JOHANNES KEPLER
UNIVERSITY LINZ

Author
Christoph Sperl

Submission
Institute for Formal Models and Verification

Thesis Supervisor
Univ.-Prof. Dr. Armin Biere

Assistant Supervisor
Dipl.-Ing. Mathias Preiner

May 30th, 2016

Bit-Vector Rewriting using Union-Find with Offsets

Master's Thesis
to confer the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

Satisfiability (SAT) solvers are extensively used in industry to solve various problems by reducing them to the SAT problem. With Satisfiability Modulo Theory (SMT) solvers it is possible to find a satisfying assignment of variables where the problem is specified in first order logic instead of boolean algebra. Rewriting and simplifying the problem on the theory level is an important concept of every SMT solver.

In this thesis we extend the union-find algorithm with offsets to store more information than just equivalences. This extended datastructure maintains dependencies of variables with constant offsets. For specific variables, the algorithm can also infer that two variables are definitely different. We combine this datastructure with automatically generated rewriting rules which propagate the knowledge globally and an algorithm for congruence closure. Furthermore we integrate these concepts to Boolector, a state-of-the-art SMT solver, and can reduce the run-time of the underlying SAT solver in 20 of 22 ASP families.

Kurzfassung

In der Industrie werden viele Probleme auf das Erfüllbarkeitsproblem der Aussagenlogik (SAT) zurückgeführt und Verfahren (SAT Solver) verwendet die dieses Problem effizient lösen. Mit den Solvern der sogenannten Satisfiability Modulo Theory (SMT) ist es möglich eine Variablenbelegung für ein Problem zu finden, dass in Prädikatenlogik erster Stufe formuliert ist anstatt der Boolschen Algebra. Das Vereinfachen des Problems auf Theorieebene ist ein wichtiges Konzept eines jeden SMT Solvers.

In dieser Arbeit erweitern wir die bekannte Union-Find Datenstruktur um einen konstanten Versatz. Für manche Variablen kann diese Datenstruktur sogar feststellen, dass sie definitiv unterschiedlich sind. Wir kombinieren diese Datenstruktur mit automatisch generierten Vereinfachungsregeln die das Wissen propagieren und mit einem Algorithmus zum Auffinden der Kongruenzrelation. Weiters integrieren wir diese Konzepte in Boolector, einem SMT Solver, und können dadurch die Laufzeit vom internen SAT Solver in 20 von 22 ASP Familien verbessern.

Contents

Chapter 1

Introduction

The first section motivates for rewriting on theory level. Further it explains the structure of the thesis and gives a motivation for each chapter separately. In the second section we highlight the main contributions of the thesis. The outline of the following chapters is summarized in the last section.

1.1 Motivation

Satisifability Modulo Theory (SMT) solvers usually do preprocessing and rewriting on the theory level, then bit-blast the formula and let the underlying satisfiabilty (SAT) solver find a satisfying solution [5, 12]. The SAT solver tries to simplify the formula further before or during the search process [8]. The simplification of the SAT solver is based on boolean algebra and not aware of any problem specific information. The SMT solver does rewriting on the theory level and is able to apply theory specific rewriting strategies. Thus simplification on theory level is as important as simplification on boolean algebra level [17]. For example the formula $a * (-1) = a$ can be simplified to $-a = a$. Then the SMT preprocessor can detect that the formula is unsatisfiable. If there is no rewriting or simplification on SMT level, then the multiplication of the original formula has to be transformed to CNF. Bit-blasting a multiplication for example for a bit-width of 256 leads to dozens of thousands of CNF clauses and variables. In this concrete example, the SMT solver can simplify the expression and already detect that there is no satisfying assignment for the variables. Thus the translation to CNF and the call of the underlying SAT solver can be avoided. In Chapter 2 we give a short overview of SAT, SMT and Boolector. Further we define basic terms and explain the well known union-find algorithm. This knowledge is the basis for the definitions and algorithms in the following chapters.

Typically a SMT solver uses a handcrafted set of rewriting rules which will be applied to simplify expressions [7, 11]. These rewriting rules are not complete and there is the possibility that an important rule is missing. Alexander Nadel implemented an algorithm to automatically generate rewriting rules and integrated this algorithm to Intel's SMT solver Hazel [11]. The set of generated rules is complete with respect to a certain criteria. The SMT solver Hazel with these auto-generated rules was in 20 of 23 ASP families faster than the base version.

As part of a project we analyzed the set of handcrafted rewriting rules of Boolector. Therefore we looked at every possible binary operation where the algorithm from Nadel is able to simplify the expression and checked if Boolector finds the same simplification. For a bit-width larger than two, we found 16 expressions with non-predicate operations and 18 expressions with predicate operations, where the generated rewriting rules simplified the operation, but Boolector was not able to find a simplification. One motivating example is the expression $(-a + 1) = a$. The generated rules simplify this expression to *false*. Boolector is not able to apply any simplification during construction of this expression.

This is the motivation to integrate the automatically generated rules into Boolector. An overview of Alexander Nadels idea [11], our implementation and the difference to his approach are shown in Chapter 3. Unfortunately there is no measurable speedup after the integration of the automatically generated rules into Boolector. The reason is that Boolector only uses these rules when creating the directed acyclic graph (DAG) and only considers local knowledge for the condition of the rules. This motivates us for implementing a datastructure for distinct sets to store equivalences, because Alexander Nadel uses the same datastructure to propagate information globally [11].

The well known union-find datastructe is a good example for a datastructure to store distinct sets. We extend this datastructure by offsets to fit more for our purpose and show this idea in Chapter 4. All our automatic generated rules from Chapter 3 use offsets to describe dependencies of two variables, e. g., variable b is equal to $-a + 1$. Therfore we extended the union-find datastructure to be able to store and query for dependencies with a constant offset. This extension also supports other types of queries, for example for specific variables it is possible to infer that they are definitely different. In Chapter 5 we implemented an algorithm for congruence closure and integrated it into Boolector. This algorithm uses the knowledge of the extended union-find datastructure from Chapter 4 to find congruent operations.

The experimental results are presented in Chapter 6. Our algorithms simplify the formula in theory level, such that the underlying SAT solver is in 20 of 22 ASP families faster than before. Furthermore we show that the number of congruent operations found during the congruence closure algorithm is an indicator for the performance increase of the SAT solver. Unfortunately our rewriting algorithms take more time than before and therefore the overall run-time is not always faster. In Chapter 7 we conclude our thesis and outline the future work.

1.2 Contributions

The main contributions of this thesis are:

- We extend the union-find algorithm with offsets and essentially get the same complexity.
- We integrate this extended union-find algorithm and a congruence closure algorithm into Boolector.
- With the extended union-find algorithm one can also check if two nodes are definitely different.

1.3 Outlook

The remainder of the thesis is organized as follows: In Chapter 2 we define basic terms, explain the well known union-find algorithm and give a short overview of SAT, SMT and Boolector. This knowledge is the basis for the definitions and algorithms in the other chapters. The idea of Alexander Nadel to automatically generate rewrite rules [11] is presented in Chapter 3. Further we give an overview and discuss the problems of our implementation and motivate for Chapter 4. In Chapter 4 we extend the union-find algorithm. First we define a new type of relation and then present the implementation of the extended algorithm, which exactly represents this relation for a set of explicitly specified equivalences. Chapter 5 explains how we implement a congruence closure algorithm for Boolector. The algorithm uses the extended union-find structure of Chapter 4. In Chapter 6 we show the effects of our implementation. In the last chapter we conclude and discuss some future work.

Chapter 2

Preliminaries

In this chapter we define mathematical terminology and notation, explain the well known union-find algorithm and describe basics about Boolector, our SMT solver. This knowledge is needed to fully understand the following chapters, where we extend and build on the basics presented here: In the first section we explain bit-vectors. They are a major part of Boolector and strongly used in this work. The next three sections contain definitions for relations, closure and equivalence classes. These definitions are used to describe the information maintained by the union-find algorithm which is presented in Section 2.5. This algorithm is the basis for our algorithm and datastructure presented in Chapter 4. The last two sections describe basics about SAT, SMT and Boolector.

2.1 Notation of Bit-Vectors

Let b be a bit-vector of constant length n. Thus b is a string of n single bits, where every bit is either 0 or 1. The value of an unsigned bit-vector is the sum of the single bits, where the bit at position i has the weight 2^i. For example `1010` as unsigned bit-vector has the value $8 + 2 = 10$. For signed bit-vectors we use the notation according to the 2's complement, so the most significant bit has the factor -2^{n-1}. For example the signed bit-vector `1010` has the value $-8 + 2 = -6$ [7].

The bit-vector $\neg b$ denotes the bit-wise *negation* or 1's complement of the bit-vector b. To negate a bit-vector is the same as flipping all bits to the other value. We also say that we *invert* a bit-vector if we negate the bit-vector.

Further, $a \oplus b$ is the addition of two bit-vectors a and b of the same bit-width n. A possible overflow of the bit-vector addition will be ignored, thus the result of $a \oplus b$ is again a bit-vector of bit-width n. The subtraction of two bit-vectors is defined as $a \ominus b$ and can be rewritten as the addition by a negative value: $a \oplus (\ominus b)$. A negative value can again be reformulated using the 2's complement: $\ominus b = \neg b \oplus 1$. For example the signed 4-bit bit-vector `0110` has the value 6. To get the bit-vector for -6 we flip all the bits and add 1 and therefore get `1010`.

2.2 Properties of Binary Relations

For the sake of completeness we define common properties of binary relations in this section [6, 14]. These definitions are important and used in the following section and in the core chapters of this thesis. Let R be a binary relation over a set S, i. e., $R \subseteq S \times S$.

Definition 2.1 (Reflexive)**.** *R is reflexive iff the identity of all elements in S is part of R, i. e., $\forall s \in S \colon (s, s) \in R$.*

Definition 2.2 (Symmetric)**.** *R is symmetric iff for all elements (s, t) in R also the swapped element (t, s) is in R, i. e., $\forall s, t \in S \colon (s, t) \in R \rightarrow (t, s) \in R$.*

Definition 2.3 (Transitive)**.** *R is transitive iff for any two elements (s, t) and (t, u) in R with such a common element t follows that the element with the other two values (s, u) is also in R, i. e., $\forall s, t, u \in S \colon (s, t) \in R \wedge (t, u) \in R \rightarrow (s, u) \in R$.*

Definition 2.4 (Equivalence Relation)**.** Any relation which is *reflexive*, *symmetric* and *transitive* is called *equivalence relation.*

Let F be a set of functions where every function $f \in F$ has a fixed arity $arity(f)$, and let S be an arbitrary set, but closed over F, i. e., $\forall f \in F \colon a_1, a_2, \ldots, a_{arity(f)} \in S \rightarrow f(a_1, a_2, \ldots, a_{arity(f)}) \in S$, and let R be a binary relation over S.

Definition 2.5 (Monotonic)**.** R is called *monotonic* iff for every function f in F it holds that if the arguments of two function applications are pairwise an element of the relation, the results of the two function applications as a pair is also an element of the relation, i. e., $\forall f \in F \colon (\forall i = 1 \ldots arity(f) \colon (a_i, b_i) \in R) \rightarrow (f(a_1, a_2, \ldots, a_{arity(f)}), f(b_1, b_2, \ldots, b_{arity(f)})) \in R$.

Definition 2.6 (Congruence Relation)**.** Any equivalence relation which is *monotonic* is called *congruence relation.*

2.3 Closure

Let R be a binary relation over a set S, which stores known equivalences. A common problem is that one wants to know all the equivalences which can be inferred from this known information.

Example 2.7. Consider the following example: The set S contains four elements: $S = \{s_1, s_2, s_3, s_4\}$. Further s_1 is the same as s_2 and s_4 is the same as s_1. This equality information is stored in R, so $R = \{(s_1, s_2), (s_4, s_1)\}$. An example information which can be inferred from this known information is that s_2 is equal to s_4. However, nothing can be inferred for s_3.

Definition 2.8 ($\mathcal{P}$-Closure)**.** A relation is called *$\mathcal{P}$-closure* of R if it is the smallest relation which contains R and where the property $\mathcal{P}$ holds.

Corollary 2.9 (Equivalence Closure)**.** A relation is called *equivalence closure* of R if it is the smallest equivalence relation which contains R [14]. We write R' for the equivalence closure of R.

Example 2.7 (continued). The equivalence closure of R for the small example above is:
$\{(s_1, s_1), (s_1, s_2), (s_1, s_4), (s_2, s_1), (s_2, s_2), (s_2, s_4), (s_4, s_1), (s_4, s_2), (s_4, s_4), (s_3, s_3)\}$

Theorem 2.10. The equivalence closure of R represents exactly the information which can be inferred from the set of known equivalences R.

Explanation of Theorem 2.10. Obviously the equivalence closure R' has to contain the known information R, because it is already known that all the pairs in R are equal. The equality is reflexive, transitive and symmetric, so R' has to be an equivalence relation. For example the trivial relation containing all pairs ($S \times S$) is also an equivalence relation which contains R, but it is in general not the smallest one. It includes all pairs and assumes that all elements are equal, which cannot be inferred in general from R. Therefore R' has to be the smallest equivalence relation, because the reflexivity, symmetry and transitivity are the only properties of the equality.

Corollary 2.11 (Congruence Closure)**.** A relation is called *congruence closure* of R if it is the smallest congruence relation which contains R [14].

2.4 Equivalence Class

Let R be a binary relation over a set S and let R' be the equivalence closure of R. From the definition follows that R' is an equivalence relation over S. Thus, the set S can be split into equivalence classes according to the equivalence relation R' [16]:

Definition 2.12 (Equivalence Class)**.** The equivalence class of $s \in S$ according to the equivalence relation R' is defined as the set of elements which are equivalent to s according to R', i. e., $\{s' \in S \mid (s', s) \in R'\}$.

We write $[s]_{R'}$ for the equivalence class of s according to the equivalence relation R'. Every equivalence relation is reflexive. It follows that every equivalence class is non-empty. Furthermore, all equivalence classes of a given set S are distinct, which means that there is no element $s \in S$ which is part of two different equivalence classes [16].

Example 2.7 (continued). The equivalence classes of R', where R' is the equivalence closure of R, are $\{a_1, a_2, a_4\}$ and $\{a_3\}$.

2.5 Union-Find Algorithm

It is a common situation that equalities of certain elements are explicitly specified, but it is necessary to obtain a representation of all equalities implied by those specified ones. Theorem 2.10 says that this representation is exactly the equivalence closure of the specified equalities. The set of elements is again called S and the relation for the explicitly specified equalities is specified as $R \subseteq S \times S$. Further, the equivalence closure of R is called R'.

In this section we describe a common datastructure for representing the equivalence closure R' of known equalities R and an algorithm for maintaining and querying this datastructure, called *union-find* algorithm [14].

2.5.1 Top Level View

The algorithm starts with an empty relation R and offers two operations:

- $union(a, b)$: This operation adds (a, b) to the relation R.
- $find(a)$: This operation returns the representative of the equivalence class of a according to the equivalence relation R'. The representative of an equivalence class is a unique element of this class and does not change between two calls of the *union* operation. With this operation one can check if two elements a and b are equal, by comparing the two representatives of their equivalence classes, i. e., $(a, b) \in R'$, iff $find(a) = find(b)$.

2.5.2 Internal Structure

Internally, every element of the set S keeps a pointer towards the representative of its equivalence class. If an element itself is the representative of its equivalence class, then this pointer is `null` or in some implementations points to itself. Initially no equivalences are explicitly specified: R is the empty set, so every element is in its own equivalence class. In other words, every element is the representative of its own class. Therefore all pointers are initialized with `null`.

The $find(a)$ operation returns the representative of the equivalence class. So the operation simply follows the pointer until it finds an element where this pointer is `null`. Then this is the representative. Note that these pointer do not have cycles.

For every call of the operation $union(a, b)$, the two equivalence classes of a and b will be merged. First the representatives of both elements are needed: $a' = find(a)$ and $b' = find(b)$. If they are not equal, i. e., $a' \neq b'$, either the pointer from a' is set to b' or vice versa. If an element is considered to be a node and a pointer to be a directed edge, then the datastructure represents a forest: Every equivalence class is a tree with the representative as root node. Every $union(a, b)$ operation merges the two trees containing a and b and the $find(a)$ operation returns the root node of a [14]. Figure 2.1 shows two examples for the internal forest from Example 2.7.

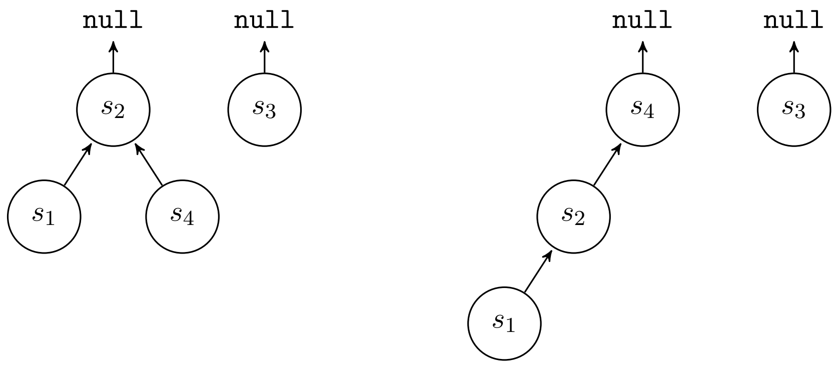

Figure 2.1: Two examples for the internal forest from Example 2.7. The difference is in the last call of *union* with the argument (s_4, s_1). In the left forest the element s_4 points towards s_2, which is the representative of s_1. In the right forest the pointer points the opposite direction: From s_2 to s_4.

2.5.3 Complexity

Let n be the number of elements in the set S. Every element has to store only one additional pointer. So the algorithm has a linear space complexity $\Theta(n)$. The initialization of the pointers has a linear time complexity $\Theta(n)$. The $find(a)$ operation follows the path from a to the root element. The time complexity of this operation depends on the length of the path, i. e., the height of the tree. The $union(a, b)$ operation calls the $find$ operation twice and sets one pointer. Therefore it has the same time complexity as $find$. The height of the tree is in the worst case the number of elements if the tree is a list [14]. This worst case scenario can be generated if the list is always merged with a single element in a way that the root of the list points to this element. This leads to the worst case time complexity of $\Theta(n)$ per operation. Fortunately, there are two ways how the run-time complexity of the algorithm can be improved as shown in the next two sections.

2.5.4 Optimization: Balancing

One way to optimize the run-time complexity of this algorithm is by balancing the trees: Whenever two trees are merged, the algorithm can decide whether either the root element of the first tree should point to the root element of the second tree or vice versa. If the algorithm decides in a way that the root element of the tree with the smaller height always points to the root element of the tree with the larger height, then the height of the merged tree will only increase by one if both heights are equal. This means that a tree of height h was constructed by merging two trees of height $h - 1$. Each of the two trees of height $h - 1$ was again constructed by merging two trees of height $h - 2$. Therefore the height of the tree is logarithmic in the number of elements in the tree. This optimization reduces the time complexity of both operations *union* and $find$ from a linear complexity $\Theta(n)$ to a logarithmic complexity $\Theta(log(n))$, where n is the number of elements in the set S [18].

2.5.5 Optimization: Path Compression

Another optimization is to compress the path from an element to the root of the tree when it is traversed: Every time when $find(a)$ is called, the algorithm traverses all the pointers from a to the root node of the tree. With this optimization, the algorithm traverses this path a second time and redirects the pointers of all visited nodes directly to the root node. In this way the tree will be flatten. For the next call of $find(a)$, the algorithm has to go fewer steps to find the root node of the tree.

The time complexity of m $find$ operations without balancing, but with path compression was calculated by Tarjan and Van Leeuwen [18] to be $\Theta(m*log_{(2+m/n)}(n))$ which is smaller than $\Theta(m*log_2(n))$. Therefore the time complexity for any operation $find$ or *union* is $O(log(n))$. If both optimizations are combined, then the time complexity is $O(m * \alpha(m + n, n))$ for m calls of $find$ and n elements in the set S [18]. The function α is a very slow growing function and can be ignored in practice [4, 14], so with both optimizations the algorithm has almost a constant time complexity for each operation.

2.6 SAT and SMT

The term SAT refers to the propositional satisfiability problem. The problem is to find an assignment for the boolean variables to satisfy a given propositional formula. This formula is usually expressed in conjunctive normal form [7]. The SAT problem is famous because it was one of the first problems which was shown to be NP-complete by Stephen Cook in 1971 [3]. Furthermore many other NP-complete problems are in practice reduced to the SAT problem. The main reason for this reduction is the existence of efficient tools to find an assignment for the variables, called SAT solvers. For more information on SAT see SAT-Handbook [1] and Donald Knuth's recent book about satisfiability [9].

The *satisfiability modulo theories* (SMT) problem is a satisfiability problem for formulas in first order logic with some theories. This thesis has its focus on theory of fixed-size bit-vectors. With the first order logic one can specify the problem in a higher level as the boolean algebra [1, 7]. A SMT solver is used to find an assignment for the bit-vector variables to solve a given formula. The complexity of an SMT formula using the theory of fixed-size bit-vectors is NExp-Time-complete [10].

2.7 Boolector, our SMT Solver

Boolector[1] is a state-of-the-art SMT solver, developed at the institute for Formal Models and Verification at the Johannes Kepler University in Linz. This SMT solver is very successful in international competitions. In the SMT Competition 2015, Boolector won first places in three out of five tracks. The current version 2.2.0 supports the theories of fixed-size bit-vectors and arrays, but no quantifiers (QF_ABV) [12].

Internally, Boolector represents an expression as a *Directed Acyclic Graph* (DAG), where an edge from node x to node y represents that node y is an input of node x. A sink node of the DAG has no outgoing edges. In other words, it has no inputs and is either a variable or a constant. A node with outgoing edges is an operation. Every operation has a fixed number of inputs. For example the `add` operation has exactly two inputs. If we need to add three nodes, we have to cascade two `add` operations. Furthermore there exist no operations for inverting a node, instead every input can be used in a normal way or inverted, i. e., every edge of the DAG points to another node and also stores the phase with it. We call the combination of a pointer to a node and a phase *node pointer*. Boolector holds a list of asserted node pointers, i. e., every node pointer of this list has to evaluate to *true*, which is the same as the one-bit bit-vector `1`.

Example 2.13. Consider the following expression: $(a \oplus b < 5) \wedge (a \ominus b \geq 10)$ where $<$ and $\geq$ denote an unsigned comparison, $\oplus$ and $\ominus$ are bit-wise addition and subtraction and a and b are 4-bit variables. The nodes in Boolector will be created bottom-up: First the variables a and b have to be created before the sum of those variables can be created. The constant 5 represented as a 4-bit bit-vector is `0101`. To save constant nodes and therefore memory, Boolector does not have constant nodes where the least significant bit is `1`. Instead, all nodes are flipped and the resulting constant node is used inverted. The constant 5 is therefore expressed as $\neg$(`1010`). Furthermore the whole expression, which is in this example represented by the top-level `and` operator, will be asserted. The list of

[1]http://fmv.jku.at/boolector/

asserted node pointers already represents a logical `and`. Therefore an `and` node will never be added to the assertion list. Its inputs will be added instead. To save more memory, nodes are shared, which means that the same node can be used several times as an input for another node. Boolector also avoids redundant operators, so there is no operator for *greater or equal* ($\geq$). In this case the *less than* operator ($<$) with an inverted result is used. Similarly there is no subtraction operator, because subtraction is nothing else than addition with a negative value and the negative value of a bit-vector is defined by its 2's complement. Figure 2.2 shows the directed acyclic graph of the expression of this example.

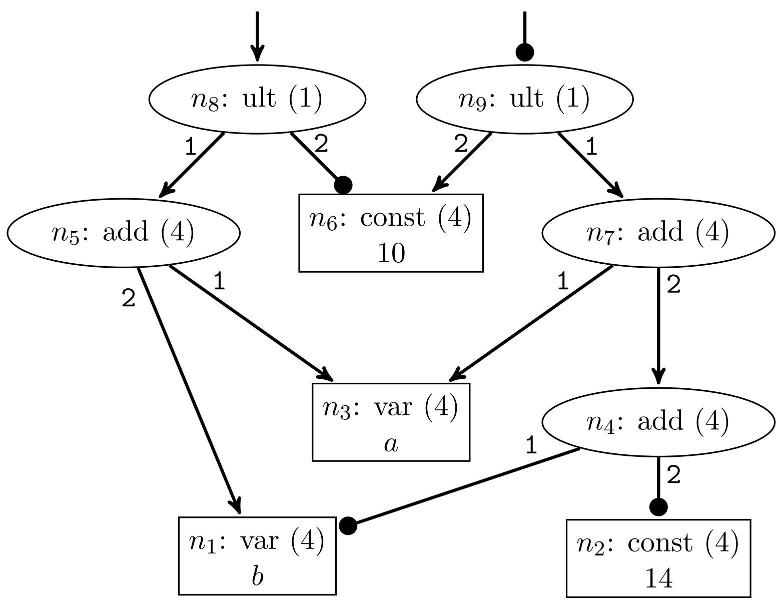

Figure 2.2: The directed acyclic graph from Example 2.13. A rectangular node represents a variable or a constant where an ellipse represents an operation. Every edge is a node pointer, a circle at the end of an edge means to use this node inverted and an arrow means to use it as it is. The description of each node contains a unique label, the type of the node and the bit-width in braces. The inputs of the nodes are ordered and therefore marked with a number, since not every operation is commutative. The two edges at the top are asserted node pointers.

During construction of the DAG only simple predefined rewriting rules will be applied, for instance instead of $a \oplus 0$, directly a will be used. When the `sat` function is called to check if the expression is satisfiable, more complicated rewriting strategies, for example term substitution, are applied to simplify the DAG further. Then the DAG will be translated to CNF and the internal SAT solver will be used to check if the expression is satisfiable [2]. Some details, for example array handling, are not discussed here. This thesis has its focus on improving rewriting of plain bit-vector expressions without arrays.

Chapter 3

Automatic Rule Generation

For SMT solvers it is common to have a handcrafted set of rewriting rules which will be applied to simplify the input formula [7, 11]. The problem with this approach is that one can never be sure that the handcrafted set does not miss an important rewriting rule. In this chapter we will explain the idea of Alexander Nadel [11] to automatically generate a restricted, but complete set of rewriting rules. Afterwards we show how we have implemented his idea and discuss the problems of our implementation.

3.1 Idea

Alexander Nadel describes in [11] an approach where he automatically generates a restricted set of rewriting rules. The following subsections give an overview of his idea.

3.1.1 Scope of Rules

In theory there are infinitely many rewriting rules, so a restriction is necessary to get the interesting rules only. Nadel [11] limits the scope for the auto-generated rewriting rules in several ways: First of all, only binary operations with equal input bit-width are interesting. The result is called x and the two inputs are called y and z. Together they build a *triplet* of the form:

$$x = op(y, z)$$

There are two kind of binary operations with equal input bit-width: *predicate operations* and *non-predicate operations*. Predicate operations always have a result bit-width of 1 and non-predicate operations have the same bit-width for the result and the input variables. The operation of a triplet where all three variables are of bit-width 1 is also a non-predicate operation [11].

A rewriting rule in general consists of two parts:

- *Condition*: The part which has to be fulfilled to apply the rewriting rule.
- *Result*: The result after applying the rewriting rule.

In Nadels approach the condition of a rewriting rule is the information about a triplet and consists of three parts:

- *Operation type*: The operation which will be applied to the two input variables.
- *Bit-width*: The bit-width of the input variables. The bit-width of the output variable x is indirectly given by the operation type and the bit-width of the input variables.
- *Premise*: Certain information about the three variables of the triplet: Each of them can be a constant, depend on another variable or be an independent variable.

The result of the rewriting rule is of the same structure as the premise, because it also contains information about the three variables: After applying the rewriting rule, each variable can again either be a constant, depend on another variable or be an independent variable.

The set of constants which are considered by Alexander Nadel [11] is called RC and limited to the following 5 values: `-2`, `-1`, `0`, `1` and `2`. This means, if a variable is a constant, but the constant is not in RC then the variable is not treated as a constant. Dependent of the bit-width of the variables, not all constants in RC can be expressed. For example a 1-bit variable can only store the constants `0` and `-1`.

If a variable is dependent on another variable, then the dependency is expressed with a lambda rewriting function. For example if x is the same as $-y-1$ then the rewriting function for x is $\lambda e.\ -e-1$ with the additional information that x depends on y. The set of rewriting functions RF is limited to the following ten functions:

$$\begin{array}{lllll} \lambda e.\ e-2 & \lambda e.\ e-1 & \lambda e.\ e & \lambda e.\ e+1 & \lambda e.\ e+2 \\ \lambda e.\ -e-2 & \lambda e.\ -e-1 & \lambda e.\ -e & \lambda e.\ -e+1 & \lambda e.\ -e+2 \end{array}$$

In this explanation we use a slightly different notation than Alexander Nadel [11]. Nadel expresses these rewriting functions as named functions, but we use anonymous lambda expressions.

Again, if a variable is dependent on another one, but the dependency cannot be expressed by one of these rewriting functions, then the variable is not considered to be dependent on another variable.

To reduce redundancy, it is not allowed that y is dependent on x or that z is dependent on any other variable, because these dependencies can also be expressed the other way around. Similar to the constants, not all dependencies can be expressed by all bit-widths. Furthermore, for predicate operations there does not exist the case where the variable x is dependent on an input variable, because the bit-widths do not match.

It is possible that both of these properties (constant and dependency) apply to a variable at the same time. Therefore the properties are ordered by their importance and the premise only stores the most important one.

1. The property of being a *constant* is most important.
2. If a property is not a constant, it can be *dependent* on another variable.
3. If 1. and 2. do not apply, then the variable is considered to be an *independent* variable.

Example 3.1. Let $x = \texttt{and}(y, z)$ be a triplet where `and` is the bit-wise *and* operation. All variables are 4-bit variables and y is equal to z.

In this example the premise will contain the following information:

- x is an independent variable.
- y is dependent on z using the rewriting function $\lambda e.\ e$.
- z is an independent variable.

Together with the other information from the condition (`and`-operation and a bit width of 4) one can generate a rule with the following result:

- x is dependent on z using the rewriting function $\lambda e.\ e$.
- y is dependent on z using the rewriting function $\lambda e.\ e$.
- z is an independent variable.

In this example, the result is equal to the premise, except for the variable x. The premise treated this variable as independent variable, but the result shows that x is the same as z.

3.1.2 Rule Generation and Verification

This subsection describes the approach of Alexander Nadel [11] to automatically generate rewriting rules and why it is guaranteed that they are correct.

Generating a rewriting rule means, finding the result for a given condition. The result consists of the three single results for the three variables. The idea is therefore to find the result for each variable independent of the result for the other variables. Finding a result for a single variable is done by trying all possibilities for this variable, starting with the most important property.

In case of variable x, the algorithm first tries if x can be rewritten to a constant value. Afterwards it checks if x can be simplified to a dependency on y or z. Checking if the variable x can be rewritten to a value v is the same as checking if x is equal to this value v in every situation, i. e., for every possible value of y and z which is consistent with the premise. A SMT solver is used to do this check: The three variables are created in a new SMT solver instance. Then the information of the premise is added to the solver instance. Further x is asserted to be not equal to the value v. Finally the triplet $x = op(y, z)$ is added and the SMT solver is called to find an assignment for the variables. If the formula is unsatisfiable then the check is successful and x is equal to v in every case. Therefore x can be rewritten as v, so v is the result for x for the given condition. This procedure is also done for the other two variables. The difference is that there exist less possible result values for the variables y and z, because they do not have so many dependency options. With this strategy it is guaranteed that the rule is correct, because an SMT solver verified the single results.

3.1.3 Rule Application

Alexander Nadel uses equivalence classes in Intels SMT solver *Hazel* to store all the information and a 0-saturation algorithm to propagate all the information through the set of triplets. After merging two equivalence classes, Hazel traverses all triplets whose representatives have changed and checks if one of the rewriting rules can be applied to get new information [11].

Hazel has no pregenerated rewriting rules. All the needed rules are generated at run-time and cached for reuse. In this way only the rules which are needed for the given problem will be generated.

According to Nadel [11], Hazel achieves much better results in the ASP family from the SMT-LIB with the new algorithm than with the base version: The SMT solver with this new idea outperforms the base version in 20 of 23 ASP families.

3.2 Implementation

In this section we show how we have implemented Alexander Nadels approach and how it differs from his implementation. The implementation is split into three parts: First the condition (premise, bit-width and operation type) has to be extracted from the DAG. Afterwards the rewriting rules have to be generated for the condition and finally the result of the rewriting rule has to be applied. This section starts with the second part.

3.2.1 Rule Generation

A rule is nothing else than a mapping from the condition to the result. We introduced a new structure for the premise of the condition and the result of the rule. This structure is called `RulegenTripletInfo` and is shown in Listing 3.1. This listing also shows the enum `RulegenTripletType` which is used by the structure and specifies the property of a variable. For simplicity every variable of the triplet uses the same `RulegenTripletType`, even though not every variable can have every property, e. g., the variable z is not allowed to be dependent on y or itself.

```
enum RulegenTripletType
{
  TRIPLET_INDEPENDENT,
  TRIPLET_CONSTANT,            // λe. offset
  TRIPLET_DEPENDENT_Z,         // λe. z + offset
  TRIPLET_DEPENDENT_NOT_Z,     // λe. ¬z + offset
  TRIPLET_DEPENDENT_Y,         // λe. y + offset
  TRIPLET_DEPENDENT_NOT_Y,     // λe. ¬y + offset
};
struct RulegenTripletInfo
{
  RulegenTripletType type_x, type_y, type_z;
  int8_t offset_x, offset_y, offset_z;
};
```

Listing 3.1: `RulgenTripletInfo`: A structure to store the premise and the result of a rewriting rule.

The meaning of the `RulegenTripletInfo` structure is straight forward. For example if `type_x` is set to be `TRIPLET_DEPENDENT_NOT_Y` and `offset_x` is set to be `-1` then it means that x is equal to $\neg y - 1$. In case any type is set to `TRIPLET_CONSTANT` then the offset specifies the constant value. The offset is of type `int8_t`, which is an 8-bit signed integer. Therefore we could specify constants and dependency offsets from -128 to 127. Instead we limit the offset variables always to be between -2 and 2, but this can easily be extended. If a type is set to `TRIPLET_INDEPENDENT` then the offset has no meaning and will be ignored.

This is already the first difference to Alexander Nadels approach presented in Section 3.1: Nadel uses rewriting functions with normal or negative variables and an offset of ± 2. We convert every negative dependency on another variable to a negated dependency and also have symmetric offset limits. Our motivation to use negated dependencies instead of negative dependencies is to ease the integration to Boolector. In Section 2.7 is mentioned that Boolector uses node pointers (a combination of a pointer to a node and a phase) to reference nodes. So it is very natural to have negated nodes in Boolector. The following table highlights the different rewriting functions which can be expressed by our implementation and by Nadels approach. The rewriting functions where the node is not negated or inverted are not shown, because both implementations have in this case exactly the same functions. All entries in the same row are equal. If a column has an entry in the specified cell then the implementation is able to express this rewriting function.

Nadels approach		Our implementation
		$\lambda e.\ \neg e - 2$
$\lambda e.\ -e - 2$	$=$	$\lambda e.\ \neg e - 1$
$\lambda e.\ -e - 1$	$=$	$\lambda e.\ \neg e$
$\lambda e.\ -e$	$=$	$\lambda e.\ \neg e + 1$
$\lambda e.\ -e + 1$	$=$	$\lambda e.\ \neg e + 2$
$\lambda e.\ -e + 2$		

Nadel's approach calculates all the rules at run-time and caches them [11]. We have a more flexible approach: A mixture of precalculated rules and generation at run-time. The precalculated rules are usually all rules up to a certain bit-width, because it is common that small bit-widths occur in problems. The best example here is the bit-width 1 which has to be part of every problem since we can only assert booleans. It is also possible to disable the generation of rules at run-time and only use the precalculated rules. Then the application of the automatically generated rules will be skipped if there are no precalculated rules for a occurring bit-width.

3.2.2 Find Condition

The easiest way to apply rewriting rules is at creation-time of the DAG. The DAG is created bottom up, so the two inputs of a binary operation already exist when the operation node is created. However, it is not known how the result will be used. Therefore `type_x` of the premise is always `TRIPLET_INDEPENDENT` in our case.

To find the premise for y and z, our algorithm first checks if the nodes are constants. Afterwards it checks if y is dependent on z, by looking at the structure of the DAG. Only the following four cases are taken into account:

- y is the same node as z.
- y is an `add`-node with z and a constant node as input.

- z is an `add`-node with y and a constant node as input.
- y and z are both `add`-nodes with a common node as input. The other input of the two operations has to be a probably different constant node.

All these cases are checked with both phases of y and z, e. g., the first case also covers the situation that y is equal to $\neg z$.

3.2.3 Apply Rule

The last subsection described how the premise can be extracted from the DAG at creation-time of an operation. The variable x is in this case always an independent variable, because at this point in time it is not known how the result of the operation will be used. The rewriting rules are a mapping from the condition (premise + bit-width + operation type) to the result. So, after the premise is extracted from the DAG, the result can be calculated.

The result consists of the single results for the three variables. In the premise, the variable x is always an independent variable. If the result of an operation is unknown, it is impossible to infer any information for the input variables. As a consequence of this, the result of y and z will be equal to the premise of y and z. The only information which can be gained from the rewriting rule is stored in the result of the variable x. If `type_x` is `TRIPLET_INDEPENDENT`, then there is no simplification possible with the rewriting rule and the common handcrafted rewriting rules are applied. Otherwise the result of x represents the simplification of $op(y, z)$. Therefore, instead of the original binary operation, the simplification of x is created. This can either be a constant or a dependency on y or z by a small offset.

3.3 Related Work

Besides Alexander Nadel [11] also Trevor Alexander Hansen [7] experimented with automatically generated rewriting rules. Hansen used a slightly different approach: The main idea is to generate all possible expressions up to a certain depth and check which pairs are equal. In this way one finds rules how one expression can be rewritten into another one. A rule is only considered as a rewriting rule if the result is simpler than the condition. Hansen implemented this idea for a depth of two, e. g., a binary operation has two binary operations as children where each of them have again two nodes as children. Hansen describes his approach as less successful, because most of the generated rules contain a large portion of constants and are never used in practice, although the number of allowed constants is also limited [7]. We think another reason why the generated rules from Hansen are less usable in practice is the fact that most of his generated rules contain shift operators. In his thesis [7] he presents 20 randomly selected rules, 14 of them contain at least one shift operator in the condition of the rule.

3.4 Discussion

The way how we use the auto generated rules is not optimal and therefore there is no measurable performance improvement. There are three simple reasons for this:

1. Boolector has naturally already a good set of static rewriting rules which covers probably the most important cases.
2. In our implementation only local knowledge is taken into account, i. e., to investigate if a node is dependent of another one, only four simple situations are checked. All these situations consider `add` operations and their inputs only. This is very restrictive. For example it could be asserted that the two input nodes of an operation have to be equal. However, this knowledge is not used in the presented approach.
3. In our implementation we do not use the full potential of the generated rules: The rules work for triplets, but we only use two of the three variables. We always set the variable x to be independent and therefore only use a fraction of all possible premises.

The first reason will always be a an issue, since it is difficult to get a huge performance improvement for a fast state-of-the-art SMT solver.

To avoid the second reason, we need a different approach to check how two nodes depend on each other. To look at the structure and check if one node is an addition of the other and a constant node is not enough. A good idea is to use a datastructure to store the global knowledge of equivalences. An option would be a datastructure to maintain the equivalence closure, e. g., the union-find datastructure.

Concerning the third reason, an improvement would be to use the variable x of the rewriting rules as well. This is not possible when the nodes are created. Therefore the DAG has to be traversed afterwards. When the DAG is fully created it is known how the result of an operation will be used and we can take this knowledge into account. Then the `type_x` of the premise is not always `TRIPLET_INDEPENDENT` anymore and then it is also possible to propagate knowledge top down.

Alexander Nadel uses in [11] equivalence classes to store all the equivalences. Unfortunately it is not specified how the algorithm can extract the information if two nodes differ by a small offset from the equivalence class structure. Exactly this information is necessary to find the premise for the rewriting rules. For example the union-find algorithm – a well known algorithm for equivalence classes – only supports to query whether two nodes are equal. Furthermore it is not possible to store the information that two nodes differ by an offset in this datastructure [4], but this information is a common result when applying the presented rewriting rules.

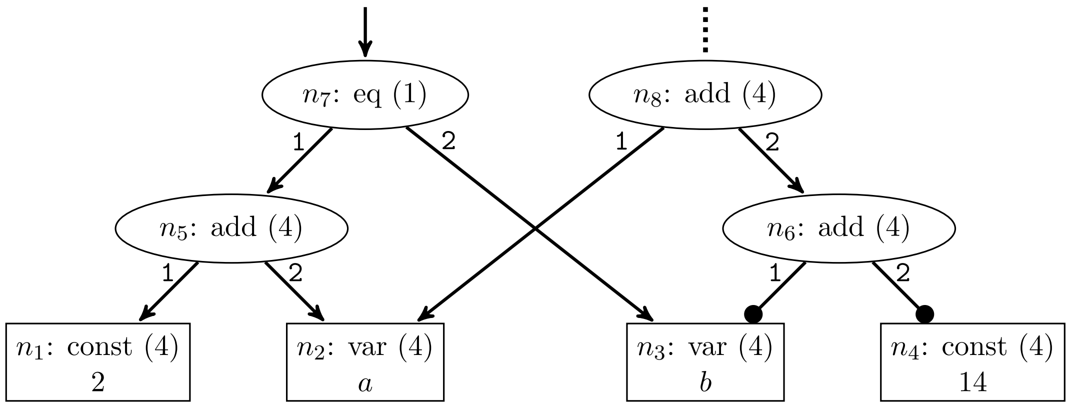

Figure 3.1: An example to show the limits of our implementation presented in this chapter.

Example 3.2. Figure 3.1 shows an example where our implementation presented in this chapter would not find the simplification for node n_8. It is asserted (node n_7) that $a + 2 = b$. Therefore $a - b$ as calculated in node n_8 simplifies to the constant -2. It is also not possible to find this simplification with a normal union-find algorithm which only stores equivalences. This algorithm would store that node n_5 is equal to node n_3, but would not be able to recognize and save that the variables a and b differ by two. This is exactly the knowledge we need for the premise of the rewriting rule, which simplifies n_8 to the constant -2. It is unclear how exactly this information is found [11].

Therefore we extend the concept of equivalence closure by integer offsets and present an extended version of the union-find datastructure to maintain this knowledge in the next chapter. With this approach we are able to simplify node n_8 of Figure 3.1 to the constant 2. Also a global substitution of variable b by the expression $a + 2$ would simplify the node n_6 to $-a - 2$ and further simplify n_8 to -2.

Chapter 4

Union-Find with Offsets

In order to find the premises for the rules in Chapter 3, we want a global structure which stores the information how some nodes are related to each other. The union-find algorithm stores equality information of nodes by maintaining the equivalence closure of some known equivalences. The disadvantage of the union-find algorithm is that one cannot take other information than equivalences of two nodes into account or query for them. For example if two nodes differ by one, it is not possible to use this information in the algorithm to gain knowledge. This information that two nodes differ by a small amount is exactly what is needed for generating the premise of the rule generation algorithm in Chapter 3. Furthermore the result of the rule generation algorithm is again the information that two nodes differ by a constant offset. Therefore we extended the union-find algorithm by constant offsets and present this idea in the following chapter.

4.1 Mathematical Background

The union-find algorithm maintains the equivalence closure of some known equivalences. In this section we will extend our known information by the concept of "offsets" and introduce the terms *offset equivalence relation* and *offset equivalence closure*. The next section then shows how the union-find algorithm is extended to maintain the offset equivalence closure of some known information.

4.1.1 Offset Pointer

Let N be a set of nodes with bit-width bw. Boolector uses each node either inverted or non-inverted (normal). We call such a reference *node pointer*:

Definition 4.1 (Node Pointer)**.** A *node pointer* p is the combination of a reference to a node $n \in N$ with a phase. It simply stores the two ways how a node can be used: Either normal (n) or inverted ($\neg n$).

The set of node pointers is called P: $P = N \cup \{\neg n \mid n \in N\}$. The possible bit-vector constants depend on the bit-width. The set of all possible constants for bit-width bw is called C_{bw}: $C_{bw} = \mathbb{N} \cap [0, 2^{bw} - 1] = 0, 1 \dots 2^{bw} - 1$. We also write C instead of C_{bw} when the bit-width is given by the context.

We now combine a node pointer with a constant:

Definition 4.2 (Offset Pointer)**.** An *offset pointer* $s = p \oplus c$ is the bit-vector addition of a node pointer $p \in P$ with an offset $c \in C$. The constant and the node have the same bit-width. We define S as the set of all offset pointers: $S = \{p \oplus c \mid p \in P, c \in C\}$. An offset pointer represents a node pointer with an offset. For every element $p \in P$ there is a trivial offset pointer which represents p: $(p \oplus 0)$. If the offset is 0, we can also omit the offset. Therefore $p \oplus 0$ is the same as p.

Listing 4.1 shows how the node pointer is represented in an implementation. A `NodePointer` consists of a pointer to a node and the information whether it is inverted, e. g., using a kind of "sign-bit". The pointer to a node can also be `null`. The structure `OffsetPointer` extends this the node pointer by an offset. The type `Offset` is a data type which can store an offset for the same bit-width as the node.

```
struct NodePointer {
  Node* node;
  bool is_inverted;
};

struct OffsetPointer {
  NodePointer node_pointer;
  Offset offset;
};
```

Listing 4.1: The definitions of the types `NodePointer` and `OffsetPointer` in pseudo code.

In Boolector every constant is represented by a node. To avoid redundancy we always express a constant value c as the offset pointer *zeroNode* $\oplus$ c. The node *zeroNode* is the constant node with all bits set to zero. This means that we never create an offset pointer $p_{const} \oplus c$ where p_{const} points to a different constant node than the *zeroNode*.

In the following we describe how this datastructure is normalized to avoid further redundancy: An offset pointer of the form $\neg$*zeroNode* $\oplus$ c is redundant and is normalized to *zeroNode* $\oplus$ $(c \ominus 1)$.

Furthermore bit-width 1 is a special case, because every node pointer p of bit-width 1 can be inverted either by adding the constant one $(p \oplus 1)$ or by inverting the node pointer $(\neg p)$. To avoid redundancy in this case, an offset pointer does not have any inverted node pointer of bit-width 1.

The function to normalize an offset pointer is shown in Listing 4.2. A non-normalized offset pointer has an inverted node pointer and either points to a constant node or has a bit-width of 1. If the offset pointer points to a constant node, then this node is the *zeroNode*, because we never create other constant pointers. The first two conditions in Listing 4.2 check if the given offset pointer is already normalized. If this is not the case then the algorithm has to invert the node pointer and correct the offset such that the offset pointer is still the same. Therefore it simply subtracts 1 from the original offset. This works, because the negation and the subtraction of 1 is the same for a one-bit variable. For constants it is similar: The node pointer points to $\neg$*zeroNode* which represents -1, but it will be inverted to *zeroNode*. To correct this negation we have to subtract one from the offset.

```
OffsetPointer correct_op (OffsetPointer op)
{
  OffsetPointer result;
  if (!is_constant_node (op.node_pointer.node) &&
      get_bit_width(op.node_pointer.node) != 1)
    return op;

  if (!op.node_pointer.is_inverted)
    return op;

  result.node_pointer.node = op.node_pointer.node;
  result.node_pointer.is_inverted = false;
  result.offset = op.offset ⊖ 1;
}
```

Listing 4.2: A function which returns the normalized version of the given offset pointer.

We can also apply mathematical operations to an offset pointer, because it is nothing else than a node pointer added with a offset of of the same constant bit-width than the node. For example the negation of an offset pointer $s = p \oplus c$ is again an offset pointer:

$$\neg s = \neg(p \oplus c) = \ominus(p \oplus c) \ominus 1 = (\ominus p \ominus c) \ominus 1 = (\ominus p \ominus 1) \ominus c = \neg p \ominus c = \neg p \oplus (\neg c \oplus 1)$$

$$\neg p \in P \wedge (\neg c \oplus 1) \in C \rightarrow \neg p \oplus (\neg c \oplus 1) \in S$$

The negation of an offset pointer as function is shown in Listing 4.3.

```
OffsetPointer invert_op (OffsetPointer op)
{
  OffsetPointer result;
```

```
    result.node_pointer.node = op.node_pointer.node;
    result.node_pointer.is_inverted = !op.node_pointer.is_inverted;
    result.offset = ⊖op.offset;
    return correct_op (result);
}
```

Listing 4.3: A function to invert an offset pointer.

If a constant value is added to an offset pointer, the result will be again an offset pointer. There exist 2^{bw} *different* offset pointers of bit-width bw which can be created by adding different constants $c' \in C$ to an offset pointer.

4.1.2 Offset Equivalence Relation

The binary relation $R \subseteq S \times S$ contains the explicitly specified equivalences. For example if $(a \oplus 0)$ and $(\neg b \oplus 5)$ are equal, then $(a \oplus 0, \neg b \oplus 5)$ is in R. We already know that the equivalence closure R' of our knowledge will be interesting. However, because of our extended definition of the elements in the set S we can add further properties beside reflexivity, symmetry and transitivity which have to hold in R', in order to make it more powerful:

- *Monotonic rule of bit-wise negation:* If two offset pointers are in R', also the negation of both offset pointers have to be in R', i. e., $\forall s_a, s_b \in S\colon (s_a, s_b) \in R' \rightarrow (\neg s_a, \neg s_b) \in R'$.
- *Monotonic rule of constant addition:* If two offset pointers are in R', also the offset pointers created by adding the same constant offset have to be in R', i. e., $\forall s_a, s_b \in S, c \in C\colon (s_a, s_b) \in R' \rightarrow (s_a \oplus c, s_b \oplus c) \in R'$.

Definition 4.3 (Offset Equivalence Relation)**.** We introduce a new relation, called *offset equivalence relation* which has the properties of an equivalence relation (reflexive, symmetric, transitive) plus the properties defined above (monotonic rule of the bit-wise negation and monotonic rule of constant addition). We write R^* for the offset equivalence relation of R.

Therefore the *offset equivalence closure* of some known information R is the smallest offset equivalence relation containing R.

Example 4.4. Let N be the set of nodes: $N = \{a, b, c, d\}$. Every node in N has a bit-width of 2. It follows that $P = \{a, \neg a, b, \neg b, c, \neg c, d, \neg d\}$. C_2 is the set of possible bit-vector constants: $C_2 = \mathbb{N} \cap [0, 3] = \{0, 1, 2, 3\}$. For example, $R = \{(a \oplus 0, \neg b \oplus 1), (\neg d \oplus 3, \neg a \oplus 2)\}$. The equivalence closure of R is the same as R plus all the identity elements. Therefore R' is in this example not more useful than R, but the offset equivalence closure R^* of R is much more informative: From the monotonic rule of constant addition follows that $(\neg d \oplus 1, \neg a \oplus 0) \in R^*$. From the monotonic rule of bit-wise negation follows now that $(d \oplus 3, a \oplus 0) \in R^*$. Now the equivalence properties can be used to show that $[a \oplus 0]_{R^*}$, the equivalence class of $a \oplus 0$, is $\{a \oplus 0, \neg b \oplus 1, d \oplus 3\}$.

Here is a list of all equivalence classes according to the offset equivalence closure:

$$\begin{array}{llll}
\{a \oplus 0, \neg b \oplus 1, d \oplus 3\} & \{\neg a \oplus 0, b \oplus 3, \neg d \oplus 1\} & \{c \oplus 0\} & \{\neg c \oplus 0\} \\
\{a \oplus 1, \neg b \oplus 2, d \oplus 0\} & \{\neg a \oplus 1, b \oplus 0, \neg d \oplus 2\} & \{c \oplus 1\} & \{\neg c \oplus 1\} \\
\{a \oplus 2, \neg b \oplus 3, d \oplus 1\} & \{\neg a \oplus 2, b \oplus 1, \neg d \oplus 3\} & \{c \oplus 2\} & \{\neg c \oplus 2\} \\
\{a \oplus 3, \neg b \oplus 0, d \oplus 2\} & \{\neg a \oplus 3, b \oplus 2, \neg d \oplus 0\} & \{c \oplus 3\} & \{\neg c \oplus 3\}
\end{array}$$

4.2 Extended Union-Find Algorithm

We want to use a similar version of the union-find algorithm to maintain the offset equivalence closure of our knowledge.

Similar to the normal union-find algorithm we build a forest with the nodes. The idea is to store an offset pointer in every node. The node pointer of the offset points towards its representative. If node a stores a node pointer $\neg b$ and an offset c, it symbolizes that a is equal to $\neg b \oplus c$. Listing 4.4 shows the extension of the `Node` structure by the offset pointer.

```
struct Node {
  OffsetPointer representative;
  ...
};
```

Listing 4.4: The extension of the existing `Node` structure by an offset pointer.

The rest of this section describes how the `union` and `find` operations work for this extended version of the algorithm.

4.2.1 Find

The `find` procedure in the union-find algorithm again finds the representative of a node. In contrast to the normal union-find algorithm, the representative, returned by the `find` operation of our extended version, is not a simple node anymore, but an offset pointer. The node pointer of such an offset pointer points to a so called *representative node*, a node whose node pointer is `null`. So the `find` operation returns an offset pointer which specifies how the node can be expressed by its representative node. Figure 4.1 shows an example forest with 6 nodes and a bit-width of 2. The forest has two representative nodes: n_1 and n_6. Therefore the call `find`(n_1) returns $n_1 \oplus 0$. The representative of n_2 is also trivial: `find`(n_2) returns $n_1 \oplus 3$. Node n_3 is equal to $\neg n_2 \oplus 2$. In combination with the representative of n_2 one can calculate that $n_3 = \neg n_1 \oplus 3$. Node n_4 is equal to $\neg n_3 \oplus 3$ and therefore equal to $n_1 \oplus 0$. Node n_5 is equal to n_3 and has therefore the representative $\neg n_1 \oplus 3$. The `find` operation returns the same offset pointers for the nodes n_1 and n_4. Therefore they are equal. The same happens with the nodes n_3 and n_5.

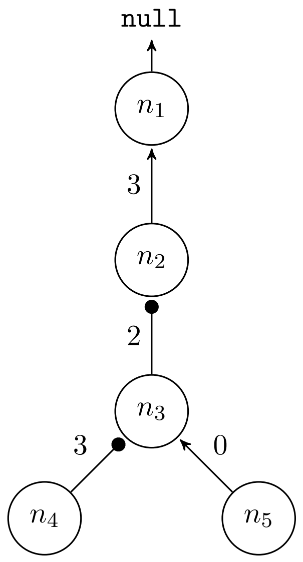

Figure 4.1: A sample forest with six nodes building two trees. The bit-width of all nodes is 2.

Listing 4.5 shows the pseudo code of the simple version of the recursive `find` function without path compression.

```
OffsetPointer find_representative (NodePointer np)
{
  OffsetPointer result;
  result.node_pointer = np;
  result.offset = 0;

  if (np.node->representative.node_pointer.node == NULL) {
    return correct_op (result);
  }

  result = find_representative (np.node->representative.node_pointer);
  result.offset = result.offset ⊕ np.node.representative.offset;

  if (np.is_inverted)
    result = invert_op (result);
  return result;
}
```

Listing 4.5: The simple recursive `find` function.

4.2.2 Union

The `union` algorithm creates the forest. Every non-redundant call to `union`, merges two trees. Initially all the nodes build its own tree, i. e., the node pointers are `null`. For every non-redundant equivalence $(a \oplus c_a, b \oplus c_b) \in R$, one edge is added to the forest and merges two trees in the following way:

- The algorithm first has to find the representative of a and b by following the offset pointers. This can be done by two calls of the `find` algorithm. With the results of the `find` operations it is able to represent a and b by their representatives a' and b' respectively: $a = a' \oplus c'_a$ and $b = b' \oplus c'_b$.
- In the next step the algorithm transforms the known equivalence $a \oplus c_a = b \oplus c_b$ into an equivalence which uses the representatives of a and b: $a' \oplus (c_a \oplus c'_a) = b' \oplus (c_b \oplus c'_b)$
- Similar to the normal union-find algorithm, the algorithm can now choose one node – either a' or b' – to be the new representative. Let us assume the algorithm chooses b' to be the new representative, so a' should point to b'. Then the algorithm transforms the equivalence into the form $a' = b' \oplus ((c_b \oplus c'_b) \ominus (c_a \oplus c'_a))$ by moving the offset from the left side to the right. Afterwards it can set the offset pointer of a': The node pointer is b' and the offset is $(c_b \oplus c'_b) \ominus (c_a \oplus c'_a)$. In case a' is an inverted node pointer, the node represented by a' does not exist in Boolector. Therefore it negates both sides of the equation and sets the appropriate values for $\neg a'$.

Example 4.4 (continued). This example in Figure 4.2 shows the forest for the explicitly specified equivalences $R = \{(a \oplus 0, \neg b \oplus 1), (\neg d \oplus 3, \neg a \oplus 2)\}$. The first element in the known equivalences is the equality $a \oplus 0 = \neg b \oplus 1$. At the moment, both nodes a and b are their own representative, so the first two steps from above are already done. Let us assume the algorithm chooses a to point towards b, so $a = \neg b \oplus 1$.

The next known equivalence is $\neg d \oplus 3 = \neg a \oplus 2$. Node d is its own representative, but the representative of a is b and a can be expressed using b by $a = \neg b \oplus 1$. The algorithm then transforms the equality to use b instead of a: $\neg d \oplus 3 = \neg(\neg b \oplus 1) \oplus 2 = b \oplus 1$. Now our algorithm has to choose again, either b points to d or vice versa. If b points towards d it reformulates the equation to $b = \neg d \oplus 2$, otherwise it transforms the equation to $d = \neg b \oplus 2$ and sets the appropriate values of the offset pointers. Figure 4.2 shows how the forest looks like in both cases, when the algorithm either chooses b or d to point to the other node for the second equivalence.

Listing 4.6 shows the pseudo code of the basic version of the `union` function without tree balancing. Instead it prefers the constant nodes to be the representatives. If none of the two nodes is a constant node then the algorithm uses the node with the smaller id as the representative. The reason for this convention is to avoid circular dependencies in combination with the reference counting system in Boolector. The return value of the `union` procedure identifies if a *new* information was inserted to

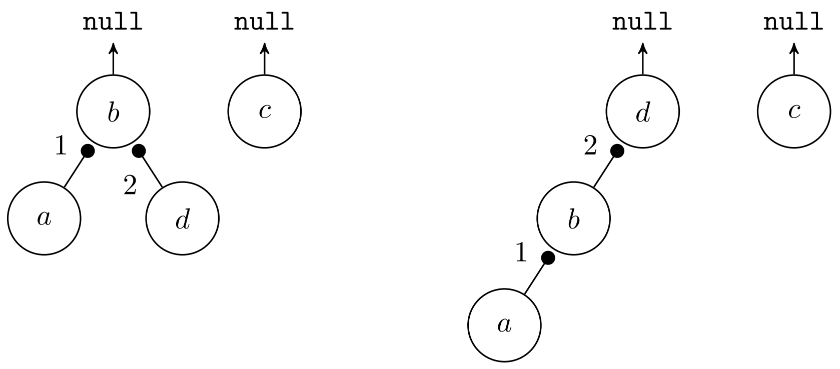

Figure 4.2: Two examples of the forest from Example 4.4. The left forest shows the case where d points towards b and the right forest shows the other option where b points towards d for the last call of the `union` operation with the arguments $\neg d \oplus 3$ and $\neg a \oplus 2$.

the structure.

```
void set_node_pointer (NodePointer from, NodePointer to, Offset offset)
{
  OffsetPointer op;
  op.node_pointer = to;
  op.offset = offset;

  if (from.is_inverted)
    op = invert_op (op);

  from.node.representative = op;
}

// meaning: from = to + offset
bool union (NodePointer from, NodePointer to, Offset offset)
{
  // get both representatives
  OffsetPointer from_op = find_representative (from);
  OffsetPointer to_op = find_representative (to);

  // calculate the offset between the representatives
  Offset repr_offset = offset ⊕ to_op.offset ⊖ from_op.offset;

  if (from_op.node_pointer.node == to_op.node_pointer.node) {
    return false;
  }

  // if one of them is a const we insert the relation always towards the const
  // otherwise we insert the relation always towards the smaller id
  // to avoid circular dependencies
  if (is_constant_node(from_op.node_pointer.node) ||
      (!is_constant_node(from_op.node_pointer.node) &&
      from_op.node_pointer.node->id < to_op.node_pointer.node->id)) {
    set_node_pointer (to_op.node_pointer, from_op.node_pointer, ⊖offset);
  }
  else {
    set_node_pointer (from_op.node_pointer, to_op.node_pointer, offset);
  }

  return true;
}
```

Listing 4.6: The simple `union` function.

4.3 More than Equality

The extended union-find algorithm presented in the last section maintains the offset equivalence closure of some known equivalences. Similar to the normal union-find algorithm, by asking if two nodes have the same offset pointer as representatives, one can check if the two nodes belong to the same equivalence class according to the offset equivalence relation. The difference is that the value returned by the `find` operation is not just a single node and represents a single equivalence class. It is a combination of a node, an inverted flag and an integer offset instead. The node of the representative is therefore the same for a bunch of equivalence classes and gives us the possibility to extract knowledge beyond the scope of equivalence classes.

4.3.1 Comparing two Nodes

In this subsection we want to have a look which information one can gain from comparing the representatives of two different nodes. Obviously, if the nodes of the representatives of two nodes are different it is not known yet, whether they will ever be equal or how they relate to each other. This is similar to the normal union-find algorithm where one cannot say anything if two nodes have a different node as its representative. Therefore the following subsection will focus on nodes with equal nodes as representatives.

Example 4.5. The two nodes a and b have the representatives $x \oplus 4$ and $x \oplus 2$ respectively. One can immediately see that a is equal to $b \oplus 2$. The bit-width bw in this example has to be larger or equal to 3, otherwise 4 would not be in C_{bw} and then $x \oplus 4$ would not be a valid offset pointer. So let us assume the nodes in this example have a bit-width of 4.

Node a is equal to $b \oplus 2$, therefore, given the value of one node, it is always possible to calculate the value of the other node. Furthermore the two values will always be different. From the definition of $\oplus$ follows that the values $x \oplus 0$, $x \oplus 1$, ..., $x \oplus 15$ will all be different for a bit-width of 4. This leads us to the following theorem:

Theorem 4.6. If two nodes have the same node pointer (same node and same inverted flag) as representative, but a different offset, then they are definitely different.

Example 4.7. We have given two nodes with a different node pointer, but again the same node, i. e., the inverted flag is different. In this case it is a bit more difficult since the negation of a bit-vector of a certain value can also be done by adding a specific value.

For example the negation of the bit-vector `1011` is equal to `0100`. This is the same value as the result of `1011` $\oplus$ `1001`. So for the specific bit-vector `1011`, the addition by 9 and the negation has the same result. If the same operations are applied to a different bit-vector it is not necessarily true that they have the same result, as shown for the bit-vector `0001`. The negation is `1110` and the result of `0001` $\oplus$ `1001` is `1010`. This means that the negation and the addition by 9 gives a different result for the specific bit-vector `0001`.

Therefore $x \oplus 9 = \neg x$ is true for some specific values of x, but not for all. We will now investigate how this behaves in general:

Theorem 4.8. Let $a = x \oplus c_1$ and $b = \neg x \oplus c_2$ be two offset pointers. They are definitely different iff the sum of c_1 and c_2 is even. Otherwise, when $c_1 \oplus c_2$ is odd, then there exist exactly two solutions for x where a is equal to b.

Proof of Theorem 4.8. Let a be equal to b, i. e., $x \oplus c_1 = \neg x \oplus c_2$. Then $\neg x$ can be reformulated as $\ominus x \ominus 1$ using the 2's complement: $x \oplus c_1 = (\ominus x \ominus 1) \oplus c_2$. The addition and subtraction operators can be used to move the variable x to one side of the equation and the constants to the other side: $x \oplus x = (c_2 \ominus c_1) \ominus 1$. The addition $x \oplus x$ is always even, so for a valid solution of x the constant $c_2 \ominus c_1 \ominus 1$ has to be even, which means that $c_2 \ominus c_1$ has to be odd. Subtracting or adding the same amount changes the least significant bit exactly this amount of times, so $c_2 \ominus c_1$ is odd iff $c_2 \oplus c_1$ is odd.

We finally have to show that there exist exactly two solutions of x for the equation $x \oplus x = c$ where c is an even number, defined as $c_2 \ominus c_1 \ominus 1$. In case of the normal addition $(+)$ of integer variables, there only exist one solution for x when the value of $x + x$ is known. The difference of $\oplus$ to the normal addition is the overflow semantic: If two values are added, there can be an overflow or not. This means, if $x \oplus x = c$ it is either the case that $x + x =$ `0`:c or that $x + x =$ `1`:c. The notation `1`:c is the concatenation of the bit `1` and the bit-vector c. To get an overflow, the sum in the most significant bit of the addition has to be greater or equal to 2. The only way this can be achieved is by having the most significant bit of x set to `1`. To avoid an overflow, the sum in the most significant bit of the addition has to be less or equal to 1. The only way this can be achieved is by having the most significant bit of x set to `0`. Therefore the two solutions of x only differ by the most significant bit, where the smaller solution (when x is interpreted as an unsigned bit-vector) is simply the integer division of $(c_2 \ominus c_1) \ominus 1$ by 2. □

Example 4.9. The representatives of two nodes are $\neg x \oplus 5$ and $x \oplus 3$. In this case the two nodes are definitely different. This statement is independent of their bit-width, as long as 5 can be expressed as a bit-vector of the same length.

Example 4.10. The representatives of two nodes with a bit-width of 4 are $\neg x \oplus 5$ and $x \oplus 2$. Then the two nodes can be equal but do not have to. If they are equal, then x has to be either `0001` or `1001`.

4.3.2 Using Additional Information

In Listing 4.6 we showed the simple version of the `union` algorithm. In this listing we ignored the `union` call if the two nodes already have the same node as their representative. With the theorems of the last subsection we can now extend this algorithm and make use of the additional information.

Whenever `union` is called with two nodes which have already the same node as their representative, the algorithm has to distinguish 3 cases: First, if the nodes are definitely different, but the call of `union` wants to set them to be equal, then the whole formula is unsatisfiable. In the second case the value of a node can be restricted to two possibilities. Finally, if the first two cases do not apply, then the information is redundant and simply ignored by the algorithm. Listing 4.7 shows the code which has to be replaced by the initial condition in line 23 of Listing 4.6.

```
if (from_op.node_pointer.node == to_op.node_pointer.node) {
  if ((from_op.node_pointer != to_op.node_pointer && repr_offset % 2 == 0) ||
      (from_op.node_pointer == to_op.node_pointer && repr_offset != 0)) {
    // they are definitely inconsistent!
    UNSAT = true;
  }
  else if (repr_offset != 0) {
    // the node pointers are different and the repr_offset is odd
    // we can restrict the value of from_op.node_pointer.node to two values
  }
  return false;
}
```

Listing 4.7: The new condition of the `union` function.

As mentioned in the comments in Listing 4.7: In some cases the algorithm can restrict the value of a variable to two possible values. This can be implemented by creating a new boolean variable and substitute the old variable by a concatenation of the new boolean variable and the remaining constant bits. We leave this optimization for future work.

4.4 Complexity

The only difference between the extended and the well known version of the union-find algorithm is the calculation with offsets. The addition of two offsets and the negation of an offset are considered as single steps. Therefore the extended version of the union-find algorithm has essentially the same time complexity as the common union-find algorithm. The two optimization strategies tree balancing (Subsection 2.5.4) and path compression (Subsection 2.5.5) can in general also be applied to this algorithm and give the same run-time optimization.

Unfortunately we cannot use tree balancing in our implementation in Boolector, because our node pointers have to point towards the node which was created first to avoid circular dependencies in the internal structure of Boolector. Fortunately we can use path compression and therefore the run-time complexity for any call of `find` or `union` is logarithmic by the number of nodes ($O(log(n))$).

The offset is considered to need constant space. The algorithm adds one offset pointer for every node. Therefore the space complexity is linear by the number of nodes ($O(n)$).

4.5 Propagation

Every time when a node is asserted or created, our algorithm tries to extract information from this triplet. It extracts the premise from the extended union-find datastructure, uses the auto generated rules from Chapter 3 and inserts the resulting information into the extended union-find datastructure.

Whenever the algorithm inserts a non-redundant information to the datastructure it propagates the newly gained information for a node n towards every node which uses n as its input. These triplets have the chance to collect new information. This propagation stops automatically, because there only exists a small and finite amount of new information per node.

4.6 Implementation Details

We implemented the extended union-find algorithm and integrated the implementation to Boolector. This section describes three parts of Boolector, where we had to solve Boolector specific integration problems.

4.6.1 Simplified Pointer

Boolector has already the concept of proxy nodes. If node a is substituted by node b, then node a is changed to a proxy node and its simplified pointer points towards b. Furthermore all parents of a are recursively recalculated with b as their new child. It was necessary to merge this concept of simplified pointers with our concept of offset pointers. Therefore our algorithm treats such a simplified pointer as an offset pointer without the inverted flag and with offset 0. In addition, a proxy node never has a representative node, because the `find` operation follows the simplified pointer in this case.

4.6.2 Reference Counter

Boolector has reference counts on nodes and deallocates the memory for a node as soon as its reference count reaches zero. Therefore if node a has an offset pointer with node b as its node, then node a has to increment the reference count for node b by one. Node a has to decrement the reference count again as soon as the offset pointer changes or node a is deallocated. There are other situations where one node depends on another node and therefore increment the reference count, for example if a node is a parent of another node. Having all these concepts together makes it difficult to avoid circular dependencies. This is also the reason why tree balancing is not possible in our union-find datastructure.

4.6.3 Skeleton Preprocessing

When Boolector creates the DAG it uses simple rewriting rules only. More complicated rewriting strategies are applied as a preprocessing step of the CNF transformation, i. e., immediately before the SMT solver converts the DAG to CNF clauses and passes them to the underlying SAT solver. One of these strategies is *skeleton preprocessing* [12]. Here, a part of the DAG – called skeleton – with nodes of bit-width 1 is converted to CNF clauses and sent to the preprocessor of the underlying SAT solver. The skeleton consists of all the asserted nodes and recursively all its children if they are `and` nodes or if they are `eq` (equal) or `ite` (if-than-else) nodes with an input bit-width of 1.

Boolector uses the result of the SAT preprocessor in the following way [12]:

- If the preprocessor returns that the skeleton is unsatisfiable then the SMT solver returns that the whole formula is *unsatisfiable*.
- If the preprocessor can simplify a boolean variable to the truth-values *true* or *false*, then Boolector asserts the corresponding node or its negation in the DAG respectively.

We combined skeleton preprocessing with our extended union-find structure in the following way:

- At the beginning our algorithm adds the information, which is stored in the extended union-find datastructure, to the skeleton. When the skeleton is converted to CNF clauses, the algorithm uses the representative for each boolean node instead of the node itself. Therefore any two nodes which have the same node as representative also have the same clause in the CNF formula.
- After the call the algorithm extracts knowledge from the preprocessor and stores it in the union-find datastructure. If the SAT preprocessor simplifies the CNF clauses in a way that two boolean variables have to be equal, then our algorithm adds this information to the union-find datastructure.

4.7 Related Work

Robert Nieuwenhuis and Albert Oliveras already describe a congruence closure algorithm with integer offsets [13]. They extend a common congruence closure algorithm by the concept of integer offsets. Unfortunately they do not describe the details of their algorithm, for example the information is missing whether they mean real integers or if they also use the overflow semantics of the addition to restrict the number of possible integer values. Furthermore they do not give experimental results of the effect of this extension. However they do prove that the extension does not affect the run-time of their original congruence closure algorithm which is the same for our approach, when also considering offset addition and negation as single steps.

Chapter 5

Congruence Closure

In the last chapter we introduced an algorithm for maintaining the offset equivalence closure, an extended version of the equivalence closure. In this chapter we will first compare congruence closure with this extended version. Afterwards we will show how we implemented congruence closure and then present additional rules how we can make our congruence closure algorithm more powerful, because we use offset pointers as representatives.

5.1 Congruence Closure vs. Offset Equivalence Closure

As defined in Corollary 2.11, the congruence closure of R is the smallest congruence relation which contains R. A congruence relation is an equivalence relation which is monotonic. A relation R is called monotonic (see Definition 2.5) if the following rule holds for every function f:

$$\frac{(a_1, b_1) \in R, (a_2, b_2) \in R, \ldots, (a_n, b_n) \in R}{(f(a_1, a_2, \ldots, a_n), f(b_1, b_2, \ldots, b_n)) \in R}$$

On the other hand, the term offset equivalence relation is defined as an equivalence relation where the monotonic rules of bit-wise negation and constant addition hold:

$$\frac{(a, b) \in R}{(\neg a, \neg b) \in R} \qquad \frac{(a, b) \in R}{\forall c \in C\colon (a \oplus c, b \oplus c) \in R}$$

Obviously one might think that congruence closure is a superset of the offset equivalence closure, because the monotonic rule for all the functions and not just the bit-wise negation and the addition hold, but this is not correct. The congruence closure in general works with simple nodes instead of offset pointers and applies the monotonic rule for every function in an abstract way [13]. The offset equivalence closure algorithm is based on offset pointers and directly applies the two monotonic rules to these offset pointers to get new ones. The following example shows the difference using concrete values.

Example 5.1. Let x and y be two nodes. Furthermore $x \oplus 5$ is equal to $y \oplus 3$. The question is, if $x \oplus 2$ is equal to y. The offset equivalence closure algorithm presented in Chapter 4 obviously says yes. Instead, the congruence closure algorithm only has the equivalence classes $\{x\}$, $\{y\}$, $\{x \oplus 5, y \oplus 3\}$ and some classes for the constant values and therefore cannot say if $x \oplus 2$ and y are equal or not. The reason for the difference is that the algorithm is not able to evaluate the $\oplus$ operation. According to the congruence closure algorithm, $(x \oplus 5) \ominus 3$ is certainly equal to $(y \oplus 3) \ominus 3$, but it cannot simplify these terms to $x \oplus 2$ and y respectively.

Of course, it is also possible to extend an existing version of the congruence closure algorithm to work with offset pointers and immediately simplify addition with constant values. Instead we used another approach for the congruence closure as explained in the next section.

5.2 Our Implementation

We have already implemented the union-find algorithm with offsets and now want to extend this approach by the idea of congruence closure. This means that the algorithm wants to find a function which is applied at least twice to the same input values, because in this case the result has to be equal as well.

In Boolector constants, variables and functions are nodes. If a node is a function it has a function type and between one and three input nodes, dependent of the type of the function. The idea of our implementation of congruence closure is to iterate through all nodes which are functions and check if there is another node of the same function type and with the same nodes as inputs. To check if two input nodes are equal, the algorithm should use the `find` function of our extended union-find algorithm and compare the representatives. To do this efficiently our implementation keeps a hash table of nodes where the hash value depends on the function type of the node and the representatives of the input parameters.

If two function nodes turn out to be equal, because they are of the same type and their input parameters are equal, then the algorithm does not add this information to the union-find datastructure. Instead one node is substituted by the other. This implementation is called `congruence_closure`.

5.2.1 Ensure Correctness of Substitution

If a node is substituted by another one, it is not necessarily the case that the DAG still represents the original formula, because the information stored in the union-find datastructure is collected from the whole DAG including the node which will be substituted. Example 5.2 shows an example where it is wrong to replace a node by another one, even though the inputs are considered to be the same according to the union-find datastructure.

Example 5.2. The DAG consists of two 4-bit variables a and b and it is asserted that $a \oplus 3$ is equal to $b \oplus 3$. The DAG is shown in Figure 5.1.

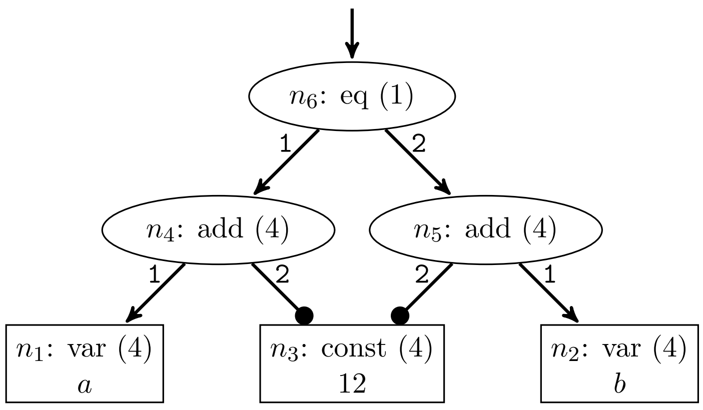

Figure 5.1: The DAG of Boolector, when $a \oplus 3$ equals to $b \oplus 3$ is asserted.

When the DAG is created, the algorithm already collects information for the union-find datastructure: The results of the additions are by 3 larger than the variables ($n_4 = n_1 \oplus 3$ and $n_5 = n_2 \oplus 3$) and the constant is trivially 3 larger than the node representing the zero for four bits ($n_3 = \mathit{zeroNode}_4 \oplus 3$). Furthermore, when the equality node is asserted, the equivalences $n_6 = \mathit{zeroNode}_1 \oplus 1$ and then also $n_4 = n_5$ are added to the structure.

Now the representative of n_1 is equal to the representative of n_2, indicating that the nodes have to be equal. This means that the two additions n_4 and n_5 have the same variables as inputs and therefore one can be substituted by the other, but then the equality would be trivially true and the whole DAG would be satisfied without making sure that n_1 is the same as n_2 in say another part of the formula.

This Example shows that substituting a node by another, just because the inputs have the same representative is not always correct. The reason is that the information in the datastructure is propagated bottom up and top down. It is not possible to substitute one node by the other if the information that the inputs of these two nodes are equal is based on the existence of these nodes. In Example 5.2

node n_4 cannot be substituted by node n_5, because the children n_1 and n_2 are only considered to be equal, because of the existence of n_4 and n_5.

We solve this problem by simply adding the information stored in the union-find datastructure to the DAG and never substitute these additional nodes. Then we can be sure that no information is based on any existing nodes which can be substituted. This means that our algorithm inserts for every node n with the representative $r \oplus c$ an additional `add` node to add r and c and an additional `eq` node to assert that n is equal to $r \oplus c$. This process is called `add_equality_information`.

Example 5.2 (continued). The following table shows an example for the representatives of the nodes:

Node	Representative	Node	Representative
n_1	$n_1 \oplus 0$	n_4	$n_1 \oplus 3$
n_2	$n_1 \oplus 0$	n_5	$n_1 \oplus 3$
n_3	$zeroNode_4 \oplus 3$	n_6	$zeroNode_1 \oplus 1$

In this example the process `add_equality_information` will only add the assertion $n_2 = n_1$, because all the others are trivial. For example $n_3 = zeroNode_4 \oplus 3$ will be rewritten to $n_3 = 3$ and that is already the case. Another example is node n_5: Here, instead of creating $n_1 \oplus 3$, the already existing node n_4 will be reused. Then for $n_5 = n_4$ the node n_6 will be used and this one is already asserted.

As soon as $n_1 = n_2$ is asserted, n_5 can be substituted by n_4. Each substitution leads to rebuilding all nodes which depend on the substituted node. As a consequence the DAG simplifies a lot: Node n_6 is trivially true, because both inputs are the same and will therefore be removed from the assertion list. As a consequence node n_6 will be deallocated, because no node references to n_6 anymore. Then also n_5 will be removed and afterwards also n_3. The only remaining part will be the added assertion as shown in Figure 5.2.

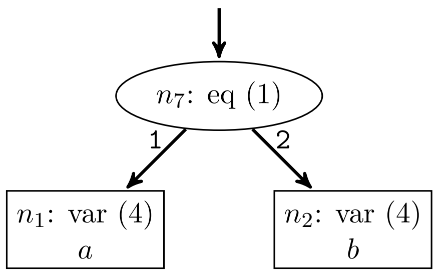

Figure 5.2: The DAG of Boolector after calling `add_equality_information` and substituting the congruent nodes.

The procedure `add_equality_information` has the additional advantage that it adds the information from the union-find datastructure to the DAG. This or at least a similar approach is necessary, because Boolector only passes the information of the DAG to the SAT solver by bit-blasting the DAG. If this procedure would not be called, then the collected information would never be used for simplification of the DAG or for giving hints to the SAT solver. The procedure `add_equality_information` and our implementation of the congruence closure algorithm are the only algorithms making changes to the DAG which we have added to Boolector. The other algorithms presented in Chapter 4 only collect information and build the knowledge base for these two algorithm.

5.2.2 Substitution Order

This subsection explains which node has to be substituted by the other when the congruence closure algorithm finds two nodes with equal inputs. In Example 5.2, either n_4 can be substituted by n_5 or n_5 by n_4. Both ways are possible and lead to the same result. This is not always the case as the following example shows.

Example 5.3. We have given a DAG of 4-bit nodes as shown in Figure 5.3. This DAG contains two `and` nodes (m_6 and m_3) where both of them have the variable e as their first input. The second input of these two `and` nodes is different. However the second inputs of both `and` nodes are considered to be equal, because of the asserted `eq` node m_9. Therefore our congruence algorithm will find the substitution of m_6 and m_3.

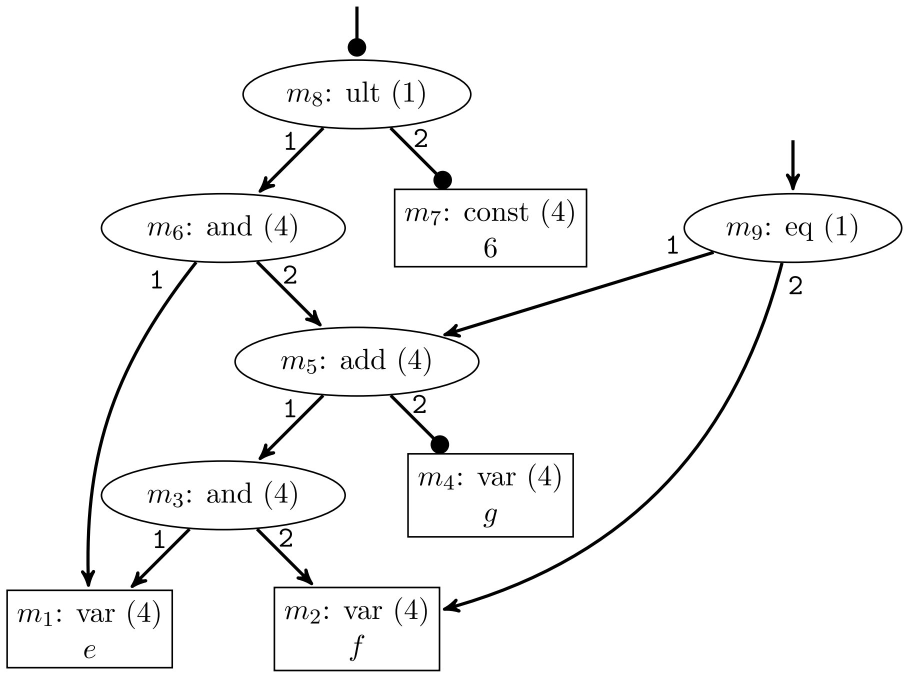

Figure 5.3: The DAG of Boolector for Example 5.3 before applying the congruence algorithm.

In this example, node m_6 depends on node m_3, so it is not possible to substitute m_3 by m_6, because m_6 simply uses somehow the result of m_3. However the other way around is possible. Node m_6 can be substituted by m_3. In this case the unsigned less-than node m_8 would directly use m_3 as its first input as shown in Figure 5.4.

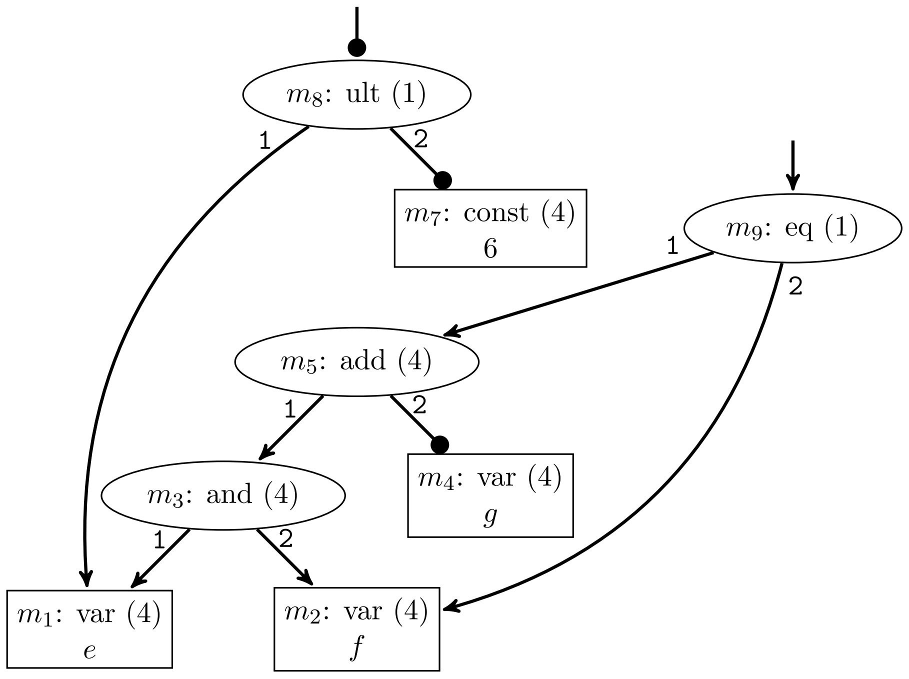

Figure 5.4: The DAG of Boolector of Example 5.3 after applying the congruence algorithm.

To solve this issue of the correct and possible substitution order, our algorithm simply substitutes the newer node by the older one. In this way it is never the case that the substituted node is a child of the substitution, because the DAG in Boolector is created bottom-up. To identify which node is older, the algorithm compares the identifiers of the nodes. Every node has a unique identifier which will never be reused and increases for every new node.

5.3 Extended Version

In the previous section we described our congruence closure algorithm by iterating through all functions and check if they have the same input nodes. For identifying if two nodes are equal the algorithm simply compares the representatives of our union-find datastructure. In this section we extend this idea to find other nodes which are congruent, but do not have exactly the same inputs. This means that we make use of properties of specific functions and do not interpret the functions in an abstract way anymore.

An example for a property of a function is the commutative property of some binary functions. Boolector has 7 binary operations with equal bit-widths of the two input parameters:

- `add`: Addition
- `and`: Bit-wise conjunction
- `eq`: Equality check
- `div`: Division
- `mul`: Multiplication
- `mod`: Modulo operation
- `ult`: Unsigned less-than

The operations `add`, `and`, `eq` and `mul` are commutative and therefore the input parameters can be swapped without changing the result. We implemented this by normalizing the order of the inputs before generating the hash value of a node. Normalizing means that the representatives of the input parameters of a commutative binary node have to be in a certain order for generating the hash value. First the algorithm sorts the two representatives by the id of their node, second by the inverted flag of the node pointer and third by the offset.

Example 5.4. The first two columns of the following table show examples of the two representatives of the input parameters of a binary commutative function. The third column indicates if our algorithm swaps the representatives for generating the hash value or if they stay the same. Lower case letters are used for referencing nodes. Node a has the smallest id, node b the second smallest and so on. The text in braces – in case they will be swapped – specifies the the reason for swapping.

$\neg a \oplus 2$	$b \oplus 4$	stay the same
$d \oplus 4$	$a \oplus 2$	will be swapped (id)
$b \oplus 2$	$\neg b \oplus 4$	stay the same
$\neg b \oplus 2$	$b \oplus 4$	will be swapped (inverted flag)
$\neg b \oplus 2$	$\neg b \oplus 6$	stay the same
$b \oplus 7$	$b \oplus 1$	will be swapped (offset)

The representatives of the nodes are offset pointers, which increases the possibility to find equivalent nodes with different inputs. So after the the normalization of the commutative operations, our algorithm distinguishes between the different operation types and performs special normalization steps for the following two operations:

- `add` operation: Both representatives itself are expressed by additions. The addition operation is commutative and associative. Therefore the addition of two representatives $(a \oplus c_a) \oplus (b \oplus c_b)$ can be reformulated to $(a \oplus 0) \oplus (b \oplus (c_a \oplus c_b))$. This is nothing else than moving the constant offset from the left representative to the right.

- `eq` operation: The `eq` operation can be normalized in a similar way as the `add` operation: The offset from the left representative will also be moved to the right representative. The algorithm reformulates the equality of two offset pointers $(a \oplus c_a) = (b \oplus c_b)$ to $(a \oplus 0) = (b \oplus (c_b \ominus c_a))$. Furthermore the algorithm reformulates the equality to have no inverted node pointer on the left hand side, i. e., if a is an inverted node, both sides will be inverted.

There are more special cases for the normalization of certain operations. For example two *if-then-else* operations are equal if the condition is negated and the two branches are swapped. We leave the search and implementation of more special cases for future work.

5.4 Related Work

There already exist algorithms and datastructures which maintain the congruence closure of some known equivalences [5, 15]. These algorithms do not need to iterate through all the nodes at a certain point in time as our implementation does. Instead their algorithm – similar as the union-find algorithm does with the equivalence closure – always updates the congruence closure immediately if an information is inserted. We have chosen our approach of iterating though the nodes, because we differentiate between the knowledge we gained from our extended union-find algorithm and the equivalence of two functions. In the first case we insert this knowledge to the DAG and in the second case we substitute nodes in the DAG and therefore reduce the number of nodes again.

Chapter 6

Experimental Results

In Chapter 4 we presented our implementation of a union-find datastructure with offsets. This structure is filled with information during the construction of the DAG. The information is used in various steps of the preprocessing loop of Boolector, either to simplify the DAG or to collect even more information which can trigger other rewriting strategies. We extended one step and added two more to this loop and described all of them in this thesis:

- `skeleton_preprocessing`: Here the preprocessor of the underlying SAT solver is used to simplify the top-level boolean skeleton of the DAG (see Subsection 4.6.3).
- `add_equality_information`: This routine adds all the information from the extended union-find datastructure to the DAG (see Subsection 5.2.1).
- `congruence_closure`: The algorithm for substituting one node by another node if both have the same operation type and if their inputs are pairwise equal (see Sections 5.2 and 5.3).

In this chapter we will show the results of different experiments and analyze them. We used the same hardware configurations and testcases for all experiments: A cluster with 30 nodes where each node runs Ubuntu 14.04.2 LTS, has a 2.83GHz Intel Core 2 Quad CPU and 8GB of memory. Every test case was limited to 1200 seconds of CPU time and 7GB of memory. We use the same time limit as Nadel in [11] to allow a better comparison with his results. The memory limit of 7GB is motivated by the maximum available memory of 8GB. For the experiments we used the ASP families from the SMT-LIB, except the *Fastfood* family. Initially the `Offset` datatype from the extended union-find algorithm was implemented as an unsigned integer and the algorithm was limited to a maximum bit-width of 16. The Fastfood family is the only ASP family with bit-vectors of a larger bit-width than 16.

6.1 Boolector$_{\text{base}}$ vs. Boolector$_{\text{new}}$

In this section we compare the results of two different versions of Boolector:

- *Boolector$_{base}$*: This is the base version of Boolector.
- *Boolector$_{new}$*: In this version we integrated all the algorithms presented in the previous chapters into Boolector: `skeleton_preprocessing`, `add_equality_information` and `congruence_closure`.

Table 6.1 shows the comparison of Boolector$_{\text{base}}$ and Boolector$_{\text{new}}$. The first column specifies the family by the first six characters. The second column shows the number of test cases in this family which have been successfully solved by both versions. The total execution time (TIME[s]), the time spent for pure SAT-solving (SAT[s]) and the time spent for rewriting (RW[s]) are summed up per family and shown for each version separately.

		Boolector$_{\text{base}}$			Boolector$_{\text{new}}$		
Family	#	TIME[s]	SAT[s]	RW[s]	TIME[s]	SAT[s]	RW[s]
15Puzz	15	450.79	359.31	73.87	563.38	213.37	319.89
Blocke	29	972.11	960.85	3.77	512.24	489.36	14.39
Channe	8	167.89	150.06	7.68	77.92	38.21	22.33
Connec	21	157.31	149.49	6.01	192.92	144.10	44.08
Disjun	1	683.00	637.86	29.27	480.32	422.81	31.23
EdgeMa	20	16579.67	16327.41	131.96	9137.94	8487.98	511.88
Genera	29	794.87	608.94	139.57	1022.96	192.63	746.07
GraphC	15	5121.75	**5116.76**	1.89	5602.17	5587.91	8.76
GraphP	9	2492.57	2476.71	9.59	1838.11	1792.58	35.57
Hamilt	29	199.81	191.34	2.69	126.30	102.68	16.10
Hanoi-	11	736.16	692.03	23.17	503.79	363.20	105.92
Hierar	12	695.75	666.78	12.50	111.10	52.33	42.23
Knight	3	156.09	154.56	0.53	59.35	54.91	3.12
Labyri	7	5948.24	5236.94	613.00	6390.35	4457.33	1726.96
MazeGe	29	728.57	720.08	5.73	690.10	664.80	20.61
SchurN	29	994.79	963.61	16.61	642.45	547.96	66.37
Sokoba	26	394.80	377.26	9.04	279.56	216.11	46.87
Solita	22	2551.27	2511.12	18.57	1936.85	1845.67	58.02
Sudoku	4	2316.28	2271.28	19.81	917.82	830.15	63.20
Travel	29	2054.01	2021.35	6.33	1137.20	1076.16	35.13
Weight	28	1992.74	**1986.30**	0.81	3090.29	3075.65	6.47
WireRo	20	5397.83	5298.54	37.47	5450.02	5152.43	212.16

Table 6.1: Run-time comparison of Boolector$_{\text{base}}$ and Boolector$_{\text{new}}$. Only testcases which have been successfully solved in both versions are considered in this comparison.

In general Boolector$_{\text{new}}$ spends less time for SAT solving than Boolector$_{\text{base}}$. The SAT solver of the base version is only faster in two families: *Graph Coloring* and *Weight Bounded Dominating Set.* Here the new version needs about 10% and 50% more time, respectively. These two entries are highlighted in bold. Figure 6.1 compares the times spent for pure SAT solving of both versions. The solid line shows the case where both versions need the same time. Most testcases are below this line, thus the new version is faster for theses testcases. However the time spent for rewriting is much higher for the new version.

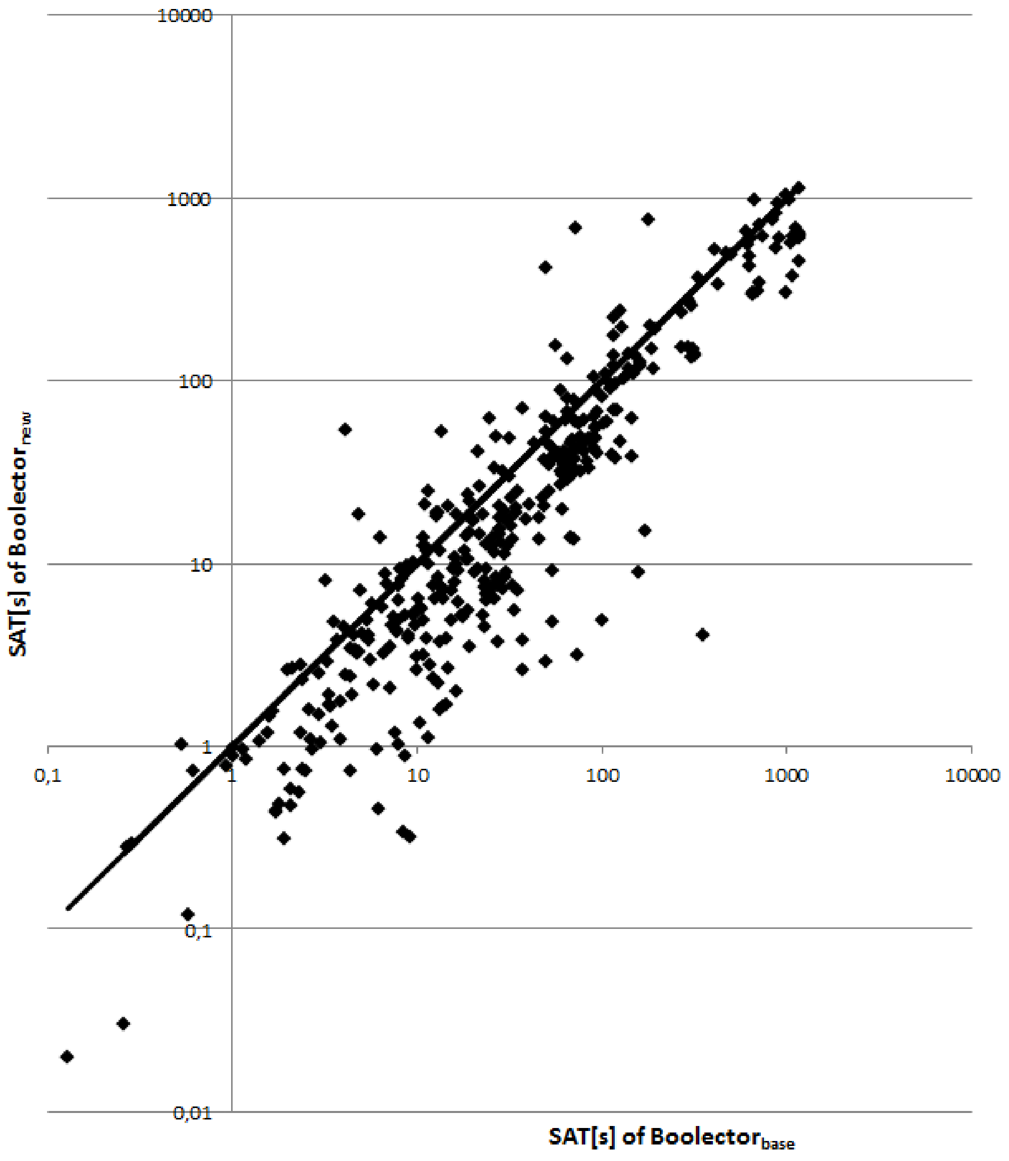

Figure 6.1: This figure compares the times spent for pure SAT solving with Boolector$_{\text{base}}$ and Boolector$_{\text{new}}$.

6.1.1 Rewriting

On average, Boolector$_{\text{new}}$ spends about 4 times as much time for rewriting than Boolector$_{\text{base}}$. Figure 6.2 shows the time spent for rewriting with Boolector$_{\text{base}}$ compared to Boolector$_{\text{new}}$. Both axes are of logarithmic scale. Every point in the plot represents a single test case. The solid line shows the case where RW of Boolector$_{\text{new}}$ takes 4.2 times as much as RW of Boolector$_{\text{base}}$.

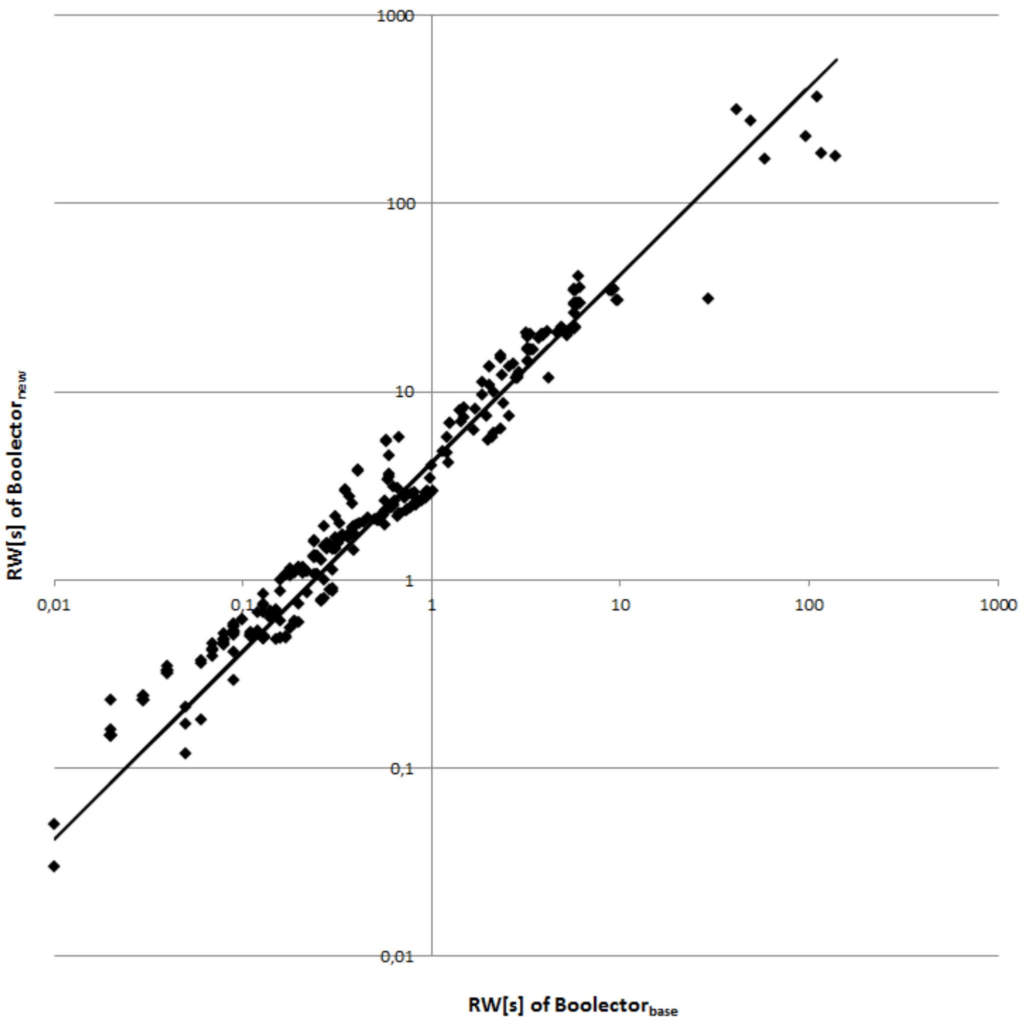

Figure 6.2: This figure compares the times spent for rewriting with Boolector$_{\text{base}}$ and Boolector$_{\text{new}}$.

The main reason is the variable substitution which takes about a quarter and skeleton preprocessing which takes almost half of the total rewriting time of Boolector$_{\text{new}}$. The `add_equality_information` algorithm inserts many `eq` nodes and asserts them. This can trigger a variable substitution in Boolector. Therefore the new version has to substitute more variables than the base version and thus the variable substitution part of the rewriting takes more time. We think that there exists a faster algorithm to perform variable substitution in Boolector, but leave this investigation for future work.

Boolector$_{\text{new}}$ calls the `skeleton_preprocessing` algorithm in every iteration of the preprocessing loop, whereas Boolector$_{\text{base}}$ called this algorithm just once. This algorithm is time consuming and therefore highly influences the run-time of the rewriting in Boolector$_{\text{new}}$. The investigation if skeleton preprocessing can be done more efficiently or if it is necessary in every iteration is left for future work.

6.1.2 SAT-Solver

For Boolector$_{\text{new}}$, the SAT-solver is in 20 of 22 families faster than before. To measure the effect of the `congruence_closure` algorithm, we use the total number of variable substitutions caused by this algorithm (VSCC) as a metric. Figure 6.3 compares VSCC to the speed-up of the SAT-solver in a

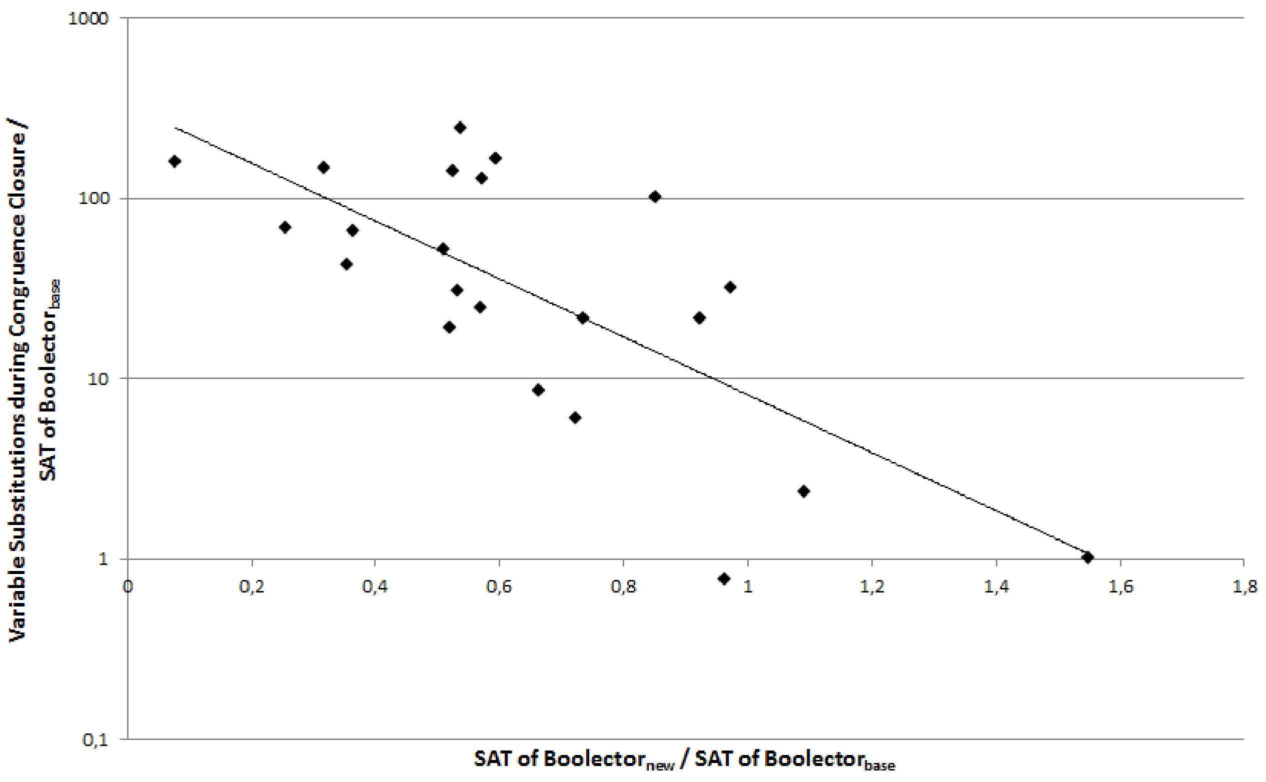

Figure 6.3: This figure compares the number of variable substitutions (normalized with the SAT-time of Boolector$_{\text{base}}$) with the speed-up of the underlying SAT-solver

semi logarithmic plot. The x-axis shows the speed-up of the SAT solver. The number of variable substitutions during the `congruence_closure` algorithm is normalized by the time for the SAT-solver of Boolector$_{\text{base}}$ and shown on the logarithmic y-axis. We abbreviate this ratio by VSCC/SAT$_{\text{base}}$. Every point represents a family of Table 6.1.

One can see that the two families where the SAT-call is slower than before, have a low value on the y-axis. This means that they have less variable substitutions compared to the run-time of the SAT-solver of the base version. There is also a third family with a very low ratio and almost no speed-up. On the other hand, the families with a large ratio of VSCC/SAT$_{\text{base}}$ have a good speed-up. We can definitely say that this ratio correlates to the speed-up.

6.1.3 Time-out and Memory-out

Table 6.2 shows all families where at least one testcase from one of the two versions took longer than 1200 seconds (time-out) or needed more than 7GB of memory (memory-out). In this regard the two versions are very similar. Only for the *Edge Matching* and *Sudoku* family, Boolector$_{\text{new}}$ solved way more testcases than Boolector$_{\text{base}}$. The new version solves for example one testcase less for the family *Labyrinth*. The rewriting for this particular testcase takes serveral hundred seconds in the base version and almost 1000 seconds in the new version. This huge difference of the rewriting time is the reason why the new version gets a time-out.

	Boolector$_{\text{base}}$		Boolector$_{\text{new}}$	
Family	time-out	memory-out	time-out	memory-out
Channe	2		2	
Disjun	3	5	3	5
EdgeMa	9		3	
GraphC	14		13	
GraphP	3		2	
Knight	6		7	
Labyri	21		22	
Solita	5		5	
Sudoku	6		0	
Weight	1		1	
WireRo	3		3	

Table 6.2: Comparing the time-outs and memory-outs for Boolector$_{\text{base}}$ and Boolector$_{\text{new}}$.

6.1.4 Comparison with Nadel's results

In this subsection we compare our experimental results with the results from Nadel's approach presented in [11] and integrated into Intel's SMT solver Hazel. Hazel$_{\text{base}}$ is the base version of Hazel and Hazel$_{\text{new}}$ includes the approach from Nadel. Nadel uses a slightly faster processor (Intel Xeon with 3GHz) for the experiments , but the same time limit as we do. In the following list we highlight four families to show differences and commonalities:

- Hazel$_{\text{new}}$ is able to solve all testcases from the *Edge Matching* family. In this family Boolector$_{\text{new}}$ outperforms Boolector$_{\text{base}}$ as well and can solve 6 testcases more. Hazel$_{\text{new}}$ only needs a fifth of the run-time as Hazel$_{\text{base}}$ to solve all these testcases, but Boolector$_{\text{new}}$ is only almost twice as fast as Boolector$_{\text{base}}$.
- Boolector$_{\text{new}}$ and Hazel$_{\text{new}}$ are able to solve all *Sudoku* testcases and get about the same speedup compared to their base versions.
- Hazel$_{\text{new}}$ solves all testcases from the family *Labyrinth*, whereas the base version of Hazel was not able to solve 8 of them. Both versions – Boolector$_{\text{base}}$ and Boolector$_{\text{new}}$ – are not able to solve more than 20 of these testcases.
- The family *Weight Bounded Dominating Set* is the family where the SAT solver of Boolector$_{\text{new}}$ takes about 50% more time than the SAT solver of Boolector$_{\text{base}}$. However, Hazel$_{\text{new}}$ solves all testcases is is 10 times faster than the base version of Hazel.

6.2 Comparison of Two Different `Offset` Datatypes

In Listing 4.1 we used the abstract type `Offset` to represent the offset of an offset pointer. Initially we used in our implementation the datatype `uint32_t` for this offset. This is a datatype to store an unsigned integer for 32 bits. Then the algorithms for adding two offsets or subtracting them are rather simple: One can use the normal integer addition and simulate the overflow semantics for the bit-vector addition ($\oplus$) by masking out the overflow bit.

Later on we exchanged the `uint32_t` datatype for the offset to the datatype `BtorBitVector`. This type can store bit-vectors of variable length. Internally, it uses a dynamically allocated array of `uint32_t` to store packets of 32 bits. With this type it is possible to use the extended union-find datastructure for variable bit-widths.

In Figure 6.4 we compared the rewriting time of Boolector$_{\text{new}}$ with the two different offset datatypes. The x-axis shows the slightly faster implementation with `uint32_t` as offset, whereas the y-axis show the implementation with the dynamic `BtorBitVector` datatype as offset. The solid line shows the average increase and has a slope of 1.08, i. e., the dynamic offset is about 8% slower than the integer offset.

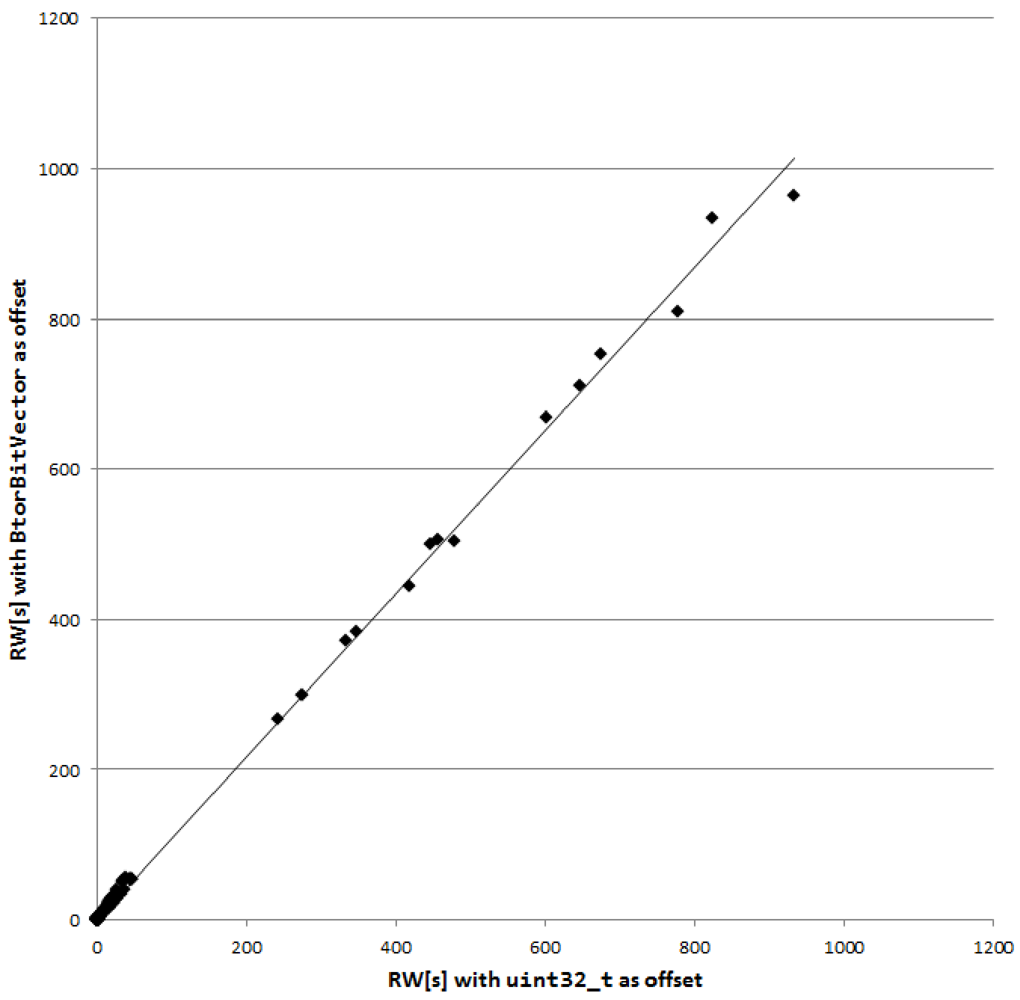

Figure 6.4: This figure compares the times spent for rewriting with Boolector$_{\text{base}}$ and Boolector$_{\text{new}}$.

6.3 Calculating Rules On-The-Fly vs. Pregenerating Rules

In Chapter 3 we mentioned that Alexander Nadel calculates all the necessary rules at run-time [11], but our algorithm uses pregenerated rules for small bit-widths to save calculation time. In this section we want to measure the time which is needed to calculate the rules at run-time instead of using the precalculated ones. Every rule is cached and therefore calculated at most once.

Figure 6.5 compares the time spent for rewriting when generating the rules on-the-fly or precalculate them. The double logarithmic plot shows three different behaviours:

1. The most common case is that the time difference of the two rewriting parts is not measurable or really small.
2. Some testcases show a different behaviour. They have a small constant offset. This is visible

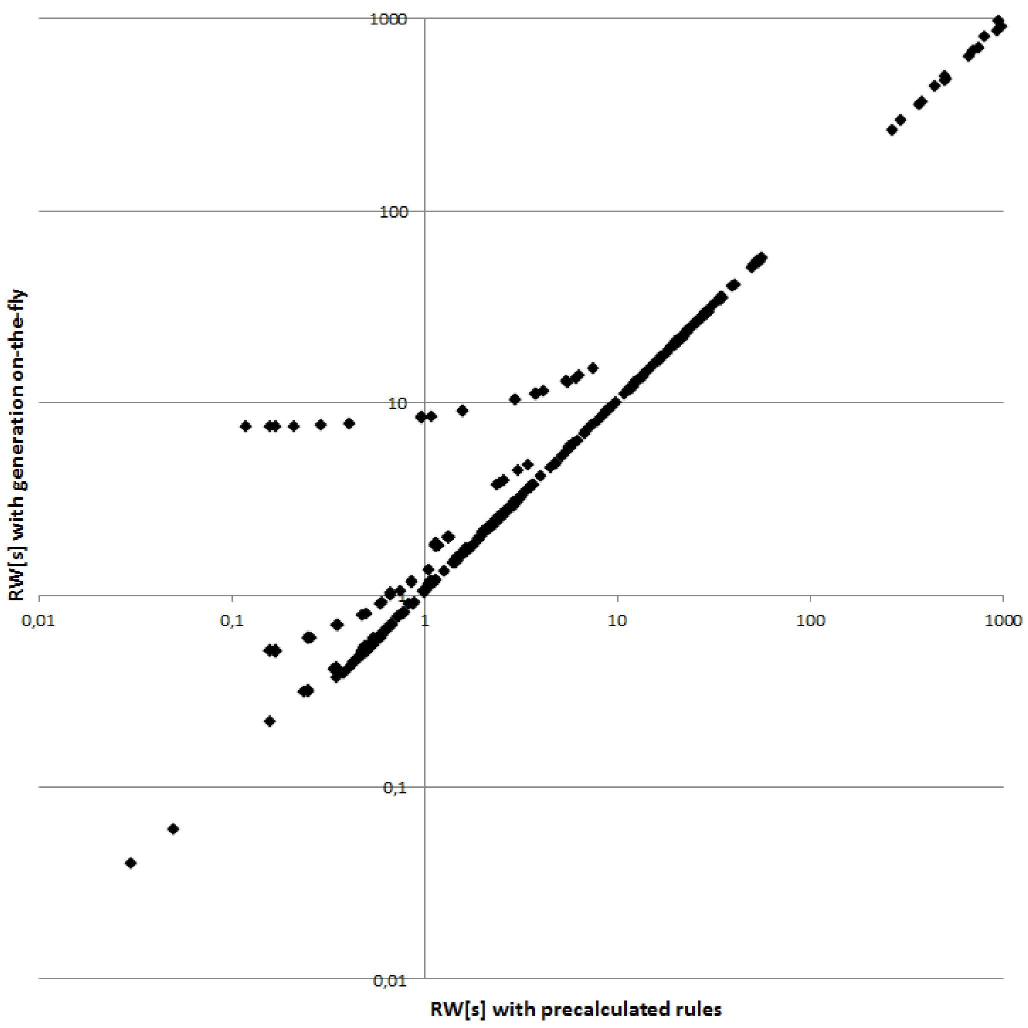

Figure 6.5: This figure compares the times spent for rewriting of Boolector$_{\text{new}}$ with pregenerated rules and generation on-the-fly.

in Figure 6.5: The rewriting time for the generation on-the-fly is a fraction of a second larger than the rewriting with precalculated rules. These testcases are from the families *Traveling Salesperson* and *Weight Bounded Dominating Set*.

3. More visible is the different behaviour from the family *Maze Generation* and the random testcases from the family *Hamiltonian Path*. Here the constant offset increases for about 10 seconds.

A small constant offset for testcases with a large execution time for the rewriting can not be seen in this double logarithmic plot. However there is also no time difference in the detailed measurement results.

Chapter 7

Conclusion and Future Work

We extended the common union-find algorithm with integer offsets. This extension has basically the same complexity than the original version and various advantages:

- The datastructure can store and maintain more information than simple equivalences. It is possible to add and query for dependencies with constant offsets.
- The algorithm can detect if two nodes are definitely different.
- Further, the algorithm can find inconsistencies or reduce the value of a variable to two possibilities.

We integrated this datastructure together with the automatic rule generation and an algorithm for congruence closure into Boolector. Thus the run-time of the underlying SAT solver reduced in 20 of 22 ASP families. Unfortunately the run-time of the rewriting strategies of Boolector increased by a factor of about 4. Therfore the overall run-time was only in some of the ASP families faster than before. This leads us to the future work:

- The rewriting process of Boolector is much slower when all algorithms presented in this thesis (`add_equality_information`, `sekeleton_preprocessing` and `congruence_closure`) are executed in the preprocessing loop of Boolector. One reason is that skeleton preprocessing is time consuming and done in every cylcle of the preprocessing loop. The other two algorithms trigger variable substitutions in Boolector. We leave the optimization of this variable substitution algorithm and the investigation if skeleton preprocessing is needed in every loop cycle for future work.
- Whenever the algorithm inserts a new information to the extended union-find datastructure, i. e., two trees are merged, it propagates this information to all parents of the node where this information was found. This propagation is local. A different approach, where all nodes which get a new representative node are triggered for propagation, could be better.
- When Boolector can restrict the value of a variable a, because some of the bits are constant, then

the variable is sliced. This means that the variable is substituted by a concatenation of constant bit-vectors and new variables with a smaller bit-width than a. For example if an unsigned 4-bit variable a has to be smaller than 2, then the three most significant bits have to be zero. Therefore the node a is substituted by the concatenation of the constant bit-vector `000` and a one-bit variable. This information that parts of the value of a node are constant is propagated through the DAG and many nodes are sliced in a similar way. This concept does not fit with the extended union-find datastructure because many nodes change their bit-width and are therefore not comparable anymore. To assert that some of the bits of a node have to be equal to a constant value, instead of slicing the node into several nodes with smaller bit-widths, would be a possible approach. We leave the implementation and investigation for future work.

List of Figures

Listings

Bibliography

[1] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[2] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[3] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[4] Thomas H Cormen. *Introduction to algorithms.* MIT press, 2009.

[5] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[6] Václav Flaška, Jaroslav Ježek, Tomáš Kepka, and Juha Kortelainen. Transitive closures of binary relations. i. *Acta Universitatis Carolinae. Mathematica et Physica*, 48(1):55–69, 2007.

[7] Trevor Alexander Hansen. A constraint solver and its application to machine code test generation. 2012.

[8] Matti Järvisalo, Marijn JH Heule, and Armin Biere. Inprocessing rules. In *Automated Reasoning*, pages 355–370. Springer, 2012.

[9] Donald E Knuth. *Art of Computer Programming, Volume 4, Fascicle 6, Satisfiability.* Addison-Wesley Professional, 2015.

[10] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In Pascal Fontaine and Amit Goel, editors, *SMT*

2012. 10th International Workshop on Satisfiability Modulo Theories, volume 20 of *EPiC Series in Computing*, pages 44–56. EasyChair, 2013.

[11] Alexander Nadel. Bit-vector rewriting with automatic rule generation. In *Computer Aided Verification*, pages 663–679. Springer, 2014.

[12] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2015.

[13] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 78–90. Springer, 2003.

[14] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Term Rewriting and Applications*, pages 453–468. Springer, 2005.

[15] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.

[16] Lawrence C Paulson. Defining functions on equivalence classes. *ACM Transactions on Computational Logic (TOCL)*, 7(4):658–675, 2006.

[17] Christian Reisenberger. Pboolector: A parallel smt solver for qf bv by combining bit-blasting and look-ahead. Master's thesis, Johannes Kepler University, Linz, 2014.

[18] Robert E Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

Acknowledgments

First I would like to thank my supervisor Armin Biere for the impressive ideas and valuable feedback on my work. A special thanks goes to my colleague and second advisor Mathias Preiner. Thank you for the discussions about various problems and the detailed explanation of Boolector.

Further I would like to thank all my colleagues, friends and family members who supported me during my studies and gave me a constant motivation.

Curriculum Vitae

Personal Information

Name	Christoph Sperl
Address	Schauberg 10, 5242 St. Johann am Walde
Telephone	+43 664 400 11 78
Email	`ch.sperl@gmx.at`
Nationality	Austrian
Date of Birth	December 10, 1990
Gender	male

Education

2015–2016	Master in Computer Science, Johannes Kepler University, Linz, Austria.
2014	Exchange Semester, Oxford Brookes University, Oxford, UK.
2011–2015	Bachelor in Computer Science, Johannes Kepler University, Linz, Austria.
2005–2010	Matura, Höhere Technische Bundeslehranstalt, Braunau am Inn, Austria.
2001–2005	Hauptschule, St. Johann am Walde, Austria.
1997–2001	Volksschule, St. Johann am Walde, Austria.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 30. Mai 2016

Christoph Sperl