JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

C32SAT:
A satisfiability checker for C expressions

MAGISTERARBEIT

zur Erlangung des akademischen Grades

DIPLOM-INGENIEUR

im Magisterstudium

INFORMATIK

Angefertigt am Institut für Formale Modelle und Verifikation

Eingereicht von:

Robert Daniel Brummayer Bakk. techn.

Betreuung:

Univ.-Prof. Dr. Armin Biere

Linz, Februar 2006

Abstract

Formal verification becomes more and more important as software complexity increases. In this thesis we propose the use of a formal verification tool called "C32SAT". We discuss its implementation and underlying concepts. The question which this thesis addresses is whether C32SAT's functional representation of boolean C expressions can be used to check satisfiability efficiently or not.

First we give a motivation for formal verification. Then we discuss the satisfiability problem of propositional logic, And-Inverter graphs, two level optimisation and transformations into conjunctive normal form. Afterwards we discuss C32SAT, its input language, semantics, operators, design decisions, architecture, algorithms, related work and benchmarks.

Keywords

Formal verification, Satisfiability problem (SAT problem), Conjunctive Normal Form (CNF), Negation Normal Form (NNF), And-Inverter Graph (AIG), Two level optimisation, Two level contradiction rules for And-Inverter graphs, Two level subsumption rules for And-Inverter graphs, Two level idempotency rules for And-Inverter graphs, Two level resolution rule for And-Inverter graphs, Transformations into conjunctive normal form, Tseitin transformation, Tseitin Transformation on And-Inverter graphs.

Contents

Chapter 1

Motivation for formal verification

The complexity of software systems is steadily increasing. The main reason is that computers are used for more complex tasks than in the past. Computers are used for concurrent and distributed tasks like scientific computation, multimedia-based computation and simulation. Thus, complex software is needed to accomplish these requirements. Another reason is the need for more and more features. Standard software products like operating systems are good examples.

There is an obvious relation between number of features, complexity, lines of code and number of bugs:

1. Adding more features leads to more complex software
2. More complex software leads to more lines of code
3. More lines of code lead to more bugs

This corresponds to Tanenbaum's first law of software:

> Adding more code adds more bugs [12].

The increasing complexity of software leads to a demand for new and more efficient techniques for software verification. Established methods like unit testing are still useful, but often not sufficient for verifying complex systems. By using formal techniques and concepts like finite state machines, labeled transition systems and petri nets it can be shown that software has no defects and satisfies its specification.

Model checking is a popular formal verification method for finite systems. By using model checking techniques it is possible to find inconsistencies already in the model. Generally, system models are represented by automatas and specifications by formulas expressed in temporal logic. The fact that inconsistencies can be found already in the model is economically advantageous. Model checking is discussed in [21], [19], [8] and [20].

1.1 Domains of formal software verification

There are many domains where software failures are intolerable. Examples are:

- Medical systems
- Aerospace systems
- Car systems
- E-Commerce
- Operating systems

The example of the Ariane 5 rocket shows which disaster a bug can cause. This example can be found in section 1.1 in [21]. An overflow in a variable of the control software caused the control system to fail and thus the rocket to explode. The financial loss was enormous.

1.2 Examples

We discuss two examples which should motivate for formal verification.

1.2.1 Swapping

The following software fragment shows how two integer variables can be swapped. In the programming language C this swapping can be done without using an additional variable:

```
...
int x;
int y;
/* assign arbitrary values to x and to y */
...
/* swap x with y */
x ^= y;
y ^= x;
x ^= y;
/* The values of x and y have been swapped */
...
```

The question is how the correctness of this code fragment can be verified. Of course every programmer could write a short program containing this fragment

and could try some assignments. The problem is that by trying out some assignments it cannot be verified that there does not exist an assignment where this code fragment does not work correctly. If 32 bit integers are used, then there are $2^{32} \times 2^{32} = 2^{64}$ possible assignments.Trying all possible assignments *explicitly* is impractical.

This is exactly the reason why formal verification becomes more and more important. The correctness of this code fragment can be shown by using a tool for formal verification like C32SAT.

Consider the static single assignment form of the code fragment:

```
...
int x0, x1, x2, x3;
int x1, y1, y2, y3;
/* assign arbitrary values to x0 and to y0 */
...
/* swap x with y */
x1 = (x0 ^ y0); y1 = y0; /* x ^= y; */
x2 = x1; y2 = (y1 ^ x1); /* y ^= x; */
x3 = (x2 ^ y2); y3 = y2; /* x ^= y; */
/* x3 == y0 and y3 == x0 */
...
```

Static single assignment (SSA) form means that every variable receives exactly one assignment during its lifetime. This form is often used in the context of optimising compilers. In this example the static single assignment form was derived manually. Nevertheless it could also be done automatically. The authors in [5] present an efficient way for computing SSA forms.

The question if this code fragment is correct can be transformed into the following question:

> Is it always the case in the 32 bit domain that if (x1 == (x0 ^ y0) and y1 == y0 and x2 == x1 and y2 == (y1 ^ x1) and x3 == (x2 ^ y2) and y3 == y2), then (x3 == y0 and y3 == x0)?

This question can be answered by C32SAT. First, the formula has to be encoded into the syntax of C32SAT:

```
(x1 == (x0 ^ y0) && y1 == y0
&&
x2 == x1 && y2 == (y1 ^ x1)
&&
x3 == (x2 ^ y2) && y3 == y2)
=>
(x3 == y0 && y3 == x0)
```

The syntax of C32SAT is discussed in section 5.2. We write this C32SAT formula into the file swapwithxor.c32sat. Now C32SAT can be used to verify whether this formula is tautological or not. If this formula is tautological then there does not exist any counter example. We call C32SAT in a specific mode where it verifies whether the input is tautological or not:

c32sat -t swapwithxor.c32sat

C32SAT yields:

FORMULA IS TAUTOLOGICAL

It has been shown by C32SAT that there does not exist any counter example in the 32 bit domain where this code fragment does not work correctly. Thus, we can conclude that the original code fragment is correct.

1.2.2 Overflow

Consider the following code fragment in the programming language C:

```
...
int x, y, z;
/* assign an arbitrary value x0 to x */
/* assign an arbitrary value y0 to y */
...
if (y != 0) {
  z = x / y;
} else {
  z = x;
}
...
```

This simple code fragment looks safe and a programmer might think that a program containing this code fragment always behaves as expected. Unfortunately, this is not the case. If we assume that x, y and z are 32 bit integers then there is one out of 2^{64} assignments where the value of z is undefined after executing this code fragment. The term "undefined" means in this context that the value can be unpredictable and depends on compiler semantics. C32SAT can be used to find such dangerous assignments. We encode this code fragment into the syntax of C32SAT:

```
(y != 0) ? (x / y) : x
```

We write this C32SAT formula into the file undef.c32sat. We call C32SAT in a specific mode where it verifies whether the result of this expression is always defined according to the C99 standard [10]. We call C32SAT in the following way:

c32sat -ad undef.c32sat

C32SAT yields:

THE RESULT OF THE FORMULA IS NOT ALWAYS DEFINED (C99)
COUNTER-EXAMPLE:
y = -1
x = -2147483648

C32SAT has found an assignment to the variables x and y where the result of this expression is undefined according to the C99 standard. This result is assigned to the variable z in the original code fragment. If the value of z is undefined then the program may behave in an unexpected way if the program uses the value of z after the code fragment.

The explanation why this assignment leads to an undefined result is the following. If y is equal to -1 then x / y is computed and returned. The operation -1 / -2147483648 leads to an overflow, because the result is equal to 2147483648 which is not representable in a signed 32 bit integer variable. The behaviour on signed integer overflow is undefined according to the C99 standard. The resulting value of an operation where a signed integer overflow occurs can be unpredictable [10].

If we want to port this code fragment to another platform, then we have to be cautious. The C99 standard does not define whether a C compiler has to use sign and magnitude, one's complement or two's complement representation [10]. Imagine the C compiler on the current platform uses two's complement representation and we want to port our fragment to another platform where the C compiler uses sign and magnitude representation. Then the result of the assignment y = -1 and x = -2147483648 may lead to different results which can be a serious problem.

There is only one out of 2^{64} assignments which can lead to a different result. Thus, it can be the case that a program containing the fragment runs for months on both platforms as expected. Unfortunately, it may be the case that after a certain time period the program on the other platform behaves in an unexpected way. Finding such an inconsistency by using methods like unit testing is nearly impossible.

Chapter 2

The SAT problem of propositional logic

Before we discuss C32SAT, we have to discuss its underlying concepts. First we introduce the concepts of the SAT domain. This section can be skipped if the reader is already familiar with the concepts of the SAT domain.

2.1 Introduction

The satisfiability problem of propositional logic is roughly said to decide if there exists an assignment to the variables of a propositional formula so that the formula evaluates to true. The satisfiability problem is often just called the "SAT problem". We discuss the SAT problem formally. Let ϕ be an arbitrary propositional formula:

- A literal is a propositional variable or its logical negation.
- $Var(\phi)$ is defined as the set of all variables appearing in ϕ.
- A variable assignment of ϕ is a mapping α: $Var(\phi) \mapsto \{\top, \bot\}$ of the variable set of ϕ to truth values.
- The set of all possible variable assignments of ϕ is denoted by $Assign(\phi)$.
- Let $|S|$ be the number of elements of the set S, then $|Assign(\phi)| = 2^{|Var(\phi)|}$ holds.
- The value $Val(\phi, \alpha)$ of ϕ under an assignment α is the resulting truth value to which ϕ evaluates under α. $Val(\phi)$ is defined inductively:
  - If $\phi \equiv \top$ then $Val(\phi, \alpha) \equiv \top$
  - If $\phi \equiv \bot$ then $Val(\phi, \alpha) \equiv \bot$

  - If ϕ is a propositional variable x, then $Val(\phi, \alpha) \equiv \alpha(x)$
  - $Val(\neg\phi, \alpha) \equiv \neg Val(\phi, \alpha)$
  - $Val(\phi_1 \wedge \phi_2, \alpha) \equiv Val(\phi_1, \alpha) \wedge Val(\phi_2, \alpha)$
  - $Val(\phi_1 \vee \phi_2, \alpha) \equiv Val(\phi_1, \alpha) \vee Val(\phi_2, \alpha)$
  - $Val(\phi_1 \rightarrow \phi_2, \alpha) \equiv Val(\phi_1, \alpha) \rightarrow Val(\phi_2, \alpha)$
  - $Val(\phi_1 \leftrightarrow \phi_2, \alpha) \equiv Val(\phi_1, \alpha) \leftrightarrow Val(\phi_2, \alpha)$

- An assignment α of ϕ for which $Val(\phi, \alpha) \equiv \top$ holds is called a model of ϕ.
- The set of all models of ϕ is denoted by $Model(\phi)$.
- ϕ is called satisfiable if and only if there exists a model of ϕ.
- ϕ is called unsatisfiable if and only if there exists no model of ϕ.

Finally, the SAT problem of propositional logic can be defined:

> Given a propositional formula ϕ the SAT problem is to decide whether ϕ is satisfiable or not [24].

There is a relation between tautology and unsatisfiability. A propositional formula which is semantically equivalent to $\top$ is tautological. A propositional formula which is semantically equivalent to $\bot$ is unsatisfiable. The logical negation of an unsatisfiable formula results in a tautological formula and vice versa. This relation is very useful. For example íf we want to show that a formula is tautological, then it is sufficient to show that its logical negation is unsatisfiable. More Information about the SAT problem and its variants like the MAXSAT problem can be found in [24].

2.2 Examples

Now we discuss two instances of the SAT problem of propositional logic. This discussion should demonstrate the various aspects of the SAT problem.

2.2.1 Satisfiable instance

Let ϕ be the following propositional formula:

$$(x_1 \wedge x_2) \rightarrow (x_1 \leftrightarrow x_3)$$

The formula ϕ contains three propositional variables. Thus, $Var(\phi)$ has exactly three elements:

$$Var(\phi) = \{x_1, x_2, x_3\}$$

The set $Assign(\phi)$ has exactly $2^{|Var(\phi)|} = 2^3 = 8$ elements:

$$\begin{aligned} \mathit{Assign}(\phi) = \{&(\bot,\bot,\bot),(\bot,\bot,\top),(\bot,\top,\bot),(\bot,\top,\top),(\top,\bot,\bot),(\top,\bot,\top),\\ &(\top,\top,\bot),(\top,\top,\top)\} \end{aligned}$$

A tuple notation is used to enumerate $\mathit{Assign}(\phi)$. Every tuple represents an assignment. For example the tuple $(\top,\top,\bot)$ represents the assignment $x_1 = \top$, $x_2 = \top$ and $x_3 = \bot$.

The set $\mathit{Model}(\phi)$ has seven elements:

$$\begin{aligned} \mathit{Model}(\phi) = \{&(\bot,\bot,\bot),(\bot,\bot,\top),(\bot,\top,\bot),(\bot,\top,\top),(\top,\bot,\bot),(\top,\bot,\top),\\ &(\top,\top,\top)\} \end{aligned}$$

Due to the fact that $\mathit{Model}(\phi) \nsubseteq \emptyset$ and $|\mathit{Model}(\phi)| > 0$ holds, it can be concluded that there exists at least one model of ϕ. Thus, ϕ is satisfiable.

2.2.2 Unsatisfiable instance

Let ϕ be the following propositional formula:

$$(x_3 \vee x_4 \vee \neg x_3) \rightarrow (x_1 \wedge x_2 \wedge \neg x_1)$$

The formula ϕ contains four propositional variables. Thus, $\mathit{Var}(\phi)$ has exactly four elements:

$$\mathit{Var}(\phi) = \{x_1, x_2, x_3, x_4\}$$

The set $\mathit{Assign}(\phi)$ has exactly $2^{|\mathit{Var}(\phi)|} = 2^4 = 16$ elements:

$$\begin{aligned} \mathit{Assign}(\phi) = \{&(\bot,\bot,\bot,\bot),(\bot,\bot,\bot,\top),(\bot,\bot,\top,\bot),(\bot,\bot,\top,\top),\\ &(\bot,\top,\bot,\bot),(\bot,\top,\bot,\top),(\bot,\top,\top,\bot),(\bot,\top,\top,\top),\\ &(\top,\bot,\bot,\bot),(\top,\bot,\bot,\top),(\top,\bot,\top,\bot),(\top,\bot,\top,\top),\\ &(\top,\top,\bot,\bot),(\top,\top,\bot,\top),(\top,\top,\top,\bot),(\top,\top,\top,\top)\} \end{aligned}$$

It can be shown that ϕ does not have a model. This can be done by using basic propositional rules which can be found in table C.7.

Due to the fact that the operators $\wedge$ and $\vee$ are commutative, the semantically equivalent formula ϕ_1 can be obtained:

$$(x_3 \vee \neg x_3 \vee x_4) \rightarrow (x_1 \wedge \neg x_1 \wedge x_2)$$

By using rule 7 on the subformula $x_1 \wedge \neg x_1$ on the right side of the implication and rule 8 on the subformula $x_3 \vee \neg x_3$ on the left side of the implication, we obtain the semantically equivalent formula ϕ_2:

$$(\top \vee x_4) \rightarrow (\bot \wedge x_2)$$

We use the fact that the operators $\wedge$ and $\vee$ are commutative and obtain the semantically equivalent formula ϕ_3:

$$(x_4 \vee \top) \rightarrow (x_2 \wedge \bot)$$

By using rule 1 on the subformula $x_2 \wedge \bot$ on the right side of the implication and rule number 4 on the subformula $x_4 \vee \top$ on the left side of the implication we obtain the semantically equivalent formula ϕ_4:

$$\top \rightarrow \bot$$

Now the result of the implication operator can be evaluated, because its operand are constant truth values. The semantics of the implication operator can be found in table C.5. Finally, the semantically equivalent formula ϕ_5 can be obtained:

$$\bot$$

ϕ is unsatisfiable, because ϕ is semantically equivalent to ϕ_5 which always evaluates to $\bot$. The assignment of the variables are of no importance, because they do not even occur in ϕ_5. Due to the fact that $\phi \equiv \phi_5$ holds every occurrence of ϕ can be substituted by ϕ_5. This can lead to a significant simplification if ϕ occurs itself in other formulas.

A model of ϕ_5 cannot exist, because the formula ϕ_5 always evaluates to $\bot$. Due to the fact that $\phi \equiv \phi_5$, it can be concluded that this is also the case for ϕ. Thus, ϕ is unsatisfiable.

2.3 The complexity of the SAT problem

Cook proved in [4] that the SAT problem of propositional logic is *NP* complete. The complexity class P contains all problems which can be solved by a deterministic Turing machine in polynomial time. The complexity class *NP* contains all problems which can be solved by a non-deterministic Turing machine in polynomial time. Obviously, $P \subseteq NP$ holds. If $P = NP$ holds is one of the most famous questions in computer science today.

The hardness of the SAT problem can be demonstrated by the fact that the number of possible assignments of a given propositional formula grows exponentially with the number of its variables. This is exactly the reason why even small instances of the SAT problem cannot be solved by an exhaustive algorithm in reasonable time. For example let ϕ be a propositional formula which has 100 variables. An exhaustive SAT algorithm would need to evaluate the result of 2^{100} = 1267650600228229401496703205376 possible assignments to the variables of ϕ in the worst case. This is impractical.

2.4 Negation normal form

A propositional formula ϕ is in Negation Normal Form (NNF) if and only if one of the following two cases hold:

1. The operator $\neg$ does not occur in ϕ
2. The operator $\neg$ occurs in ϕ, but only in front of propositional variables

In section 1.5 in [20] the authors present an algorithm which takes an arbitrary propositional formula as input and generates a semantically equivalent NNF as result.

2.5 Conjunctive normal form

Before we discuss software in the domain of SAT, we introduce the concept of the conjunctive normal form. Most of the tools in the domain of the SAT problem expect the input to be in conjunctive normal form.

A propositional formula ϕ is in Conjunctive Normal Form (CNF) if and only if the formula consists of a conjunction of clauses which are a disjunction of literals [20].

Generally, the number of literals per clause can vary. A variant of the SAT problem is characterised by the fact that the number of literals per clause is constant. For example the problem to decide whether a given CNF where every clause has exactly three literals is satisfiable or not is called the "3-SAT problem". This problem is discussed in section 10.3 in [22].

It is worth mentioning that a formula in CNF is also in NNF. The opposite does not hold. For example the formula $(x \wedge y) \vee (\neg x \wedge z)$ is in NNF but not in CNF.

The main advantage of CNF is that it can be determined quickly if a given formula evaluates to $\bot$ under a concrete assignment. Once a clause can be found which evaluates to $\bot$, it can be concluded that the whole formula evaluates to $\bot$.

2.6 SAT solver

A SAT solver takes an arbitrary SAT problem as input. The solver computes whether the input is satisfiable or not. Additionally, if the input is satisfiable, then most solvers print the corresponding model which has been found.

Popular SAT solvers are based on the second version of the Davis-Putnam procedure (DPLL) [17]. The first version of the Davis-Putnam procedure [18] (DP) was published 1960. The second version uses unlike the first version Unit-Resolution which is also called Boolean Constraint Propagation. Many of todays algorithms in the SAT domain use the insight which was gained by the second version of the Davis-Putnam procedure. In [1] the authors discuss various aspects of successful SAT solvers like LIMMAT and NANOSAT.

There is an active SAT community which organises meetings and competitions every year. During these events researchers present their latest algorithms and

tools. There is also an online resource for research on the SAT problem called SATLIB [9]. In [2] a report of a SAT competition can be found.

2.6.1 DIMACS file format

As already mentioned before, SAT solvers take an arbitrary SAT problem as input. This input is a CNF which is usually represented by the DIMACS file format [6]. This format consists of the following two sections:

1. preamble
2. clauses

The preamble starts with a section which can be used for comments. Comment lines begin with the character 'c' and are used for human-readable information about the SAT instance. The problem line appears after the comment section. It has the following format:

p FORMAT VARIABLES CLAUSES

The field FORMAT is used to determine the format that will be expected by the SAT solver. It should contain the word "cnf". The fields VARIABLES and CLAUSES are used to determine the number of variables and clauses of the SAT instance.

Clauses appear after the problem line. Variables are encoded as numbers starting with 1. Logical negation is represented by a minus. Clauses are represented by sequences of numbers which are terminated by 0.

Example

Let ϕ be the following CNF:

$$(x1 \vee \neg x2 \vee \neg x3) \wedge (\neg x1 \vee x4) \wedge x1$$

The DIMACS representation of ϕ could be:

```
c This is a comment line
p cnf 4 3
1 -2 -3 0
-1 4 0
1 0
```

Chapter 3

And-Inverter graphs

Now we discuss And-Inverter graphs and how an acyclic directed graph representation can be used to share syntactically equivalent subformulas. C32SAT uses And-Inverter graphs in connection with subformula sharing as internal representation of boolean C expressions.

3.1 Introduction

An And-Inverter Graph (AIG) is a directed graph which can be used to represent propositional formulas. The logical conjunction and the logical negation are the only operators which are available. It can be shown that these two operators are sufficient for representing any arbitrary propositional formula. The authors in [15] discuss boolean circuit representations like Boolean Expression Diagrams (BEDs), Reduced Boolean Circuits (RBCs) and AIGs.

An AIG provides three kinds of nodes and two kinds of edges. The three kinds of nodes are:

- Logical constants
- Propositional variables
- Logical conjunctions

The two kinds of edges are:

- Regular edges
- Inverted edges

An important aspect of AIGs is that the logical negation is represented as an edge attribute. An edge can be regular or inverted. In figures, a dot is used to represent inversion. For example Fig. 3.3 represents the formula $x \rightarrow y$ graphically.

Operator	Representation by $\wedge$ and $\neg$
$\neg x$	$\neg x$
$x \wedge y$	$x \wedge y$
$x \vee y$	$\neg(\neg x \wedge \neg y)$
$x \rightarrow y$	$\neg(x \wedge \neg y)$
$x \leftrightarrow y$	$\neg(x \wedge \neg y) \wedge \neg(\neg x \wedge y)$

Table 3.1: Representation of the logical operators by $\wedge$ and $\neg$

Figure 3.1: Graphical representation of x (left) and $\neg x$ (right) by an AIG

It has to be possible to represent the operators $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$ by the operators $\wedge$ and $\neg$ to be able to represent any arbitrary formula by an AIG. Table 3.1 shows how this representation can look like. The operator $\neg$ and the operator $\wedge$ can be represented by themselves. Fig. 3.1 shows a graphical representation of $\neg x$ and Fig. 3.2 shows a graphical representation of $x \wedge y$. By using the De Morgan rules, a representation of the operator $\vee$ by the operators $\wedge$ and $\neg$ can be found. Fig. 3.2 shows a graphical representation of $x \vee y$. The operator $\rightarrow$ can be eliminated by using the rule $x \rightarrow y \equiv \neg x \vee y$. Fig. 3.3 shows a graphical representation of $x \rightarrow y$. Finally, the operator $\leftrightarrow$ can be eliminated by using the rule $x \leftrightarrow y \equiv (x \rightarrow y) \wedge (y \rightarrow x)$. Fig. 3.4 shows a graphical representation of $x \leftrightarrow y$.

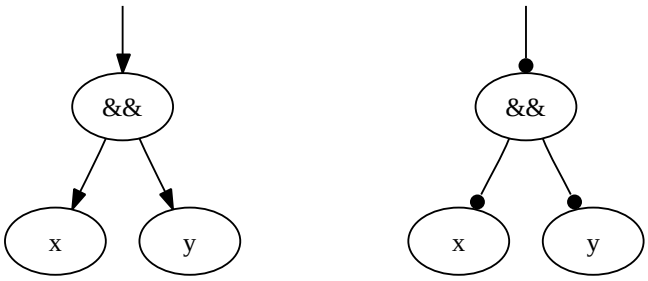

Figure 3.2: Graphical representation of $x \wedge y$ (left) and $x \vee y$ (right) by an AIG

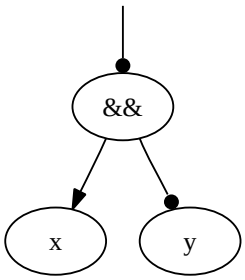

Figure 3.3: Graphical representation of $x \rightarrow y$ by an AIG

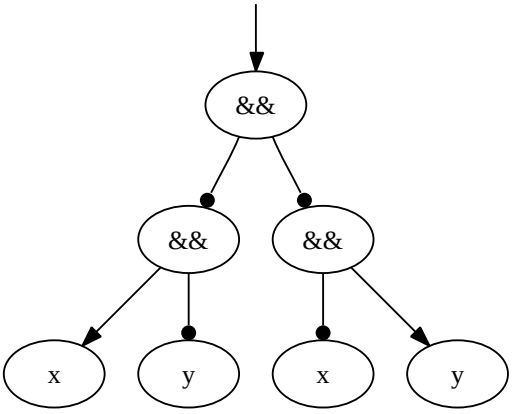

Figure 3.4: Graphical representation of $x \leftrightarrow y$ by an AIG

3.2 Example

Now we discuss an example which shows the representation of a propositional formula by an AIG. Let ϕ be the following propositional formula:

$$(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$$

The equivalences in table 3.1 are used to obtain a semantically equivalent formula which uses only the operators $\wedge$ and $\neg$.

First we eliminate the equivalence operator:

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

Then we eliminate the right implication operator:

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

Then we eliminate the disjunction operator:

$$(\neg(a \wedge \neg b) \wedge \neg(\neg a \wedge b)) \rightarrow (\neg(\neg a \wedge b) \wedge \neg(a \wedge \neg b))$$

Finally, we eliminate the implication operator:

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

This formula uses only the operators $\wedge$ and $\neg$ and is semantically equivalent to the original formula. Now this formula can be represented by an AIG. This representation is shown in Fig. 3.5.

3.3 DAG representation

The AIGs which have been discussed so far have all been trees. A tree is a graph which is directed, acyclic and has exactly $n + 1$ nodes where n is the number of edges. Another kind of graph which can also be used to represent AIGs is called Directed Acyclic Graph (DAG). The only difference between a tree and a DAG is that a DAG does not have the restriction that it must have exactly one node more than edges.

A DAG representation of an AIG is particularly useful if the corresponding formula has syntactically equivalent subformulas. In a tree, equal subformulas are represented as often as they occur in the formula. No sharing is possible. This means a waste of computer memory. If a DAG is used instead of a tree, then sharing syntactically equal subformulas is possible. The only disadvantage of a DAG representation is that it has to be computed whether subformulas are syntactically equivalent or not. This can be done by using a hash table.

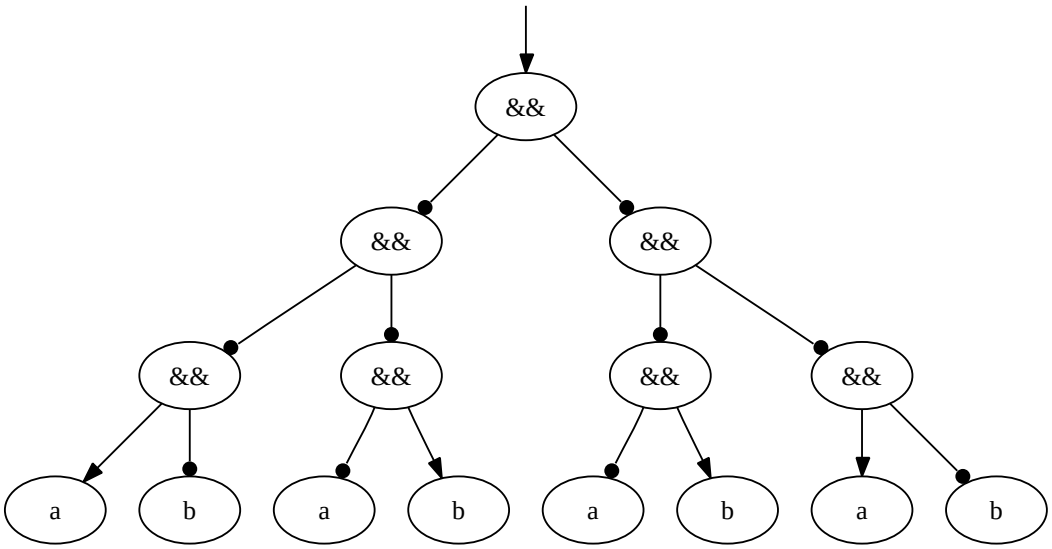

Figure 3.5: Graphical representation of $(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$ by an AIG

Fig. 3.6 shows the DAG representation of the following formula which has already been presented. It shows how syntactically equivalent subformulas can be shared.

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

3.4 Optimisation

3.4.1 Two level minimisation

Local optimisation methods like the two level minimisation operate on a local area of the graph and have only local information available. They can be combined with global optimisation methods which work on the whole graph and have global information available. Local optimisation methods like the two level minimisation can be used before node creation to reduce the size of the graph. Afterwards global optimisation methods can be applied to reduce the size even more.

Two level minimisation can be used as a local optimisation before node creation in connection with subformula sharing [15]. The following formula is an example of a two level minimisation rule which can be found in [15]:

$$(a \leftrightarrow c) \vee (a \leftrightarrow d) \vee (b \leftrightarrow c) \vee (b \leftrightarrow d) \vdash ((a \vee b) \wedge (c \wedge d)) \leftrightarrow (c \wedge d)$$

If we know that at least one of the premises holds, then we can replace $(a \vee b) \wedge (c \wedge d)$ by $c \wedge d$. For example if we know that b and c represent the same shared

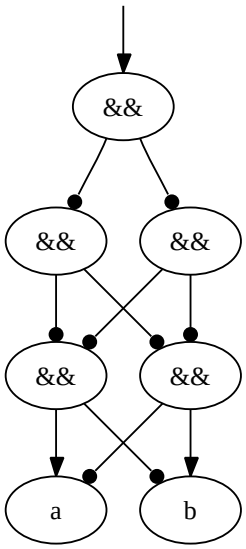

Figure 3.6: Graphical representation of $(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$ by an AIG-DAG

subformula then we can conclude that $(a \vee b) \wedge (c \wedge d)$ can be replaced by $(c \wedge d)$. This example is shown in Fig. 3.7. It shows the original graph (left) and the optimised graph (right).

3.4.2 Two level minimisation rules for AIGs

We propose not to do all possible two level optimisations for AIGs, but only certain optimisations which do not affect global subformula sharing negatively. Our rules are expected to be used as local optimisations before node creation in connection with global subformula sharing. Benchmarks can be found in section 6.1.

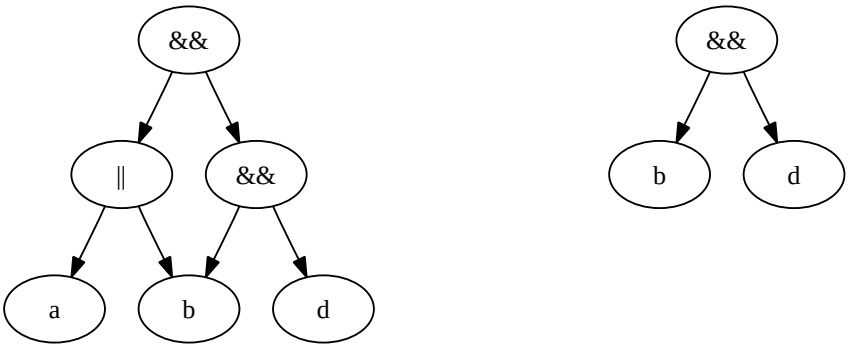

Figure 3.7: Example of a two level minimisation rule

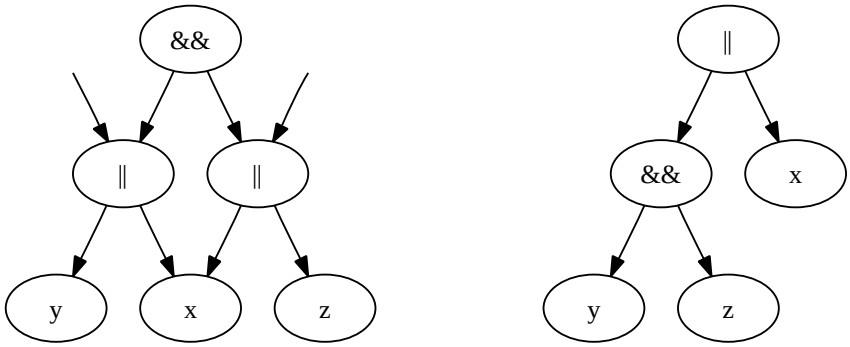

Figure 3.8: Distributivity rule affecting global subformula sharing negatively

Fig. 3.8 shows an example of a rule where global subformula sharing is affected negatively. This rule is the rule of distributivity from $\vee$ over $\wedge$ which can be found in table C.7. The original graph is shown left and the graph after applying the rule of distributivity is shown right. The additional arrows which point to the two disjunctions indicate that the subformulas $x \vee y$ and $x \vee z$ are already shared by other subformulas.

The graph after applying the rule of distributivity needs one operator node less than the original graph. From a local point of view this is clearly an optimisation. Unfortunately, this need not be the case from a global point of view. Remember that local optimisation rules are used before node creation. If we do not apply the distributivity rule, then we have to generate only one new subformula. This new subformula is the conjunction of the two disjunctions. If we apply the distributivity rule, then we have to generate *two* new subformulas. These new subformulas are $y \wedge z$ and $x \vee (y \wedge z)$. Generally, this effect leads to more shared subformulas and thus to a bigger CNF. This effect is is discussed in more detail in [15]. This is exactly the reason why we propose the use of local optimisation rules which do not affect global subformula sharing negatively.

Our optimisation rules can be divided into the following four classes:

1. Contradiction
2. Subsumption
3. Idempotency
4. Resolution

It is assumed that also one level minimisation is done. This one level minimisation uses basic rules of propositional logic like the rule of idempotency which can be found in section C.7. Additionally we assume that this one level minimisation uses the fact that logical conjunction is commutative. For example the

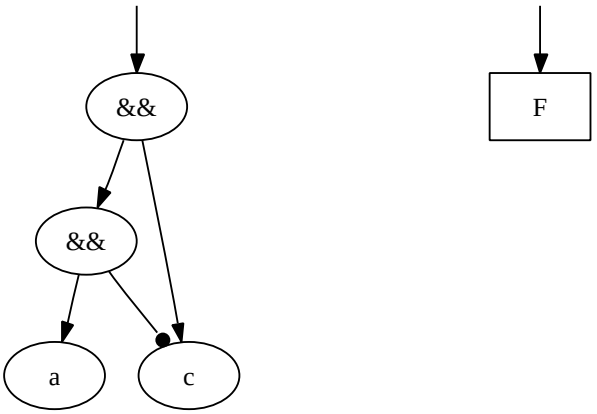

Figure 3.9: Example of the first rule of contradiction

formula $(a \wedge b) \wedge \neg(b \wedge a)$ should be optimised by one level minimisation, because the subformulas $(a \wedge b)$ and $(b \wedge a)$ can be represented by one shared node.

Rules of contradiction

The first rule of contradiction can be written in the following way:

$$\neg(a \leftrightarrow c) \vee \neg(b \leftrightarrow c) \vdash ((a \wedge b) \wedge c) \leftrightarrow \bot$$

If we know that at least one of the premises holds, then we can replace $(a \wedge b) \wedge c$ by $\bot$. For example if we know that b represents the logical negation of the shared subformula c, then we can conclude that $(a \wedge b) \wedge c$ can be replaced by $\bot$. This example is shown in Fig. 3.9. It shows the original graph (left) and the optimised graph (right).

The second rule of contradiction can be written in the following way:

$$\neg(a \leftrightarrow c) \vee \neg(a \leftrightarrow d) \vee \neg(b \leftrightarrow c) \vee \neg(b \leftrightarrow d) \vdash ((a \wedge b) \wedge (c \wedge d)) \leftrightarrow \bot$$

If we know that at least one of the premises holds, then we can replace $(a \wedge b) \wedge (c \wedge d)$ by $\bot$. For example if we know that c represents the logical negation of the shared subformula b, then we can conclude that $(a \wedge b) \wedge (c \wedge d)$ can be replaced by $\bot$. This example is shown in Fig. 3.10. It shows the original graph (left) and the optimised graph (right).

Rules of subsumption

The first rule of subsumption can be written in the following way:

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

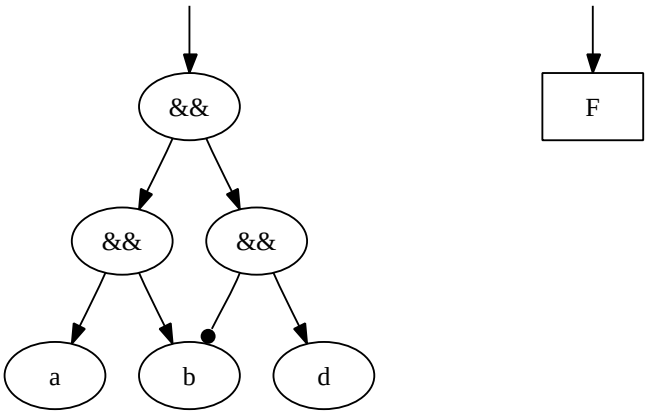

Figure 3.10: Example of the second rule of contradiction

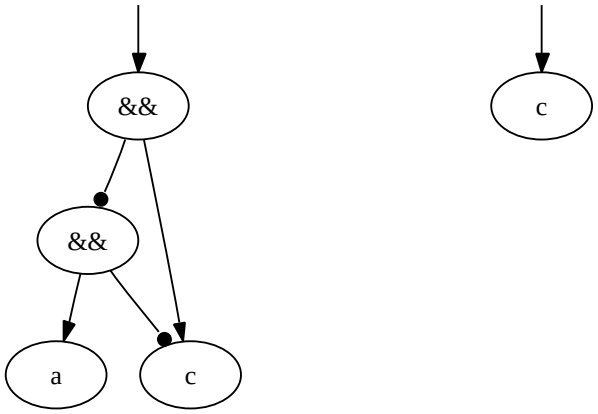

Figure 3.11: Example of the first rule of subsumption

If we know that at least one of the premises holds, then we can replace $\neg(a \wedge b) \wedge c$ by c. For example if we know that b represents the logical negation of the shared subformula c, then we can conclude that $\neg(a \wedge b) \wedge c$ can be replaced by c. This example is shown in Fig. 3.11. It shows the original graph (left) and the optimised graph (right).

The second rule of subsumption can be written in the following way:

$$\neg(a \leftrightarrow c) \vee \neg(a \leftrightarrow d) \vee \neg(b \leftrightarrow c) \vee \neg(b \leftrightarrow d) \vdash (\neg(a \wedge b) \wedge (c \wedge d)) \leftrightarrow (c \wedge d)$$

If we know that at least one of the premises holds, then we can replace $\neg(a \wedge b) \wedge (c \wedge d)$ by $c \wedge d$. For example if we know that c represents the logical negation of the shared subformula b, then we can conclude that $\neg(a \wedge b) \wedge (c \wedge d)$ can be replaced by $c \wedge d$. This example is shown in Fig. 3.12. It shows the original graph (left) and the optimised graph (right).

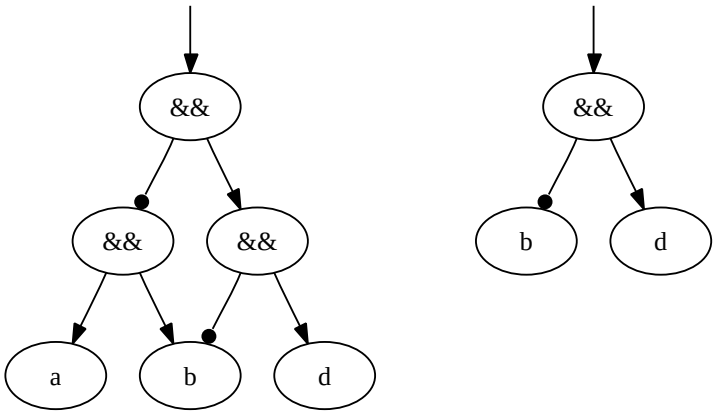

Figure 3.12: Example of the second rule of subsumption

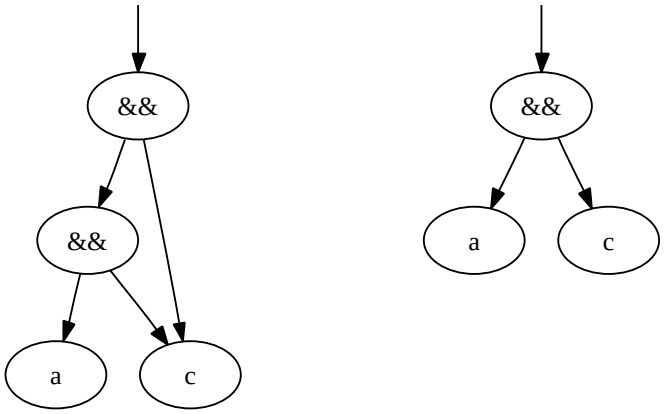

Figure 3.13: Example of the first rule of idempotency

Rules of idempotency

The first rule of idempotency can be written in the following way:

$$(a \leftrightarrow c) \vee (b \leftrightarrow c) \vdash ((a \wedge b) \wedge c) \leftrightarrow (a \wedge b)$$

If we know that at least one of the premises holds, then we can replace $(a \wedge b) \wedge c$ by $a \wedge b$. For example if we know that b and c represent the same shared subformula, then we can conclude that $(a \wedge b) \wedge c$ can be replaced by $a \wedge b$. This example is shown in Fig. 3.13. It shows the original graph (left) and the optimised graph (right).

The second rule of idempotency can be written in the following way:

$$(a \leftrightarrow c) \vee (b \leftrightarrow c) \vdash ((a \wedge b) \wedge (c \wedge d)) \leftrightarrow ((a \wedge b) \wedge d)$$

If we know that at least one of the premises holds, then we can replace $(a \wedge b) \wedge (c \wedge d)$ by $(a \wedge b) \wedge d$. For example if we know that b and c represent the same

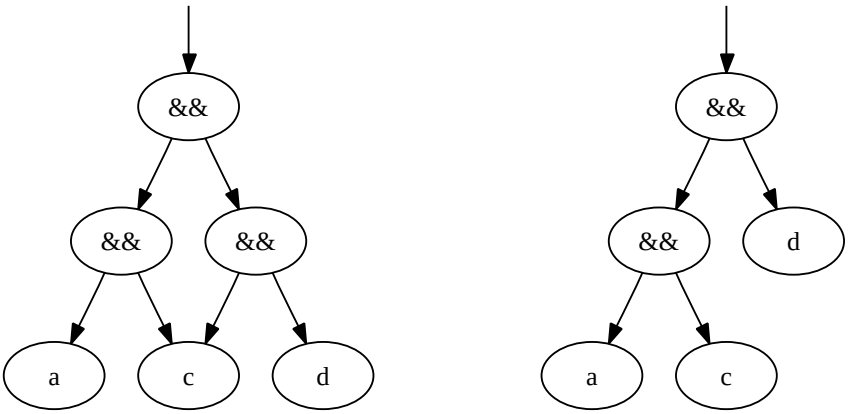

Figure 3.14: Example of the second rule of idempotency

shared subformula, then we can conclude that $(a \wedge b) \wedge (c \wedge d)$ can be replaced by $(a \wedge b) \wedge d$. This example is shown in Fig. 3.14. It shows the original graph (left) and the optimised graph (right).

Rule of resolution

The rule of resolution can be written in the following way:

$$(a \leftrightarrow d) \wedge \neg(b \leftrightarrow c) \vdash (\neg(a \wedge b) \wedge \neg(c \wedge d)) \leftrightarrow \neg a$$

If we know that the premise holds, then we can replace $\neg(a \wedge b) \wedge \neg(c \wedge d)$ by $\neg a$. For example if we know that a and d represent the same shared subformula and b represents the logical negation of the shared subformula c, then we can conclude that $\neg(a \wedge b) \wedge \neg(c \wedge d)$ can be replaced by $\neg a$. This example is shown in Fig. 3.15. It shows the original graph (left) and the optimised graph (right).

Selection criteria

Our original selection criterion was that the optimised graph is a constant or consists of a subformula which is already part of the original AIG. This criterion guarantees that global subformula sharing is not affected negatively. The rules of contradiction and subsumption accomplish this criterion. Whether the rule of resolution accomplishes this criterion or not, depends on how negation is implemented. For example in the programming language C, negation can be represented by flipping the least significant bit of the pointer. If this bit is set to one, then the pointer represents an inverted edge. If this bit is set to zero, then the pointer represents a regular edge. Of course we have to reset this bit, if we want to access the data to which the pointer points to. This implementation is possible, because todays computers align memory addresses word by word. If

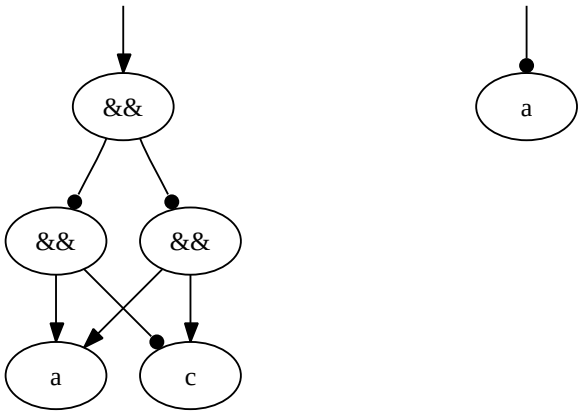

Figure 3.15: Example of the rule of resolution

such an implementation is used, then the rule of resolution also accomplishes this selection criterion. We verified manually that we found all two level minimisation rules which accomplish this criterion.

Nevertheless, other criteria for rules which do not affect global subformula sharing negatively can be found. The rules of idempotency simply use the underlying concept of idempotency which leads to an optimised AIG. This optimised AIG needs one conjunction node less than the original AIG.

Chapter 4

Transformations into CNF

Now we discuss how an arbitrary propositional formula can be transformed into CNF. We discuss the standard approach and the Tseitin transformation. C32SAT uses the Tseitin transformation to transform its internal expression representation into CNF.

4.1 The standard approach

The standard way to transform an arbitrary propositional formula into CNF is the following algorithm [20]. Let ϕ be an arbitrary propositional formula:

1. Eliminate every occurrence of the operator $\leftrightarrow$ in ϕ by using the rule $x \leftrightarrow y \equiv (x \rightarrow y) \wedge (y \rightarrow x)$.
2. Eliminate every occurrence of the operator $\rightarrow$ in ϕ by using the rule $x \rightarrow y \equiv \neg x \vee y$.
3. Use the De Morgan rules which can be found in table C.8.
4. Finally, use the distributivity rules 11 and 12 in table C.7 to transform ϕ into CNF.

This algorithm has the disadvantage that the size of the resulting formula can explode exponentially. Fortunately, there are other transformations available. In [14] the authors discuss how the exponential explosion can be prevented by using heuristic techniques.

4.2 Tseitin transformation

The Tseitin transformation [13] transforms an arbitrary propositional formula into CNF. Unlike the standard approach, the resulting formula is not semantically equivalent to the original formula. The reason is that the Tseitin transformation

introduces new propositional variables. The relation between the original and the resulting formula is the following: If the resulting formula is satisfiable, then the original formula is satisfiable and vice versa.

The Tseitin transformation works in the following way: It is assumed that the formula is represented by a tree. First we traverse the tree and introduce a new propositional variable for every operator. These variables are placeholders for the results of the corresponding operations. Afterwards we traverse the tree a second time. During this traversion we generate a CNF subformula for every operator. Additionally, we generate a CNF subformula which binds the result of the top level operator to $\top$. Finally, we combine all subformulas by using conjunction.

The number of new variables grows linearly with the number of operators in the original formula. Unfortunately, these variables lead to the effect that the resulting formula is not semantically equivalent to the original formula. For example if the original formula is tautological, then the resulting formula is not tautological anymore. This fact is demonstrated by an example later.

4.2.1 Tseitin transformation rules

Now we introduce the rules of the Tseitin transformation. These rules are used to generate the corresponding subformulas. The variable x represents the new variable which is introduced for the corresponding operator. The variables y and z represent the operands.

$x \leftrightarrow \neg y \equiv (\neg x \vee \neg y) \wedge (x \vee y)$
$x \leftrightarrow (y \wedge z) \equiv (\neg x \vee y) \wedge (\neg x \vee z) \wedge (x \vee \neg y \vee \neg z)$
$x \leftrightarrow (y \vee z) \equiv (x \vee \neg y) \wedge (x \vee \neg z) \wedge (\neg x \vee y \vee z)$
$x \leftrightarrow (y \rightarrow z) \equiv (x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$
$x \leftrightarrow (y \leftrightarrow z) \equiv (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y \vee z)$

4.2.2 Example

The concepts of the Tseitin transformation are demonstrated by an example. Let ϕ be the following formula:

$$(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$$

The truth table 4.1 shows that this formula is tautological. This formula can be represented by a binary tree. First we traverse[1] this tree. During this traversion we introduce a new variable for every operator. The result of this traversion is shown in Fig. 4.1. The new variables are written in parenthesis below the operators. Afterwards we traverse the tree a second time. During this traversion

[1] The kind of traversion is of no importance. We use in-order traversion in our examples.

a	b	$a \leftrightarrow b$	$a \vee \neg b$	$a \rightarrow b$	$(a \vee \neg b) \wedge (a \rightarrow b)$	ϕ
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 4.1: Truth table of $(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$

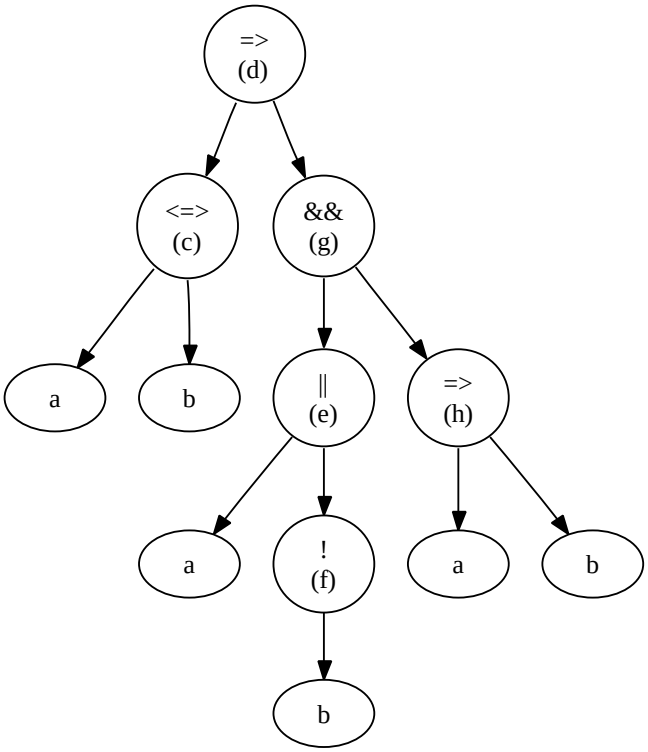

Figure 4.1: Tree representation of $(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$

we use the corresponding Tseitin rule on every operator to generate the corresponding subformula. Then we generate the subformula which binds the result of the top level operator to $\top$. After this we combine all subformulas using logical conjunction:

$((\neg c \vee \neg a \vee b) \wedge (\neg c \vee a \vee \neg b) \wedge (c \vee \neg a \vee \neg b) \wedge (c \vee a \vee b)) \wedge ((d \vee c) \wedge (d \vee \neg g) \wedge (\neg d \vee \neg c \vee g)) \wedge ((e \vee \neg a) \wedge (e \vee \neg f) \wedge (\neg e \vee a \vee f)) \wedge ((\neg f \vee \neg b) \wedge (f \vee b)) \wedge ((\neg g \vee e) \wedge (\neg g \vee h) \wedge (g \vee \neg e \vee \neg h)) \wedge ((h \vee a) \wedge (h \vee \neg b) \wedge (\neg h \vee \neg a \vee b)) \wedge d$

Finally, we eliminate unnecessary parenthesis:

$(\neg c \vee \neg a \vee b) \wedge (\neg c \vee a \vee \neg b) \wedge (c \vee \neg a \vee \neg b) \wedge (c \vee a \vee b) \wedge (d \vee c) \wedge (d \vee \neg g) \wedge (\neg d \vee \neg c \vee g) \wedge (e \vee \neg a) \wedge (e \vee \neg f) \wedge (\neg e \vee a \vee f) \wedge (\neg f \vee \neg b) \wedge (f \vee b) \wedge (\neg g \vee e) \wedge (\neg g \vee h) \wedge (g \vee \neg e \vee \neg h) \wedge (h \vee a) \wedge (h \vee \neg b) \wedge (\neg h \vee \neg a \vee b) \wedge d$

It can be shown that the resulting formula is unlike the original formula not tautological. For example if we assign $\bot$ to the variable d then the whole formula evaluates to $\bot$. Thus, the formula cannot be tautological.

4.2.3 Transformation on circuits

Due to the fact that basic hardware gates correspond to logical operators, the Tseitin transformation can also be used on circuits. This can be very useful if semantical equivalence of circuits has to be verified.

Consider the following situation. Imagine a tool is used to optimise circuits in the way that the optimised circuit needs less gates than the original circuit. A company uses this tool on one of its main circuits. Now it can be the case that this tool has a bug and produces a result which is mostly but not totally equivalent to the original circuit. The company wants to be sure that the optimised circuit is semantically equivalent to the original circuit. Trying out a few assignments to the inputs of the two circuits is not sufficient to show that the two circuits are semantically equivalent. One verification method, which uses the Tseitin transformation, is the following:

We connect every output of the original circuit with the corresponding output of the optimised circuit by using XOR-gates. All these XOR-gates are combined by using disjunction. If there is an assignment to the inputs of the two circuits where at least one output is different, then at least one XOR-gate evaluates to $\top$ and thus the disjunction of the XOR-gates evaluates to $\top$.

Now the question if two circuits are semantically equivalent can be answered by solving a satisfiability problem. The problem is if there exists an assignment to the inputs, so that the disjunction of the XOR-gates evaluates to $\top$. This problem can be solved by using the Tseitin transformation on a representation of the circuit. For example this representation can be an AIG. Performing the

Tseitin transformation on And-Inverter graphs is discussed in section 4.2.4. The resulting CNF can be passed to a SAT solver which yields whether the formula is satisfiable or not. If the formula is satisfiable, then the two circuits are not semantically equivalent and one obtains a helpful counter example where the outputs of the two circuits are different. If the formula is unsatisfiable, then it has been verified that the two circuits are semantically equivalent.

4.2.4 Transformation on AIGs

The Tseitin transformation can be used to transform an arbitrary AIG into CNF. Due to the fact that AIGs represent negation by an edge attribute the rules of the Tseitin transformation have to be adapted.

AIG Tseitin transformation rules

We introduce the rules of the AIG Tseitin transformation. These rules are used to generate the corresponding subformulas. The variable x represents the new variable which is introduced for the corresponding conjunction operator. The variables y and z represent the operands.

$$x \leftrightarrow (y \wedge z) \equiv (\neg x \vee y) \wedge (\neg x \vee z) \wedge (x \vee \neg y \vee \neg z)$$
$$x \leftrightarrow (y \wedge \neg z) \equiv (\neg x \vee y) \wedge (\neg x \vee \neg z) \wedge (x \vee \neg y \vee z)$$
$$x \leftrightarrow (\neg y \wedge z) \equiv (\neg x \vee \neg y) \wedge (\neg x \vee z) \wedge (x \vee y \vee \neg z)$$
$$x \leftrightarrow (\neg y \wedge \neg z) \equiv (\neg x \vee \neg y) \wedge (\neg x \vee \neg z) \wedge (x \vee y \vee z)$$

The step where we generate a subformula for every operator has to be adapted. Now we have to check whether the edges to the operands are negated or not, so we can choose the corresponding rule. The rest of the Tseitin transformation algorithm remains unchanged.

If the AIG is a DAG, then one has to be cautious. It has to be made sure that every node is visited only once. This can be done by using a hash table or a flag which determines if one node has already been visited. Using a DAG in the context of the Tseitin transformation has an advantage. The Tseitin transformation profits from the fact that syntactically equivalent subformulas can be shared in a DAG. Generally, this leads to a shorter CNF. The next example demonstrates this effect.

Example

Consider the following formula:

$$(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$$

The AIG-tree representation and the AIG-DAG representation have already been discussed in section 3.2 and section 3.3. First we show the Tseitin transformation

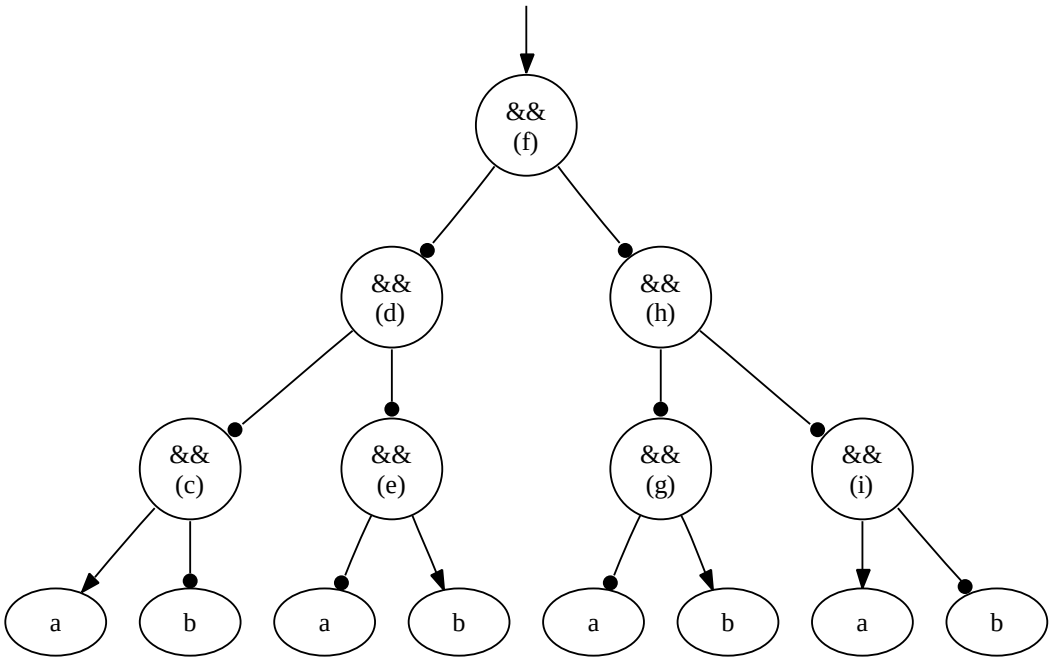

Figure 4.2: AIG-tree representation of $(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$

on a tree representation. Afterwards we show the Tseitin transformation on a DAG representation.

Now we discuss the Tseitin transformation on AIG-trees. We traverse the tree and introduce a new variable for every operator. The result of this traversion is shown in Fig. 4.2. The new variables are written in parenthesis below the operators. Afterwards we traverse the tree a second time. During this traversion we use the corresponding Tseitin rule on every operator to generate the corresponding subformula. Then we generate the subformula which binds the result of the top level operator to $\top$. After this we combine all subformulas by using logical conjunction:

$$((\neg c \vee a) \wedge (\neg c \vee \neg b) \wedge (c \vee \neg a \vee b)) \wedge ((\neg d \vee \neg c) \wedge (\neg d \vee \neg e) \wedge (d \vee c \vee e)) \wedge ((\neg e \vee \neg a) \wedge (\neg e \vee b) \wedge (e \vee a \vee \neg b)) \wedge ((\neg f \vee \neg d) \wedge (\neg f \vee \neg h) \wedge (f \vee d \vee h)) \wedge ((\neg g \vee \neg a) \wedge (\neg g \vee b) \wedge (g \vee a \vee \neg b)) \wedge ((\neg h \vee \neg g) \wedge (\neg h \vee \neg i) \wedge (h \vee g \vee i)) \wedge ((\neg i \vee a) \wedge (\neg i \vee \neg b) \wedge (i \vee \neg a \vee b)) \wedge f$$

Finally, we eliminate unnecessary parenthesis:

$$(\neg c \vee a) \wedge (\neg c \vee \neg b) \wedge (c \vee \neg a \vee b) \wedge (\neg d \vee \neg c) \wedge (\neg d \vee \neg e) \wedge (d \vee c \vee e) \wedge (\neg e \vee \neg a) \wedge (\neg e \vee b) \wedge (e \vee a \vee \neg b) \wedge (\neg f \vee \neg d) \wedge (\neg f \vee \neg h) \wedge (f \vee d \vee h) \wedge (\neg g \vee \neg a) \wedge (\neg g \vee b) \wedge (g \vee a \vee \neg b) \wedge (\neg h \vee \neg g) \wedge (\neg h \vee \neg i) \wedge (h \vee g \vee i) \wedge (\neg i \vee a) \wedge (\neg i \vee \neg b) \wedge (i \vee \neg a \vee b) \wedge f$$

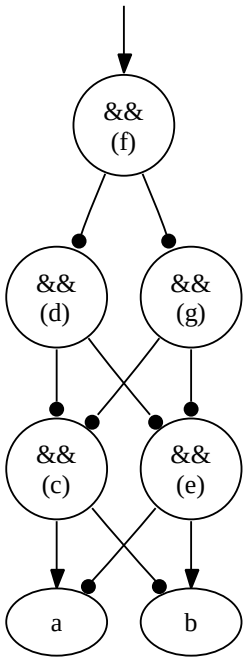

Figure 4.3: AIG-DAG representation of $(a \leftrightarrow b) \rightarrow ((a \vee \neg b) \wedge (a \rightarrow b))$

Due to the fact that equal subformulas are not shared in a tree, the Tseitin transformation treats these subformulas as different subformulas. For example the variables e and g represent the same subformula $(\neg a \wedge b)$.

Now we discuss the Tseitin transformation on AIG-DAGs. We traverse the DAG and introduce a new variable for every operator. The result of this traversion is shown in Fig. 4.3. The new variables are written in parenthesis below the operators. Afterwards we traverse the DAG a second time. During this traversion we use the corresponding Tseitin rule on every operator to generate the corresponding subformula. Then we generate the subformula which binds the result of the top level operator to $\top$. After this we combine all subformulas using logical conjunction:

$((\neg c \vee a) \wedge (\neg c \vee \neg b) \wedge (c \vee \neg a \vee b)) \wedge ((\neg d \vee \neg c) \wedge (\neg d \vee \neg e) \wedge (d \vee c \vee e)) \wedge ((\neg e \vee \neg a) \wedge (\neg e \vee b) \wedge (e \vee a \vee \neg b)) \wedge ((\neg f \vee \neg d) \wedge (\neg f \vee \neg g) \wedge (f \vee d \vee g)) \wedge ((\neg g \vee \neg c) \wedge (\neg g \vee \neg e) \wedge (g \vee c \vee e)) \wedge f$

Finally, we eliminate unnecessary parenthesis:

$(\neg c \vee a) \wedge (\neg c \vee \neg b) \wedge (c \vee \neg a \vee b) \wedge (\neg d \vee \neg c) \wedge (\neg d \vee \neg e) \wedge (d \vee c \vee e) \wedge (\neg e \vee \neg a) \wedge (\neg e \vee b) \wedge (e \vee a \vee \neg b) \wedge (\neg f \vee \neg d) \wedge (\neg f \vee \neg g) \wedge (f \vee d \vee g) \wedge (\neg g \vee \neg c) \wedge (\neg g \vee \neg e) \wedge (g \vee c \vee e) \wedge f$

If we compare the results of the tree and the DAG representation, then we can see that the resulting CNF of the DAG representation is shorter. The reason is that shared subformulas are only transformed once. Usually, AIG-DAG representations are more compact than AIG-tree representation and lead to a shorter CNF, because syntactically equivalent subformulas can be shared.

Chapter 5

C32SAT

5.1 Introduction

C32SAT is a tool for formal verification. The main idea of C32SAT is to generate a SAT instance out of a boolean C expression[1]. It encodes the expression into a circuit represented by AIGs. Then this circuit is transformed into CNF by the Tseitin transformation and passed to an efficient SAT solver. Finally, C32SAT interprets the result of the SAT solver and returns the corresponding C32SAT result. If verification fails, then C32SAT supplies a concrete counter example.

Generally, C32SAT is applicable in the domain of formal verification. Nevertheless, it is not limited to this domain. Additional examples where C32SAT can be used are:

- Satisfiability checking
- Tautology checking
- Equivalence checking
- Constraint computation
- Algebraic equations computation

5.2 Input language

C32SAT's input language is defined by the C32SAT grammar. The C32SAT grammar is a context free grammar which has the LL(1) property. It is inspired by grammars of common programming languages like C, C++ and Java. The grammar of the C programming language can be found in appendix A in [16].

[1] A boolean C expression is an expression in the programming Language C which evaluates to $\top$ (integer value $\neq 0$) or $\bot$ (integer value $= 0$).

The C32SAT grammar is defined by the following rules in extended Backus-Naur form:

```
c32sat = Ite.
Ite = Imp [ "?" Imp ":" Imp ].
Imp = Or { ImpOp Or }.
Or = And { "||" And }.
And = BOr { "&&" BOr }.
BOr = BXor { "|" BXor }.
BXor = BAnd { "^" BAnd }.
BAnd = Eq { "&" Eq }.
Eq = Rel { EqOp Rel }.
Rel = Shift { RelOp Shift }.
Shift = Add { ShiftOp Add }.
Add = Mul { AddOp Mul }.
Mul = Neg { MulOp Neg }.
Neg = { NegOp } Basic.
Basic = Ident | Integer | "(" Ite ")".

ImpOp = "=>" | "<=>".
EqOp = "==" | "!=".
RelOp = "<" | "<=" | ">" | ">=".
ShiftOp = ">>" | "<<".
AddOp = "+" | "-".
MulOp = "*" | "/" | "%".
NegOp = "!" | "-" | "~".

Number = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
Integer = (( "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" )
          { Number } ) | ( "0" ).
Char = "a" | "b" | "c" | ... | "z" | "A" | "B"| "C" | ... | "Z" | "_".
Ident = Char { Char | Number }.
```

Context free grammars are often used to define the syntax of programming languages. Context free grammars are more powerful than regular grammars and are sufficient for most problems. The LL(1) property means that the parser can choose between alternatives by using just one look ahead token. Context free grammars and the LL(1) property are discussed in [23].

5.3 Data types

C32SAT does not know any explicit data type. By default every variable has the data type of a signed 32 bit integer. This design decision has been made, because the main idea of C32SAT is to generate a SAT instance out of a boolean C expression. In the programming language C there is no boolean data type. An integer value not equal to zero represents the boolean value $\top$ and an integer value equal to zero represents the boolean value $\bot$. Thus, the data type integer is needed to be able to deal with boolean C expressions.

We decided to use signed integers so C32SAT can also deal with negative numbers. We could also have taken unsigned integers, but then C32SAT would not be able to treat negative numbers. Nevertheless, the implementation of the operators would not differ much from the current implementation which uses signed integer operands. The implementation of the conditional operator, the logical operators, the bitwise operators and the equality operators would be identical. Only the implementation of the unary minus operator, the shift operators, the additive operators and the multiplication operators would have to be adapted, because they use the fact that two's complement representation is used.

5.3.1 Internal representation

An integer variable is represented by an AIG vector. Every element points to an AIG which represents a boolean variable. Two's complement is used to encode negative numbers.

During an analysis phase C32SAT looks for variables which are only used in a boolean context. This can be done by inspecting the parse tree if there are only boolean operators used in the context of the corresponding variable. If this is the case, then only the element of the vector which represents the least significant bit has to point to an AIG which represents a boolean variable. The other elements point to the constant AIG_FALSE which is the AIG representation of $\bot$. This optimisation is correct, because if a variable is only used in boolean context then it is sufficient that this variable can represent the values zero and one. Other optimisations like inspecting the range of a variable and use less than 32 bits in a dynamic approach are possible, but non-trivial.

5.4 Undefined results and overflows

There are many cases in the programming language C where the result of an operation is not fully defined according to the C99 standard. This is a serious problem if one wants to write portable programs. By using C32SAT it can be verified that there is no assignment to the variables of a boolean C expression, so that the result is undefined. We use the term "undefined" in a broad sense.

The C99 standard uses terms like “undefined behaviour“, “unspecific behaviour“ and “implementation-defined behaviour“. In C32SAT the term “undefined“ is used when the resulting value of an operation is not fully defined and depends on compiler semantics. An example for undefined behaviour is shifting an integer by a negative value.

According to the C99 standard the result of a signed integer overflow is undefined and depends on compiler semantics. A two's complement representation cannot be guaranteed. Nevertheless, in some cases it can be useful to use such integer overflows if one is aware of the compiler overflow semantics. It can also be useful if C32SAT is used to verify real circuits. C32SAT can be configured to treat integer overflows as two's complement overflows or as undefined values (default). The overflow treatment of C32SAT can be configured by a specific command line option. This command line option is shown in A.2.4.

5.5 Operators

Table 5.1 shows all C32SAT operators. L-R means that operators which have equal priority are evaluated from left to right. R-L means that operators which have equal priority are evaluated from right to left. Nearly every operator is also available in the programming language C. These operators have the same relative priority. The only operators which are not available in the programming language C are the logical implication operator and the logical equivalence operator which have priority two.

We decided that the two new logical operators should have higher priority than the conditional operator and less priority than the other logical operators. We could also have decided to swap the priorities between the conditional operator and the new logical operators, but this would lead to a disjoint set of binary logical operators which is not sound.

We discuss the implementation of all operators in the next sections. The implementation which is shown is basic and does not treat undefined values and overflows. The focus is on the algorithms. We also discuss the extended semantics of these operators. Extended semantics are needed to be able to deal with undefined and unpredictable results. The default semantics of the operators are defined in the C99 standard.

In the tables which define the extended semantics U represents an undefined value, O represents an overflow, INT_MIN represents the smallest signed integer value and INT_MAX represents the the greatest signed integer value[2]. The variables a, b and c represent arbitrary signed integer variables.

Generally, there are two main possibilities to encode whether a value is undefined or not. The first variant is to extend propositional logic so that there are

[2]The values of INT_MIN and INT_MAX depend on the bit width.

Priority	Operator	Associativity
13	! - ~	R-L
12	* / %	L-R
11	+ -	L-R
10	>> <<	L-R
9	< <= > >=	L-R
8	== !=	L-R
7	&	L-R
6	^	L-R
5	|	L-R
4	&&	L-R
3	||	L-R
2	=> <=>	L-R
1	?	-

Table 5.1: Overview of all C32SAT operators

three possible values: $\top$, $\bot$ and undefined. This allows treatment of undefined results at bit level. The second possibility is to use one additional bit which determines whether a whole bit vector is undefined or not. We decided to implement the second possibility, because there is no need to treat undefined results at bit level. If the result of an operation is undefined according to the C99 standard, then the whole result is undefined. It is never the case that only some bits of the result are undefined.

5.5.1 Pseudo code notes

Before we discuss the extended semantics and the implementation of the operators we have to discuss some issues according to pseudo code. The pseudo code which we use is a dialect of the C programming language. Outgoing parameters are denoted by the keyword *out*.

The basic functions *and_aig*, *or_aig*, *imp_aig*, *dimp_aig*[3], *invert_aig*, and *xor_aig* are the AIG representations of $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$, and XOR. The constants AIG_TRUE and AIG_FALSE are the AIG representations of $\top$ and $\bot$. The integer variable *number_of_bits* represents the number of bits which is used for the bit width. Additionally it is assumed that every element of an AIG vector is initialised with AIG_FALSE.

The following functions occur frequently and have to be defined:

[3]dimp means double implication

Operand 1	Operand 2	Operand 3	Condition	Result
a	b	c	-	$a\ ?\ b : c$
a	b	U	$a \neq 0$	b
a	b	U	$a = 0$	U
a	U	c	$a \neq 0$	U
a	U	c	$a = 0$	c
U	b	c	$b = c$	b
U	b	c	$b \neq c$	U
U	b	U	-	U
U	U	c	-	U
U	U	U	-	U

Table 5.2: Extended semantics of ?

```
AIG disjunction_aig_vector (AIGVector aig_vector){
  AIG result = AIG_FALSE;
  for (int i = 0; i < number_of_bits; i++){
    result = or_aig (result, aig_vector[i]);
  }
  return result;
}


AIG conjunction_aig_vector (AIGVector aig_vector){
  AIG result = AIG_TRUE;
  for (int i = 0; i < number_of_bits; i++){
    result = and_aig (result, aig_vector[i]);
  }
  return result;
}
```

5.5.2 Conditional operator

Table 5.2 defines the extended semantics of the conditional operator. The function *conditional* shows the implementation of the conditional operator in pseudo code:

Operand 1	Operand 2	Condition	Result
a	b	-	a && b
a	U	$a \neq 0$	U
a	U	$a = 0$	0
U	b	$b \neq 0$	U
U	b	$b = 0$	0
U	U	-	U

Table 5.3: Extended semantics of &&

```
AIGVector conditional (AIGVector condition, AIGVector if_case,
                       AIGVector else_case){
  AIGVector result;
  AIG aig_cond = disjunction_aig_vector (condition);
  for (int i = 0; i < number_of_bits; i++){
    result[i] = or_aig (and_aig (if_case[i], aig_cond),
                        and_aig (else_calse[i],
                                 invert_aig (aig_cond)));
  }
  return result;
}
```

5.5.3 Logical operators

C32SAT supports the following logical operators:

- The logical conjunction &&
- The logical disjunction ||
- The logical implication =>
- The logical equivalence <=>

The logical implication and the logical equivalence are the only operators in C32SAT which are not available in the programming language C. Nevertheless, these operators can be simulated in C easily. One has to replace expressions like x => y with !x || y and x <=> y with (!x || y) && (x || !y) or !x == !y.

The tables 5.3, 5.4, 5.5, 5.6 and 5.7 define the extended semantics of the corresponding operator. The functions *conjunction*, *disjunction*, *implication*, *equivalence* and *negation* show the implementation of the corresponding operator in pseudo code:

Operand 1	Operand 2	Condition	Result
a	b	-	$a \mid\mid b$
a	U	$a \neq 0$	1
a	U	$a = 0$	U
U	b	$b \neq 0$	1
U	b	$b = 0$	U
U	U	-	U

Table 5.4: Extended semantics of ||

Operand 1	Operand 2	Condition	Result
a	b	-	$!a \mid\mid b$
a	U	$a \neq 0$	U
a	U	$a = 0$	1
U	b	$b \neq 0$	1
U	b	$b = 0$	U
U	U	-	U

Table 5.5: Extended semantics of =>

Operand 1	Operand 2	Result
a	b	$(!a \mid\mid b)$ && $(a \mid\mid !b)$
a	U	U
U	b	U
U	U	U

Table 5.6: Extended semantics of <=>

Operand	Result
a	$!a$
U	U

Table 5.7: Extended semantics of !

```
AIGVector conjunction (AIGVector x, AIGVector y){
   AIGVector result;
   result[0] = and_aig (disjunction_aig_vector(x),
                        disjunction_aig_vector(y));
   return result;
}


AIGVector disjunction (AIGVector x, AIGVector y){
   AIGVector result;
   result[0] = or_aig (disjunction_aig_vector(x),
                       disjunction_aig_vector(y));
   return result;
}


AIGVector implication (AIGVector x, AIGVector y){
   AIGVector result;
   result[0] = or_aig (invert_aig(disjunction_aig_vector(x)),
                       disjunction_aig_vector(y));
   return result;
}


AIGVector equivalence (AIGVector x, AIGVector y){
   AIGVector result;
   result[0] = and_aig (or_aig (invert_aig(disjunction_aig_vector(x)),
                                disjunction_aig_vector(y)),
                        or_aig (disjunction_aig_vector(x),
                                invert_aig (disjunction_aig_vector(y))));
   return result;
}


AIGVector negation (AIGVector x){
   AIGVector result;
   result[0] = invert_aig(disjunction_aig_vector(x));
   return result;
}
```

Operand 1	Operand 2	Result
a	b	a & b
a	U	U
U	b	U
U	U	U

Table 5.8: Extended semantics of &

Operand 1	Operand 2	Result
a	b	$a \mid b$
a	U	U
U	b	U
U	U	U

Table 5.9: Extended semantics of |

5.5.4 Bitwise operators

C32SAT supports the following bitwise operators:

- The bitwise conjunction &
- The bitwise inclusive disjunction |
- The bitwise exclusive disjunction ^
- The bitwise complement ~

The tables 5.8, 5.9, 5.10 and 5.11 define the extended semantics of the corresponding operator. The functions *b_conjunction*, *b_disjunction*, *b_xor* and *b_complement* show the implementation of the corresponding operator in pseudo code.

Operand 1	Operand 2	Result
a	b	a ^ b
a	U	U
U	b	U
U	U	U

Table 5.10: Extended semantics of ^

Operand	Result
a	$\tilde{a}$
U	U

Table 5.11: Extended semantics of~

```
AIGVector b_conjunction (AIGVector x, AIGVector y){
  AIGVector result;
  for (int i = 0; i < number_of_bits; i++){
    result[i] = and_aig (x[i], y[i]);
  }
  return result;
}


AIGVector b_disjunction (AIGVector x, AIGVector y){
  AIGVector result;
  for (int i = 0; i < number_of_bits; i++){
    result[i] = or_aig (x[i], y[i]);
  }
  return result;
}


AIGVector b_xor (AIGVector x, AIGVector y){
  AIGVector result;
  for (int i = 0; i < number_of_bits; i++){
    result[i] = xor_aig (x[i], y[i]);
  }
  return result;
}


AIGVector b_complement (AIGVector x){
  AIGVector result;
  for (int i = 0; i < number_of_bits; i++){
    result[i] = invert_aig(x[i]);
  }
  return result;
}
```

Operand 1	Operand 2	Result
a	b	$a == b$
a	U	U
U	b	U
U	U	U

Table 5.12: Extended semantics of ==

Operand 1	Operand 2	Result
a	b	a != b
a	U	U
U	b	U
U	U	U

Table 5.13: Extended semantics of !=

5.5.5 Equality operators

C32SAT supports the following equality operators:

- The equality operator ==
- The inequality operator !=

Table 5.12 defines the extended semantics of the equality operator and table 5.13 defines the extended semantics of the inequality operator. The functions *equal* and *not_equal* show the implementation of the corresponding operator in pseudo code.

```
AIGVector equal (AIGVector x, AIGVector y){
  AIGVector result;
  AIG equal_aig = AIG_TRUE;
  for (int i = 0; i < number_of_bits; i++){
     equal_aig = and_aig (dimp_aig (x[i], y[i]), equal_aig);
  }
  result[0] = equal_aig;
  return result;
}
```

Operand 1	Operand 2	Result
a	b	$a < b$
a	U	U
U	b	U
U	U	U

Table 5.14: Extended semantics of $<$

Operand 1	Operand 2	Result
a	b	$a <= b$
a	U	U
U	b	U
U	U	U

Table 5.15: Extended semantics of $<=$

```
AIGVector not_equal (AIGVector x, AIGVector y){
  AIGVector result, temp;
  temp = equal (x, y);
  result[0] = invert_aig (temp[0]);
  return result;
}
```

5.5.6 Relational operators

C32SAT supports the following relational operators:

- The less-than operator $<$
- The less-than-or-equal operator $<=$
- The greater-than operator $>$
- The greater-than-or-equal operator $>=$

The tables 5.14, 5.15, 5.16 and 5.17 define the extended semantics of the corresponding operator. The functions *less*, *less_eq*, *greater* and *greater_eq* show the implementation of the corresponding operator in pseudo code. The implementation is inspired by the principle of ripple carry adders. This inspiration can be seen in the function *ripple_compare_aig* which is used to implement the relational operators.

Operand 1	Operand 2	Result
a	b	$a > b$
a	U	U
U	b	U
U	U	U

Table 5.16: Extended semantics of $>$

Operand 1	Operand 2	Result
a	b	$a >= b$
a	U	U
U	b	U
U	U	U

Table 5.17: Extended semantics of $>=$

```
void ripple_compare_aig (AIG x, AIG y, AIG lt_in, AIG eq_in,
                         AIG gt_in, out AIG lt_out,
                         out AIG eq_out, out AIG gt_out){
  AIG lt_temp = and_aig (lt_in,
                         and_aig (invert_aig (eq_in),
                                  invert_aig(gt_in)));
  AIG eq_temp = and_aig (invert_aig (lt_in),
                         and_aig (eq_in, invert_aig(gt_in)));
  AIG gt_temp = and_aig (invert_aig (lt_in),
                         and_aig (invert_aig(eq_in), gt_in));
  lt_out = or_aig (lt_temp, and_aig (x, invert_aig(y));
  eq_out = and_aig (eq_temp, dimp_aig (x, y));
  gt_out = or_aig (gt_temp, and_aig (invert_aig(x), y));
}


void ripple_compare_aig_vector (AIGVector x, AIGVector y, out AIG lt,
                                out AIG eq, out AIG gt){
  AIG lt_temp = and_aig (x[number_of_bits - 1],
                         invert_aig(y[number_of_bits -1]));
  AIG eq_temp = dimp_aig (x[number_of_bits - 1],
                          y[number_of_bits - 1]);
  AIG gt_temp = and_aig (invert_aig (x[number_of_bits - 1]),
                         y[number_of_bits -1]);
  for (int i = number_of_bits - 2; i >= 0; i--){
    ripple_compare_aig (x[i], y[i], lt_temp, eq_temp,
```

```
                        gt_temp, lt, eq, gt);
    lt_temp = lt;
    eq_temp = eq;
    gt_temp = gt;
  }
}
```

Finally, the implementation of the relational operators can be shown:

```
AIGVector less (AIGVector x, AIGVector y){
  AIGVector result;
  AIG lt, eq, gt;
  ripple_compare_aig_vector (x, y, lt, eq, gt);
  result[0] = lt;
  return result;
}


AIGVector less_eq (AIGVector x, AIGVector y){
  AIGVector result;
  AIG lt, eq, gt;
  ripple_compare_aig_vector (x, y, lt, eq, gt);
  result[0] = or_aig (lt, eq);
  return result;
}


AIGVector greater (AIGVector x, AIGVector y){
  AIGVector result;
  AIG lt, eq, gt;
  ripple_compare_aig_vector (x, y, lt, eq, gt);
  result[0] = gt;
  return result;
}


AIGVector greater_eq (AIGVector x, AIGVector y){
  AIGVector result;
  AIG lt, eq, gt;
  ripple_compare_aig_vector (x, y, lt, eq, gt);
  result[0] = or_aig (gt, eq);
  return result;
}
```

Operand 1	Operand 2	Condition	Result
a	b	$b < 0$	U
a	b	$b \geq$ number of bits	U
a	U	-	U
U	b	-	U
U	U	-	U

Table 5.18: Extended semantics of << part 1

Operand 1	Operand 2	Condition	Result
a	b	$INT\_MIN \leq a * 2^b \leq INT\_MAX$	$a << b$
a	b	$a * 2^b < INT\_MIN$	O
a	b	$a * 2^b > INT\_MAX$	O

Table 5.19: Extended semantics of << part 2

5.5.7 Shift operators

C32SAT supports the following shift operators:

- The left shift operator <<
- The right shift operator >>

The extended semantics of the left shift operator are defined by the tables 5.18 and 5.19. First you have to check table 5.18. If no rule matches there, then you have to check table 5.19. Finally, table 5.20 defines the extended semantics of the right shift operator. The functions *shift_left* and *shift_right* show the implementation of the corresponding operator in pseudo code. The implementation of the shift operators use the principle of barrel shifters which can be found in hardware. The functions *shift_n_bits_left* and *shift_n_bits_right* which are used by the functions *shift_left* and *shift_right* are shown first.

Operand 1	Operand 2	Condition	Result
a	b	$b < 0$	U
a	b	$b \geq$ number of bits	U
a	b	-	$a >> b$
a	U	-	U
U	b	-	U
U	U	-	U

Table 5.20: Extended semantics of >>

```
AIGVector shift_n_bits_left (AIGVector x, int bits, AIG shift){
  AIGVector result;
  for (int i = 0; i < bits; i++){
    result[i] = or_aig (and_aig (x[i],
                                 invert_aig (shift)),
                        AIG_FALSE);
  }
  for (int i = bits, i < number_of_bits; i++){
    result[i] = or_aig (and_aig (x[i],
                                 invert_aig (shift)),
                        and_aig (x[i - bits],
                                 shift));
  }
  return result;
}


AIGVector shift_n_bits_right (AIGVector x, int bits, AIG shift){
  AIGVector result;
  for (int i = 0; i < number_of_bits - bits; i++){
    result[i] = or_aig (and_aig (x[i],
                                 invert_aig (shift)),
                        and_aig (x[i + bits],
                                 shift));
  }
  for (int i = number_of_bits - bits, i < number_of_bits; i++){
    result[i] = or_aig (and_aig (x[i],
                                 invert_aig (shift)),
                        and_aig (x[number_of_bits - 1],
                                 shift));
  }
  return result;
}
```

Finally, the implementation of the shift operators can be shown. The function *log2*(x) computes the logarithm to the basis two of x and the function *pow2*(x) computes two to the power of x.

Operand	Condition	Result
a	-	$-a$
a	$a = INT\_MIN$	O
U	-	U

Table 5.21: Extended semantics of unary -

```
AIGVector shift_left (AIGVector x, AIGVector y){
   AIGVector result, temp;
   result = shift_left_n_bits (x, 1, y[0]);
   for (int i = 1; i < log2(number_of_bits); i++){
     temp = result;
     result = shift_left_n_bits (result, pow2(i), y[i]);
   }
   return result;
}
```

```
AIGVector shift_right (AIGVector x, AIGVector y){
   AIGVector result, temp;
   result = shift_right_n_bits (x, 1, y[0]);
   for (int i = 1; i < log2(number_of_bits); i++){
     temp = result;
     result = shift_right_n_bits (temp, pow2(i), y[i]);
   }
   return result;
}
```

5.5.8 Unary minus operator

C32SAT uses two's complement representation. Table 5.21 defines the extended semantics of the unary minus operator: The unary minus is implemented by calling the function *twos_complement*. The function *twos_complement* uses the function *add* which is shown in section 5.5.9 and the function *b_complement* which is shown in section 5.5.4. Finally, the function *unary_minus* shows the implementation of the unary minus operator in pseudo code.

Operand 1	Operand 2	Condition	Result
a	b	$INT\_MIN \leq a+b \leq INT\_MAX$	$a+b$
a	b	$a+b < INT\_MIN$	O
a	b	$a+b > INT\_MAX$	O
a	U	-	U
U	b	-	U
U	U	-	U

Table 5.22: Extended semantics of binary +

Operand 1	Operand 2	Condition	Result
a	b	$INT\_MIN \leq a-b \leq INT\_MAX$	$a-b$
a	b	$a-b < INT\_MIN$	O
a	b	$a-b > INT\_MAX$	O
a	U	-	U
U	b	-	U
U	U	-	U

Table 5.23: Extended semantics of binary -

```
AIGVector twos_complement (AIGVector x){
  AIGVector one;
  one[0] = AIG_TRUE;
  return add (complement(x), one);
}


AIGVector unary_minus (AIGVector x){
  return twos_complement (x);
}
```

5.5.9 Additive operators

C32SAT supports the following additive operators:

- The binary plus operator
- The binary minus operator

Table 5.22 defines the extended semantics of the binary plus operator and table 5.23 defines the extended semantics of the binary minus operator. The functions

add and *sub* show the implementation of the corresponding operator in pseudo code. C32SAT uses the fact that the expression $x - y$ is equal to the expression $x + (-y)$. Adding is implemented in the default way by concatenating full adders bit by bit. The function *full_add* shows the implementation of a full adder.

```
AIG full_add (AIG x, AIG y, AIG cin, out AIG cout){
  cout = or_aig (and_aig (x, cin),
                 or_aig (and_aig (y, cin),
                         and_aig (x, y)));
  return xor_aig (x, xor_aig (y, cin));
}


AIGVector add (AIG x, AIG y){
  AIGVector result;
  AIG temp;
  AIG carry = AIG_FALSE;
  for (int i = 0; i < number_of_bits; i++){
    result[i] = full_add (x[i], y[i], carry, temp);
    carry = temp;
  }
  return result;
}


AIGVector sub (AIG x, AIG y){
  return add (x, twos_complement(y));
}
```

5.5.10 Multiplicative operators

C32SAT supports the following multiplicative operators:

- The multiplication operator *
- The division operator /
- The modulo operator %

The tables 5.24, 5.25 and 5.26 define the extended semantics of the corresponding operator. The functions *mult*, *div* and *mod* show the implementation of the corresponding operator in pseudo code. The algorithm which is used for the multiplication is called "long multiplication". It uses the function *shift_n_bits_left*

Operand 1	Operand 2	Condition	Result
a	b	$INT\_MIN \leq a * b \leq INT\_MAX$	$a * b$
a	b	$a * b < INT\_MIN$	O
a	b	$a * b > INT\_MAX$	O
a	U	-	U
U	b	-	U
U	U	-	U

Table 5.24: Extended semantics of *

Operand 1	Operand 2	Condition	Result
a	b	-	a/b
a	b	$b = 0$	U
a	b	$a = INT\_MIN$ and $b = -1$	O
a	U	-	U
U	b	-	U
U	U	-	U

Table 5.25: Extended semantics of /

Operand 1	Operand 2	Condition	Result
a	b	-	a/b
a	b	$b = 0$	U
a	U	-	U
U	b	-	U
U	U	-	U

Table 5.26: Extended semantics of %

which is shown in section 5.5.7 and the function *add* which is shown in section 5.5.9.

The implementations of *div* and *mod* use the function *shift_and_remainder*. This function uses an algorithm which computes quotient and remainder. This algorithm can be found in section A.2 in [7]. This algorithm works only with unsigned integers, thus a normalisation of the input has to be done. The function *divide_and_remainder* uses the function *shift_n_bits_left* which is shown in section 5.5.7, the function *conditional* which is shown in section 5.5.2, the function *less* which is shown in section 5.5.6, the function *twos_complement* which is shown in section 5.5.8 and finally the functions *add* and *sub* which are shown in section 5.5.9.

```
AIGVector mult (AIGVector x, AIGVector y){
  AIGVector result;
  AIGVector temp1;
  AIGVector temp2;
  for (int i = 0; i < number_of_bits; i++){
    for (int j = 0; j < number_of_bits; j++){
      temp1 = and_aig (x[j], y[i])
    }
    temp2 = add (result,
                 shift_n_bits_left(temp1, i, AIG_TRUE);
    result = temp2;
  }
  return result;
}


void divide_and_remainder (AIGVector x, AIGVector y,
                           out AIGVector quotient,
                           out AIGVector remainder){
  AIGVector temp;
  AIGVector zero;
  AIGVector sub_result;
  AIGVector quotient_temp;
  AIGVector quotient_need_sign;
  AIGVector x_neg;
  AIGVector y_neg;
  quotient = conditional (less (x, zero),
                          twos_complement(x), x);
  divisor_neg = conditional (less (y, zero), y,
                             twos_complement(y));
  for (int i = 0; i < number_of_bits; i++){
```

```
    /* shift register pair */
    temp = shift_n_bits_left (remainder, 1, AIG_TRUE);
    remainder = temp;
    remainder[0] = quotient[number_of_bits - 1];
    temp = shift_n_bits_left (quotient, 1, AIG_TRUE);
    quotient = temp;
    /* subtract */
    sub_result = add (remainder, divisor_neg);
    /* compute quotient */
    quotient_temp = quotient;
    quotient_temp[0] = AIG_TRUE;
    temp = conditional (less(sub_result, zero),
                        quotient, quotient_temp);
    quotient = temp;
    /* restore ? */
    temp = conditional (less(sub_result, zero),
                        remainder, sub_result);
    remainder = temp;
  }
  /* sign quotient and remainder if necessary */
  x_neg = less(x, zero);
  y_neg = less (y, zero);
  quotient_need_sign[0] = xor_aig (x_neg[0], y_neg[0]);
  temp = conditional (quotient_need_sign,
                      twos_complement(quotient), quotient);
  quotient = temp;
  temp = conditional (x_neg, twos_complement(remainder),
                      remainder);
  remainder = temp;
}


AIGVector div (AIGVector x, AIGVector y){
  AIGVector quotient;
  AIGVector remainder;
  divide_and_remainder (x, y, quotient, remainder);
  return quotient;
}
```

```
AIGVector mod (AIGVector x, AIGVector y){
  AIGVector quotient;
  AIGVector remainder;
  divide_and_remainder (x, y, quotient, remainder);
  return remainder;
}
```

5.5.11 Unsupported operators

Our goal was to keep the input language of C32SAT simple so that C32SAT can be used without spending hours of time in learning the input language. This is the reason why some language details of C which makes the language very powerful, but also error prone are currently not supported in C32SAT. We summarise the operators of the programming language C which are currently not supported in C32SAT:

1. The array operator ([]), the dereferencing operator (*), the address operator (&) and the pointer operator (->) are not supported, because C32SAT does not support pointer operations.

2. The dot operator (.), the sizeof operator (sizeof) and the type cast operator are not supported, because C32SAT does not support user-defined data types.

3. The unary plus operator (+) is not supported, because it was only added to the programming language C for symmetry with the unary minus operator.

4. The assignment operator (=) and its variants (+=), the increment operator (++) and the decrement operator (--) are not supported, because C32SAT does not support operators with side effects.

5. The comma operator (,) is not supported, because it is rarely used in the programming language C.

Nevertheless, these operators are expected to be supported in future versions of C32SAT.

5.6 Division by zero

Currently, the result of a division by zero is treated as an undefined value. This is dangerous, because logical conjunction and logical disjunction can be used to mask out undefined values. Thus, formulas in which a division by zero occurs can be tautological. Consider the following C32SAT formula:

Operand 1	Operand 2	Condition	Result
a	b	-	$a \mid\mid b$
a	U	$a \neq 0$	1
a	U	$a = 0$	U
U	b	$b \neq 0$	1
U	b	$b = 0$	U
U	U	-	U
a	E	-	E
E	b	-	E
U	E	-	E
E	U	-	E
E	E	-	E

Table 5.27: Extended semantics of || including E

```
1 || (x / 0)
```

The result of the logical disjunction is true, because at least on operand is $\neq 0$. This formula is tautological although a division by zero occurs. A division by zero causes a terminating trap in real programs. Note that the C99 standard does not enforce short-circuit evaluation, so C compilers can, but do not have to support short-circuit evaluation. Thus, this expression can be dangerous in real programs, although the result of the logical disjunction can be evaluated without inspecting the subexpression $x/0$.

In future versions of C32SAT we want to introduce a flag E[4]. This flag should indicate whether a division by zero occurred or not. Table 5.27 shows how the semantics of the logical disjunction could be extended.

5.7 The four main modes

The following main modes are available:

1. Satisfiability mode
2. Tautology mode
3. Defined result mode
4. Undefined result mode

[4]E means Error

Before we discuss the semantics of these modes, we have to define the following mathematical objects:

- Let $\mathit{defined}(x)$ be a predicate which returns $\top$, if x is a defined signed integer value according to the semantics of the C32SAT operators and $\bot$ if not.
- Let $\mathit{Bool}(x)$ be a predicate which maps every signed integer $\neq 0$ to $\top$ and every signed integer $= 0$ to $\bot$.
- Let $\mathit{Val}(\phi, \alpha)$ be the signed integer value or a special value U[5] to which the C32SAT formula ϕ evaluates under the assignment α.
- Let $\mathit{Assignment}(\phi)$ be the set of all possible assignments to the variables of the C32SAT formula ϕ.

5.7.1 Satisfiability mode

By using the satisfiability mode it can be verified whether a C32SAT formula is satisfiable or not. The question which C32SAT addresses in the satisfiability mode is the following:

> Does an assignment to the variables of a given C32SAT formula exist which leads to a result which is defined and equal to true?

The semantics of the satisfiability mode can be expressed formally. Let ϕ be an arbitrary C32SAT formula. Then ϕ is satisfiable if and only if:

$$\exists \alpha \in \mathit{Assignment}(\phi) : \mathit{Bool}(\mathit{Val}(\phi, \alpha)) \wedge \mathit{defined}(\mathit{Val}(\phi, \alpha))$$

5.7.2 Tautology mode

The tautology mode can be used to verify whether a given formula is tautological or not. The question which C32SAT addresses in the tautology mode is the following:

> Does every assignment to the variables of a given C32SAT formula lead to a result which is defined and equal to true?

The semantics of the tautology mode can be expressed formally. Let ϕ be an arbitrary C32SAT formula. Then ϕ is tautological if and only if:

$$\forall \alpha \in \mathit{Assignment}(\phi) : \mathit{Bool}(\mathit{Val}(\phi, \alpha)) \wedge \mathit{defined}(\mathit{Val}(\phi, \alpha))$$

[5] This special value is used if the result is undefined.

5.7.3 Defined result mode

The defined result mode can be used to verify compliance to the C99 standard. The question which C32SAT addresses in the defined result mode is the following:

> Does every assignment to the variables of a given C32SAT formula lead to a defined result according to the C99 standard?

The semantics of the defined result mode can be expressed formally. Let ϕ be an arbitrary C32SAT formula. Then the result is always defined if and only if:

$$\forall \alpha \in Assignment(\phi) : defined(Val(\phi, \alpha))$$

5.7.4 Undefined result mode

The undefined result mode can be used to detect code fragments which are always undefined and thus dangerous. The question which C32SAT addresses in the undefined result mode is the following:

> Does every assignment to the variables of a given C32SAT formula lead to an undefined result according to the C99 standard?

The semantics of the undefined result mode can be expressed formally. Let ϕ be an arbitrary C32SAT formula. Then the result is always undefined if and only if:

$$\forall \alpha \in Assignment(\phi) : \neg defined(Val(\phi, \alpha))$$

5.7.5 Relations between the main modes

Generally, in C32SAT it is not the case that negating an unsatisfiable C32SAT formula leads to a tautological formula and vice versa. The reason is that the definitions of the semantics of the satisfiability and tautology mode contain the subformula $defined(Val(\phi, \alpha))$.

Nevertheless, if the given C32SAT formula uses only operators which cannot introduce undefined values, then the semantics of the satisfiability mode and the tautology mode can be expressed in the following way:

Semantics of the satisfiability mode:

$$\exists \alpha \in Assignment(\phi) : Bool(Val(\phi, \alpha))$$

Semantics of the tautology mode:

$$\forall \alpha \in Assignment(\phi) : Bool(Val(\phi, \alpha))$$

The reason is that the subformula $defined(Val(\phi, \alpha))$ always evaluates to $\top$. Thus, these two formulas can be simplified. It can be seen that if a C32SAT

formula uses only operators which cannot introduce undefined values, it is the case that negating an unsatisfiable C32SAT formula leads to a tautological formula and vice versa.

Let ϕ be an arbitrary C32SAT formula. Then additionally the following relations hold:

1. If ϕ is tautological, then the result of ϕ is always defined.
2. If ϕ is satisfiable, then the result of ϕ is not always undefined.
3. If the result of ϕ is always undefined, then ϕ is not tautological.
4. If the result of ϕ is always undefined, then ϕ is not satisfiable.
5. If the result of ϕ is always undefined, then the result of ϕ is not always defined.
6. If the result of ϕ is always defined, then the result of ϕ is not always undefined.

5.8 Architecture

The whole software system consists of two subsystems. The first subsystem is C32SAT itself and the second subsystem is its test suite. Fig. 5.1 shows the architecture of C32SAT. Only the most important modules are shown. Common modules which occur nearly in every software system are not shown to keep the diagram concise. For example the stack module and the linked list module are not shown. The application module containing the glue logic is also not shown, because it is obvious that it uses the most important modules.

The diagram shows how the main modules depend on each other. The modules Parser, Scanner and Parse Tree represent the frontend of C32SAT. This frontend is responsible for reading the input and converting it into a parse tree which is used by the backend afterwards. The other modules represent the backend. This backend converts the parse tree into a circuit represented by an AIG vector. The vector is transformed into a single AIG by using disjunction over the elements. Finally, the Tseitin transformation transforms this AIG into CNF which is passed to a SAT solver.

The data flow is shown in Fig. 5.2. It shows the main data flow, the corresponding modules and its transformations.

Transformation modules are used so that the corresponding modules do not depend on each other. For example the module Parse Tree does not know that the module AIGVector exists and vice versa. The only module which knows that both exist is the transformation module Parse Tree Transformation. Thus,

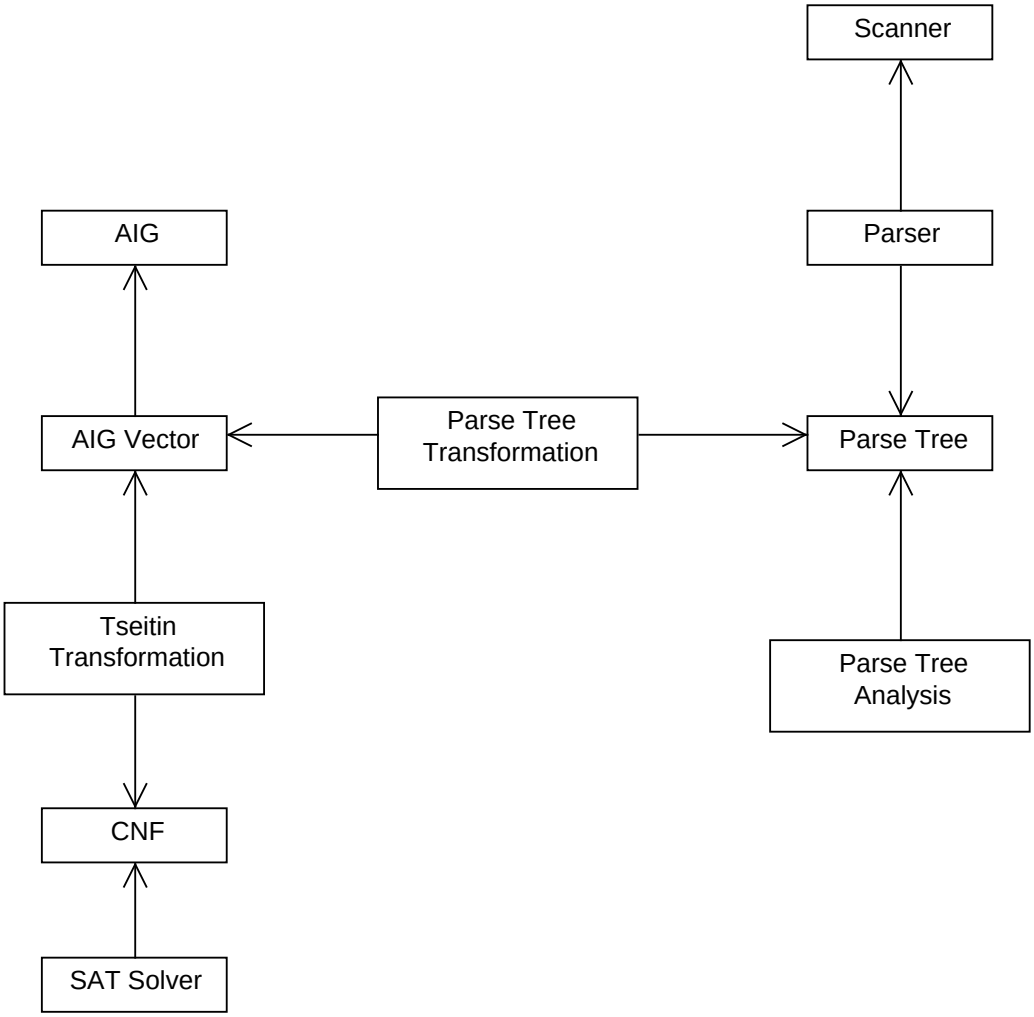

Figure 5.1: Main architecture of C32SAT

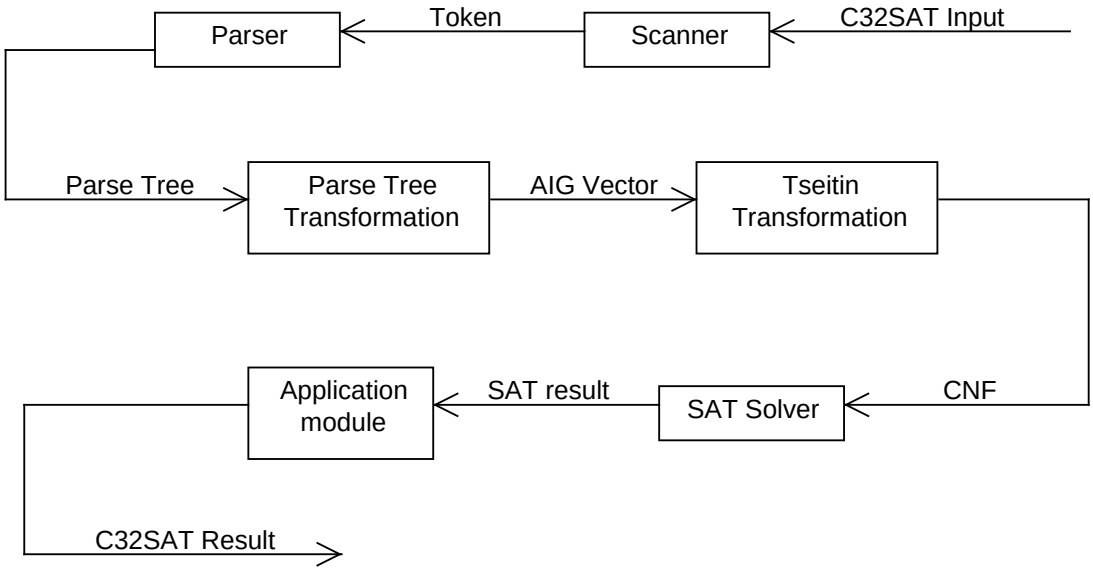

Figure 5.2: Main data flow of C32SAT

interface changes of one module do not affect the other module. The only module which has to be adapted is the transformation module.

The test suite contains a test-module for every module in the C32SAT subsystem. Every test-module contains a test-function for every function in the corresponding module. This approach is called "Unit Testing" and is currently popular in the world of high level programming languages. We made the decision to use unit testing, because this approach is reliable and applicable to our modular software architecture.

5.9 Related work

To the best of our knowledge there exists only one similar tool which does C software verification by transforming its input into propositional logic and pass it to a SAT solver afterwards. This tool is called COGENT [3]. The main differences and similarities of C32SAT and COGENT are discussed in the next section.

5.9.1 Cogent

COGENT in version 1.0 is a theorem prover which provides direct support for queries in the form of pure ANSI-C expressions together with quantifiers. COGENT builds on the source code of CBMC which is a bounded model checker for ANSI-C programs. Like C32SAT it provides machine-level accurate reasoning for the class of expressions that occur in programs and program invariants. As in C32SAT the implementation of COGENT is based on a directed compilation of expressions into propositional logic which is passed to an efficient SAT solver afterwards.

We summarise the main differences between COGENT and C32SAT:

1. COGENT supports all C99 operators. C32SAT supports a subset of the C99 operators.

2. COGENT uses non-determinism in cases where the C99 standard does not provide concrete semantics. C32SAT treats undefined behaviour by using a flag which determines whether the resulting value is undefined or not.

3. COGENT supports pointers, pointer arithmetics and unbounded arrays. C32SAT does not support pointers, pointer arithmetics and unbounded arrays.

4. COGENT supports structures and unions. C32SAT does not support structures and unions.

5. COGENT does not support 64 bit integers. C32SAT supports 64 bit integers.

6. COGENT does not support variable bit width. C32SAT supports 8, 16, 32 and 64 bits.

7. COGENT does not support an undefined result mode where one can find an assignment which leads to an unpredictable result according to the C99 standard. C32SAT supports such an undefined result mode.

8. COGENT always tries to prove that the input is valid[6] by showing that its negation is unsatisfiable. C32SAT offers satisfiability and tautology modes.

9. Regarding non-logical operators COGENT and C32SAT have different satisfiability and tautology semantics. This is a result of different treatment of undefined behaviour.

10. COGENT does not offer the option whether to treat overflows as undefined behaviour or not. C32SAT offers this option.

11. COGENT uses a relational expression representation. C32SAT uses a functional expression representation.

Examples

We show three examples demonstrating differences and similarities between COGENT and C32SAT. Consider the following example:

```
(x + 1) != (-2147483647 - 1)
```

We call COGENT on this formula and obtain the following result:

parsing
converting
solving
negation SATISFIABLE (not valid)
x=2147483647
Time: 0.015

Cogent yields that this formula is not tautological, because the logical negation is satisfiable. Additionally, COGENT yields that if we assign 2147483647 to x, then $(x + 1) == (-2147483647 - 1)$ is satisfiable. The subformula $(2147483647 + 1)$ leads to an overflow. Cogent deals with this overflow by using non-determinism. This non-determinism is encoded by using free unconstrained variables.

We call C32SAT in tautology mode on this formula and obtain the following result:

[6] COGENT uses the term valid instead of tautological

```
FORMULA IS NOT TAUTOLOGICAL
COUNTER-EXAMPLE:
x = 2147483647
```

C32SAT yields that this formula is not tautological, because there exists an assignment to the variables where the result of the formula is undefined. In section 5.7.2 the semantics of the tautology mode are defined. Although COGENT and C32SAT have different semantics they yield the same result.

As a second example we consider the following formula:

```
(x / 0) != 3
```

We call COGENT on this formula and obtain the following result:

```
parsing
converting
solving
negation SATISFIABLE (not valid)
x=2147483646
Time: 0.094
```

Cogent yields that this formula is not tautological, because the logical negation is satisfiable. The result of the subformula $(x/0)$ is not defined according to the C99 standard. Cogent deals with such undefined behaviour by using non-determinism. As a result of this non-determinism the logical negation of the formula is satisfiable.

We call C32SAT in tautology mode on this formula and obtain the following result:

```
FORMULA IS NOT TAUTOLOGICAL
COUNTER-EXAMPLE:
x = 0
```

C32SAT yields that this formula is not tautological, because there exists an assignment to the variables where the result of the formula is undefined. If we divide x by zero then the result is undefined. The comparison of inequality where at least one of the two operands is undefined leads to an undefined result. C32SAT has found an assignment where the result of the formula is undefined. Thus, this formula is not tautological according to C32SAT tautology semantics.

Finally, we consider the following formula:

```
(x / 0) || 1
```

We call COGENT on this formula and obtain the following result:

```
parsing
converting
solving
negation UNSATISFIABLE (valid)
Time: 0.094
```

We call C32SAT in tautology mode on this formula and obtain the following result:

```
FORMULA IS TAUTOLOGICAL
```

COGENT and C32SAT yield that this formula is tautological. Division by zero is treated in the same way except that COGENT uses non-determinism where C32SAT uses undefined values.

Chapter 6

Benchmarks

6.1 Two level optimisation rules for AIGs

The two level optimisation rules for AIGs which are discussed in section 3.4.2 were implemented in C32SAT. Table 6.1 shows the number of AIGs and the time measured in seconds which was needed to compute the result. The time includes SAT solving. The benchmarks compare C32SAT with C32SAT using two level minimisation rules for AIGs (default). The benchmarks were executed on a 32 Bit Microsoft Windows XP system with 512 MB RAM running on an AMD Athlon 64 3400+ processor. C32SAT used the SAT solver Booleforce 0.5.

It can be seen that in most cases the two level minimisation rules for AIGs lead to a more compact formula representation. It can also be seen that in most cases the more compact formula representation leads to a faster computation. Unfortunately, there are some cases where the SAT solver takes more time, although the AIG and the resulting CNF are more compact.

The percentage of the gained compactness varies. On the one hand there are formulas where the benefit of the optimisation is only minimal and on the other hand there are formulas where the benefit is significant. For example as a result of the two level minimisation there are formulas where the result can be computed directly without calling the SAT solver. This is the case when one of the rules of contradiction replace a subgraph by the AIG constant $\bot$. This constant propagates and leads to a constant result.

6.2 COGENT

The table 6.2 shows some benchmarks comparing C32SAT's execution time to COGENT's execution time. Benchmark 1 to 6 are tautological examples from the C32SAT test suite. Benchmarks 7 and 8 are two representative examples of the cogent benchmark suite. Benchmarks 9 to 11 are factorisation problems. Finally, benchmarks 12 to 15 are discrete logarithm computations. The execution

Kind	Size	Size with opt.	%	Time	Time with opt.
Satisfiable	182	182	100	0	0
Satisfiable	399	394	99	0	0
Satisfiable	1568	1445	92	0	0
Satisfiable	17863	10850	61	1	0
Tautological	25626	22480	88	17	73
Tautological	30808	0	0	1	0
Tautological	35144	21370	61	1	1
Not tautological	46039	30277	66	17	9
Tautological	49064	45738	93	40	41
Not tautological	51278	44985	88	4	4
Tautological	75660	33660	44	32	4
Unsatisfiable	93604	0	0	1	0

Table 6.1: Benchmarks of the two level optimisation rule for AIGs

time was measured in seconds and includes SAT solving. The benchmarks were executed on a 32 Bit Microsoft Windows XP system with 512 MB RAM running on an AMD Athlon 64 3400+ processor.

It can be seen that in many cases the execution time is nearly the same. Nevertheless, there are cases where execution time differs a lot. For example consider the benchmarks 4 and 15.

Our benchmarks show that C32SAT's functional expressions representation is at least as applicable to the domain of formal verification as COGENT's relational expression representation. There are cases where COGENT is faster and there are cases where C32SAT is faster. It is not the case that one tool is always faster than the other one. We believe that C32SAT can be used in cases where COGENT needs a lot of time and vice versa.

Number	COGENT	C32SAT
1	0	0
2	0	0
3	1	1
4	6	74
5	51	41
6	3	4
7	0	2
8	0	2
9	6	1
10	4	1
11	2	1
12	1	0
13	4	0
14	4	0
15	163	5

Table 6.2: COGENT and C32SAT benchmarks

Chapter 7

Summary

We presented C32SAT and its underlying concepts. We gave a motivation for formal verification and discussed concepts like the SAT problem, AIGs, two level minimisation and transformations in CNF. We discussed C32SAT's input language, semantics, operators, design decisions, architecture, algorithms and related work. Our benchmarks and comparison to COGENT show that C32SAT's functional expression representation is applicable to check satisfiability of boolean C expressions efficiently.

We want to support all C99 operators in future versions of C32SAT. Additionally we want to introduce a new flag which indicates whether a division by zero occurred or not. These extensions should make C32SAT more useful in order to verify code fragments.

C32SAT is available in source. We use the BSD license. You can download C32SAT at the web page [1] of the institute for Formal Models and Verification (FMV).

We strongly believe that formal verification becomes more and more important as software complexity and thus testing costs increases. C32SAT is our approach to deal with increasing software complexity.

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

Appendix A

C32SAT 1.0 Tutorial

This tutorial guides you through your first steps with C32SAT.

A.1 The four main modes

The following main modes are available:

1. Satisfiability mode
2. Tautology mode
3. Defined result mode
4. Undefined result mode

A.1.1 Satisfiability mode

By using the satisfiability mode it can be verified whether a C32SAT formula is satisfiable or not. The question which C32SAT addresses in the satisfiability mode is the following:

> Does an assignment to the variables of a given C32SAT formula exist which leads to a result which is defined and equal to true?

The satisfiability mode is the default main mode. A formal definition of the semantics of the satisfiability mode can be found in section 5.7.1.

Two boolean examples

The satisfiability mode can be used to verify the basic rules of propositional logic. Consider the following semantical equivalence:

$$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$$

This equivalence represents the distributivity of $\wedge$ over $\vee$ and can be found in table C.7. The left and the right side of the formula are semantically equivalent. This means that there does not exist any assignment to the variables of the formula so that the value on the left side is different from the value on the right side. We use C32SAT to verify this equivalence. We represent the formula by a C32SAT formula and negate the whole formula. If we want to verify that the formula is tautological then it is sufficient to show that its logical negation is unsatisfiable[1]. Consider the file distributivity.c32sat:

```
!((x && (y || z)) <=> ((x && y) || (x && z)))
```

We call C32SAT in the following way:

c32sat -s distributivity.c32sat[2]

C32SAT yields:

FORMULA IS UNSATISFIABLE

Thus, we can conclude that the original distributivity rule is correct.

As a second example we consider the following formula:

$$(y \vee z) \leftrightarrow ((x \wedge y) \vee (x \wedge z))$$

The question is if the left side and the ride side are semantically equivalent. If the two sides are equivalent then the operator $\leftrightarrow$ could be replaced by $\equiv$. We try to verify that the original formula is tautological by showing that its negation is unsatisfiable. We derive the following C32SAT formula and save it to the file equiv.c32sat:

```
!((y || z) <=> ((x && y) || (x && z)))
```

We call C32SAT in the following way:

c32sat -s equiv.c32sat

C32SAT yields for example:

[1] In C32SAT it is the case that if you use only operators where the result is always defined, then negating an unsatisfiable formula leads to tautological formula and vice versa. If you use operators like the division operator where undefined results can occur, then this relation does not hold anymore. See section 5.7.5 for more details.

[2] -s means satisfiability

FORMULA IS SATISFIABLE
ASSIGNMENT(S):
y = 0
z = 1
x = 0

Thus, a counter example has been found. The two sides are not semantically equivalent.

32 bit examples

The satisfiability mode can also be used to verify formulas which use 32 bit operators like the shift operator. Note that boolean and 32 bit operators can also be mixed. We show this later. Now we demonstrate how C32SAT can be used to solve equation systems. Consider the following equation system:

$$x + y = 62$$
$$x - 6 = 4 * (y - 6)$$

The domain of of the variables x and y are 32 bit integers. C32SAT can be used to answer the question whether there exists a solution of the equation system in the 32 bit domain or not. We derive the following C32SAT formula and save it to the file equation.c32sat:

```
(x + y == 62)
&&
(x - 6 == 4 * (y - 6))
```

We call C32SAT in the following way:

c32sat -s equation.c32sat

C32SAT yields:

FORMULA IS SATISFIABLE
ASSIGNMENT(S):
x = 46
y = 16

Thus, we have found a solution for the equation system.

As a second example we show how the satisfiability mode can be used to verify that the following equivalence holds in the 32 bit domain. Let BXOR(x, y) be

the bitwise XOR-operation on x and y and BCOMP(x) the bitwise complement of x:

$$BXOR(x, y) = BXOR(BCOMP(x), BCOMP(y))$$

We verify that this formula is tautological by showing that its negation is unsatisfiable. Consider the C32SAT formula in the file xorrel.c32sat:

```
!((x ^ y) == (~x ^ ~y))
```

We call C32SAT in the following way:

c32sat -s xorrel.c32sat

C32SAT yields:

FORMULA IS UNSATISFIABLE

Thus, we have verified that the original equivalence holds.

Special case

Consider the following formula:

$$x/0$$

This formula is unsatisfiable, because the result of this formula is always undefined.

A.1.2 Tautology mode

The tautology mode can be used to verify whether a given C32SAT formula is tautological or not. The question which C32SAT addresses in the tautology mode is the following:

> Does every assignment to the variables of a given C32SAT formula lead to a result which is defined and equal to true?

A formal definition of the semantics of the tautology mode can be found in section 5.7.2.

Boolean examples

Consider the following semantical equivalence:

$$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$$

As already mentioned before this equivalence represents the distributivity of $\wedge$ over $\vee$ and can be found in table C.7. The tautology mode can be used to show directly that this equivalence holds. We derive the following C32SAT formula and save it to the file distributivity.c32sat:

```
(x && (y || z)) <=> ((x && y) || (x && z))
```

We call C32SAT in the following way:

c32sat -t distributivity.c32sat[3]

C32SAT yields:

FORMULA IS TAUTOLOGICAL

Thus, we have shown directly that the two sides of the formula are semantically equivalent.

As another example consider the following formula:

$$(y \vee z) \leftrightarrow ((x \wedge y) \vee (x \wedge z))$$

The question is if the left side and the ride side are semantically equivalent. If the two sides are equivalent then the operator $\leftrightarrow$ could be replaced by $\equiv$. We derive the following C32SAT formula and save it to the file equiv.c32sat:

```
(y || z) <=> ((x && y) || (x && z))
```

We call C32SAT in the following way:

c32sat -t equiv.c32sat

C32SAT yields for example:

FORMULA IS NOT TAUTOLOGICAL
COUNTER-EXAMPLE:
x = 0
y = 1
z = 0

Thus, we have found a counterexample. The two sides are not semantically equivalent.

[3] -t means tautological

32 bit examples

The tautology mode can be used to show that the bitwise xor (BXOR) can be used to set all bits of an integer variable to zero:

$$BXOR(x, x) = 0$$

The question is if this formula holds for every 32 bit integer assignment. We derive the following C32SAT formula and save it to the file xor.c32sat:

```
(x ^ x) == 0
```

We call C32SAT in the following way:

c32sat -t xor.c32sat

C32SAT yields:

FORMULA IS TAUTOLOGICAL

Thus, we have verified that the formula holds for every 32 bit integer.

Another example is the following formula:

$$BXOR(x, y) = BAND((BOR(x, y), BCOMP(BAND(x, y)))$$

This formula expresses that the bitwise xor (BXOR) can be represented by the bitwise and (BAND), the bitwise or (BOR) and the bitwise complement (BCOMP). The question is if this equivalence holds for every 32 bit assignment to the variables x and y. We derive the following C32SAT formula and save it to the file xorsub.c32sat:

```
(x ^ y) == ((x | y) & ~(x & y))
```

We call C32SAT in the following way:

c32sat -t xorsub.c32sat

C32SAT yields:

FORMULA IS TAUTOLOGICAL

Thus, we have verified that this formula holds for every 32 bit assignment to the variables x and y.

Another formula can derived from the previous two formulas. Consider the following formula:

$$(x = y) \rightarrow (BAND((BOR(x, y), BCOMP(BAND(x, y))) = 0)$$

If we know that x is equal to y then we know that $BXOR(x, y) = 0$, because y can be substituted by x and we have shown $BXOR(x, x) = 0$ before. We want to verify that this is always the case. We encode this problem into the following C32SAT formula and save it to the file xorsub2.c32sat:

```
(x == y) => (((x | y) & ~(x & y)) == 0)
```

Note that this formula is a mixed formula which contains boolean and 32 bit operators. We call C32SAT in the following way:

c32sat -t xorsub2.c32sat

C32SAT yields:

FORMULA IS TAUTOLOGICAL

Thus, we know that our intention was right. This formula holds for every 32 bit assignment to the variables x and y.

Special case

A user could think of using the tautology mode to show that the + operator is commutative in the 32 bit domain. The user derives the following C32SAT formula and saves it to the file commutative.c32sat:

```
x + y == y + x
```

The user calls C32SAT in the following way:

c32sat -t commutative.c32sat

C32SAT yields for example:

FORMULA IS NOT TAUTOLOGICAL
COUNTER-EXAMPLE:
x = 336875542
y = 1810608106

At first sight this result seems strange. Nevertheless, the result is correct according to the semantics of C32SAT's tautology mode. It is not the case that every assignment to the variables x and y leads to a result which is defined and true. For example if we substitute x by 336875542 and y by 1810608106, then we obtain an integer overflow. The reason is that the result cannot be represented by a signed 32 bit integer variable. C32SAT treats the result of integer overflows as undefined values by default.

C32SAT can be configured so that it does not treat the result of integer overflows as undefined values. If we call C32SAT with the option *-allow-overflow*, then C32SAT treats integer overflows as defined twos complement overflows. Now we call C32SAT with the additional option *-allow-overflow*:

c32sat -t -allow-overflow commutative.c32sat

C32SAT yields:

FORMULA IS TAUTOLOGICAL

This is exactly what we wanted to verify. We wanted to verify that the + operator is commutative in the 32 bit domain.

A.1.3 Defined result mode

The defined result mode can be used to verify compliance to the C99 standard. The question which C32SAT addresses in the defined result mode is the following:

> Does every assignment to the variables of a given C32SAT formula lead to a defined result according to the C99 standard?

A formal definition of the semantics of the defined result mode can be found in section 5.7.3.

Boolean examples

Due to the fact that boolean operators cannot introduce undefined values, boolean examples always lead to defined results. Thus, the defined result mode does not make sense for boolean examples.

32 bit examples

Consider the following code fragment:

...

```
int x, y, z;
/* assign an arbitrary value x0 to x */
/* assign an arbitrary value y0 to y */
...
if (x >= 0 && x <= 100 && y >= 0 && y < 32){
  z = x >> y;
} else {
  z = x | y;
}
...
```

The question is if the value of z is always defined after executing this code fragment. The defined result mode can be used to verify this property. We derive the following C32SAT formula and save it to the file def1.c32sat:

```
((x >= 0) && (x <= 100) && (y >= 0) && (y < 32))
?
(x >> y) : (x | y)
```

We call C32SAT in the following way:

c32sat -ad def1.c32sat[4]

C32SAT yields:

THE RESULT OF THE FORMULA IS ALWAYS DEFINED (C99)

Thus, we have verified that the value which is assigned to z in the original code fragment is always defined after executing the fragment. Hence, this code fragment can smoothly be ported to an other platform.

As another example we consider the following code fragment:

```
...
int x, y, z;
/* assign an arbitrary value x0 to x */
/* assign an arbitrary value y0 to y */
...
if (x >= 0 && x <= 100 && y < 32){
  z = x >> y;
} else {
  z = x | y;
```

[4]-ad means always defined

```
}
...
```

It is nearly the same code fragment as before. Only the If-statement has been altered. The question is if the value of z is always defined after executing this code fragment. We derive the following C32SAT formula and save it to the file def2.c32sat:

```
((x >= 0) && (x <= 100) && (y < 32))
?
(x >> y) : (x | y)
```

We call C32SAT in the following way:

c32sat -ad def2.c32sat

C32SAT yields for example:

THE RESULT OF THE FORMULA IS NOT ALWAYS DEFINED (C99)
COUNTER-EXAMPLE:
x = 33
y = -1084108303

Now we see that the value of z can be undefined after executing this code fragment. The reason is that if x is between zero and one hundred and y is negative, then we enter the if-case. In this case we shift x to the right by a negative number. The result of a shift operation where the second operand is negative is undefined according to the C99 standard and thus unpredictable. Hence, if it cannot be assured that y never contains a negative value during this fragment, then porting it to an other platform is dangerous.

Special case

In the current version of C32SAT the result of a division by zero is treated as an undefined value. This treatment has an unpleasant side effect. Consider the following C32SAT formula in the file division.c32sat:

```
(x / 0) && 0
```

We call C32SAT in the following way:

c32sat -ad division.c32sat

C32SAT yields:

THE RESULT OF THE FORMULA IS ALWAYS DEFINED (C99)

This formula can never be undefined although a division by zero, causing a trap in real programs, occurs. The result of the logical conjunction is $\perp$, because at least one operand is $\perp$. The value of the other operand is irrelevant in this case. This can be dangerous in real programs, because a division by zero causes a trap. In future versions we want to introduce a flag which indicates whether a division by zero occurred or not. We discuss this future work in 5.6.

A.1.4 Undefined result mode

The undefined result mode can be used to detect code fragments which are always undefined and thus dangerous. The question which C32SAT addresses in the undefined result mode is the following:

> Does every assignment to the variables of a given C32SAT formula lead to an undefined result according to the C99 standard?

A formal definition of the semantics of the undefined result mode can be found in section 5.7.4.

Boolean examples

As it is the case in the defined result mode, boolean operators cannot introduce undefined values. Thus, the undefined result mode does not make sense for boolean examples.

32 bit examples

Consider the following code fragment:

```
...
int x, y, z;
...
if (((x - y) & (x + y) * (-3 >> x)) == 0){
...
}
...
```

Consider the following situation: By using the defined result mode we have found out that the expression in the if-statement is not always defined. Now we want to find out if every assignment leads to an undefined result. We derive the following C32SAT formula and save it to the file if1.c32sat:

```
((x - y) & (x + y) * (-3 >> x)) == 0
```

We call C32SAT in the following way:

c32sat -au if1.c32sat[5]

C32SAT yields:

THE RESULT OF THE FORMULA IS ALWAYS UNDEFINED (C99)

Now we know that every assignment leads to an undefined result. The subformulas $(x - y)$ and $(x + y)$ do not always lead to an undefined intermediate result. Thus, the subformula $(-3 >> x)$ has to be responsible. We verify this by calling C32SAT with the following C32SAT formula in the file if2.c32sat:

```
-3 >> x
```

We call C32SAT in the following way:

c32sat -au if2.c32sat

C32SAT yields:

THE RESULT OF THE FORMULA IS ALWAYS UNDEFINED (C99)

Our intention was right. This subformula is responsible for making the result of the whole formula undefined. According to the C99 standard shifting a negative integer to the right is implementation-defined behaviour. Implementation defined behaviour is unspecified behaviour where each implementation documents how the choice is made [10]. Nevertheless, the resulting value depends on compiler semantics and is potentially dangerous if such a code fragment is ported to an other platform. In C32SAT the term undefined is used when the resulting value of an operation is not fully defined and depends on compiler semantics.

[5] -au means always undefined

A.2 Options

We discuss C32SAT's command line options in this section.

A.2.1 Help

The command line option *-h* is used to print a short command line option summary.

A.2.2 Verbose

The command line option *-v* is used for verbose output. C32SAT prints additional information like the number of conjunctions in an AIG.

A.2.3 Pretty print

The command line option *-p* is used to parse and pretty print the input.

Consider the following C32SAT formula:

```
(a || b) && (c || d) && (e || !f)
```

Calling C32SAT with the command line option *-p* results in the following formula:

```
(a || b)
&&
(c || d)
&&
(e || !f)
```

The pretty printer is optimised for formulas which are combined by logical operators. This way of printing should make them more readable.

A.2.4 Bit width

The integer bit width can be configured by using the command line options *-8bits*, *-16bits*, *-32bits* and *-64bits*. If no width is specified then C32SAT assumes 32 bits.

A.2.5 Dump CNF

The command line option *-dump-cnf* is used to dump the generated CNF to standard output. C32SAT uses the DIMACS file format for dumping. This format is discussed in section 2.6.1.

Example

Consider the following C32SAT formula:

```
a && b
```

Calling C32SAT with the command line option *-dump-cnf* results in the following output:

```
p cnf 3 4
3 0
-3 1 0
-3 2 0
-1 -2 3 0
```

A.3 Overflow treatment

C32SAT can be configured to treat overflows as undefined results or as two's complement overflows by using the command line options *-disallow-overflow* and *-allow-overflow*. The behaviour on signed integer overflow is undefined according to the C99 standard. Two's complement cannot be guaranteed. Thus, the option *-disallow-overflow* is the default option. Nevertheless, allowing integer overflows makes it possible to verify real hardware which uses two's complement representation. This is one main reason why the treatment of integer overflows can be configured in C32SAT.

Appendix B

Installation

This section describes briefly how C32SAT can be installed on your system. C32SAT is available on the following platforms:

- Linux
- Microsoft Windows
- Mac OS

B.1 Required software

The installation process of C32SAT requires the following software:

- Gnu Compiler Collection (GCC)
- Gnu Make
- Shell (e.g. BASH)

If C32SAT is installed on a Windows Operating System, then the usage of CYGWIN[1] or MINGW[2] is recommended.

B.2 Unpacking

The C32SAT archive has to be unpacked first. This can be done by using the Unix program *tar*:

tar xfvz c32sat*.tar.gz

[1] http://www.cygwin.com
[2] http://www.mingw.org

B.3 Compiling

The following command which has to be executed in the C32SAT root directory starts the build process:

make project

Optional:
The following command lists all available make options:

make help

B.4 Test cases

This step is optional. C32SAT has its own test suite. The following command runs this suite:

make run_tests

All test cases should succeed. If not all test cases succeed, then C32SAT will not work correctly on your system.

Appendix C

Propositional logic

C.1 Propositional logic operators

Table C.1 shows all propositional logic operators.

C.1.1 The semantics of the propositional logic operators

The semantics of the propositional logic operators are defined by the tables C.2, C.3, C.4, C.5 and C.6.

C.2 Rules of propositional logic

Table C.7 shows the basic rules of propositional logic. Table C.8 shows the De Morgan rules. More information can be found in [11].

Name	Symbol
Negation	$\neg$
Conjunction	$\wedge$
Disjunction	$\vee$
Implication	$\rightarrow$
Equivalence	$\leftrightarrow$

Table C.1: The operators of propositional logic

x	$\neg x$
$\bot$	$\top$
$\top$	$\bot$

Table C.2: Truth table of $\neg$

x	y	$x \wedge y$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Table C.3: Truth table of $\wedge$

x	y	$x \vee y$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$

Table C.4: Truth table of $\vee$

x	y	$x \rightarrow y$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Table C.5: Truth table of $\rightarrow$

x	y	$x \leftrightarrow y$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Table C.6: Truth table of $\leftrightarrow$

Number	Rule	Name
1	$x \wedge \bot \equiv \bot$	Neutrality of $\wedge$
2	$x \vee \bot \equiv x$	Neutrality of $\vee$
3	$x \wedge \top \equiv x$	Identity of $\wedge$
4	$x \vee \top \equiv \top$	Identity of $\vee$
5	$x \wedge x \equiv x$	Idempotency of $\wedge$
6	$x \vee x \equiv x$	Idempotency of $\vee$
7	$x \wedge \neg x \equiv \bot$	Contradiction
8	$x \vee \neg x \equiv \top$	Tautology
9	$x \wedge y \equiv y \wedge x$	Commutativity of $\wedge$
10	$x \vee y \equiv y \vee x$	Commutativity of $\vee$
11	$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$	Distributivity of $\wedge$ over $\vee$
12	$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$	Distributivity of $\vee$ over $\wedge$
13	$x \wedge (x \vee y) \equiv x$	Absorption rule of $\wedge$
14	$x \vee (x \wedge y) \equiv x$	Absorption rule of $\vee$
15	$\neg\neg x \equiv x$	Double negation

Table C.7: Basic rules of propositional logic

Number	Rule
1	$\neg(x \wedge y) \equiv \neg x \vee \neg y$
2	$\neg(x \vee y) \equiv \neg x \wedge \neg y$

Table C.8: De Morgan rules

Bibliography

[1] Armin Biere. The evolution from LIMMAT to NANOSAT. Technical Report 444, Dept. Computer Science, ETH Zürich, 2004.

[2] M. Buro and H. Kleine Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, 1993.

[3] Byron Cook, Daniel Kroening, and Natasha Sharygina. Accurate theorem proving for program verification. Technical Report 464, ETH Zürich.

[4] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of ACM STOC'71*, pages 151–158, 1971.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] DIMACS. Satisfiability suggested format, 1993.

[7] John L. Hennessy and David A. Patterson. *Computer Archtitecture - A Quantitative Approach - Second Edition.* Morgan Kaufmann, 1996.

[8] Gerard J. Holzmann. *The SPIN Model Checker.* Addison Wesley, 2004.

[9] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. pages 283–292.

[10] ISO/IEC. Programming languages - C (ISO/IEC 9899:1999 (E)), 1999.

[11] John Kelly. *The Essence of Logic 1st Edition.* Prentice Hall, 1996.

[12] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 2001.

[13] G S Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1968.

[14] Armin Biere und Alessandro Cimatti und Edmund M. Clarke und M. Fujita und Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[15] Per Bjesse und Arne Borälv. DAG-aware circuit compression for formal verification. ICCAD'04, 2004.

[16] Brian Kernighan und Dennis Ritchie. *The C Programming Language - Second Edition.* Prentice Hall, 1986.

[17] Martin Davis und George Logemann und Donald Loveland. A machine program for theorem proving. In *Communications of the ACM*, volume 5, 1962.

[18] Martin Davis und Hilary Putman. A computing procedure for quantification theory. In *Journal of the ACM (JACM)*, volume 7, 1960.

[19] B. Berard und M. Bidoit und A. Finkel und F. Laroussinie und A. Petit und L. Petrucci und Ph. Schnoebelen und P. McKenzie. *Systems and Software Verification - Model-Checking Techniques and Tools.* Springer, 2001.

[20] Michael Huth und Mark Rya. *Logic in Computer Science - Modelling and Reasoning about Systems.* Cambridge University Press, 2004.

[21] Edmund M. Clarke Jr. und Orna Grumberg und Doron A. Peled. *Model Checking.* The MIT Press, 1999.

[22] John E. Hopcroft und Rajeev Motwani und Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, 2002.

[23] Alfred V. Aho und Ravi Sethi und Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* Prentice Hall, 1986.

[24] Holger H. Hoos und Thomas Stützle. *Stochastic Local Search - Foundation And Applications.* Morgan Kaufmann, 2005.

List of Figures

List of Tables

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Magisterarbeit 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.

Linz, Februar 2006

Robert Daniel Brummayer Bakk. techn.