271.6208	30.37744
E10_L_SI_BM	EXPL	SEQ_ITP	BFS		10	EMPTY	LAZY	MDD	40	0	447.4623	82.72166
E50_L_SI_BM	EXPL	SEQ_ITP	BFS		50	EMPTY	LAZY	MDD	40	0	490.3466	86.37771
E100_L_SI_BM	EXPL	SEQ_ITP	BFS		100	EMPTY	LAZY	MDD	40	0	532.2481	85.76752
E250_L_SI_BM	EXPL	SEQ_ITP	BFS		250	EMPTY	LAZY	MDD	40	0	653.3514	84.86327

Figure 5.6: XSTS benchmark best configurations

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work described the novel data structure of substitution diagram. The main motivation behind the solution is to propose a new, more flexible approach for solving the AllSAT/AllSMT problem in model checking. The decision diagram-based method offers a compact structure in contrast with the one-by-one listing of solutions, enhanced with a strategy to call SMT solvers tailored to the structure of the diagram. The other main advantage is the wide range of potential use cases coming from the increased flexibility.

We designed, formalized and implemented the proposed solution in the Theta model checking framework as another configuration point. We have evaluated its correctness and performance in various experiments involving control flow automata and extended symbolic transition systems. As expected, the approach has a comparably high overhead when used with predicate abstraction, where the number of solutions is generally low, and is comparable to the naive loop-based solution when used with explicit-value abstraction. We note that only two use cases have been evaluated, where the original solution is also applicable, but the low overhead promises a competitive solution when it comes to more complex use cases such as decision diagram-based model checking.

6.2 Future work

We divide future directions into two groups. On the short term, the existing implementation can be improved possibly significantly. During the experiments, the idea of reusing existing substitution diagrams for multiple AllSAT/AllSMT instances occurred. In certain cases, when similar problems have to be solved multiple times, this could lead to an improvement in execution time at the cost of some additional memory usage.

On the longer term, further theoretical evaluation and a more thorough analysis of experiment results could help in better identification of configurations where the solution works best. Implementing the remaining two use cases will also provide interesting results, but the integration with decision diagram-based symbolic model checking is a similarly large task in itself.

Related to solvers, it would be interesting to try the ideas presented in [13] (using cutsets to identify nodes). Also, getting access to the internal data structures of solvers could further enhance the strategy of computing solutions by reusing computed (partial) results.

Acknowledgements

This work was partially supported by the ÚNKP-20-1 New National Excellence Program of the Ministry of Innovation and Technology. We would like to thank Ákos Hajdu for the valuable help and suggestions related to Theta.

Bibliography

[1] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. DOI: 10.1007/3-540-49059-0_14. URL https://doi.org/10.1007/3-540-49059-0_14.

[2] Gianfranco Ciardo, Robert M. Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *Int. J. Softw. Tools Technol. Transf.*, 8(1):4–25, 2006. DOI: 10.1007/s10009-005-0188-7. URL https://doi.org/10.1007/s10009-005-0188-7.

[3] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. ISBN 978-0-262-03270-4. URL http://books.google.de/books?id=Nmc4wEaLXFEC.

[4] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient smt solver. volume 4963, pages 337–340, 04 2008. DOI: 10.1007/978-3-540-78800-3_24.

[5] Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, Online first, 2019. ISSN 0168-7433. DOI: 10.1007/s10817-019-09535-x. URL https://link.springer.com/article/10.1007/s10817-019-09535-x.

[6] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[7] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. ISBN 978-3-662-50496-3. DOI: 10.1007/978-3-662-50497-0. URL https://doi.org/10.1007/978-3-662-50497-0.

[8] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 113–116. ACM, 2018. DOI: 10.1145/3183440.3183489.

[9] Q. Phan and P. Malacaria. All-solution satisfiability modulo theories: Applications, algorithms and benchmarks. In *2015 10th International Conference on Availability, Reliability and Security*, pages 100–109, 2015. DOI: 10.1109/ARES.2015.14.

[10] Gyula Sallai and Tamás Tóth. Boosting Software Verification with Compiler Optimizations. In *Proceedings of the 24th PhD Mini-Symposium*, pages 66–69, Budapest, Hungary, January 2017. Budapest University of Technology and Economics, Department of Measurement and Information Systems. DOI: 10.5281/zenodo.291903. URL https://doi.org/10.5281/zenodo.291903.

[11] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000. DOI: 10.1007/3-540-40922-X_8. URL https://doi.org/10.1007/3-540-40922-X_8.

[12] SV-COMP. Competition on software verification, 2019. URL https://sv-comp.sosy-lab.org/2019/.

[13] Takahisa Toda and Takehide Soh. Implementing efficient all solutions SAT solvers. *ACM Journal of Experimental Algorithmics*, 21(1):1.12:1–1.12:44, 2016. DOI: 10.1145/2975585. URL https://doi.org/10.1145/2975585.

[14] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102257.