AUTOMATIC TEST DATA GENERATION FOR CONTROL FLOW COVERAGE USING BOOLEAN CONSTRAINTS

Approved: _____ Date: Nov. 12, 2013
Committee Chair

Approved: _____ Date: Nov. 12, 2013
Committee Member

Approved: _____ Date: 11/12/2013
Committee Member

Suggested content descriptor keywords:

Automated Test Data Generation, Test Coverage, Control Flow Coverage, Boolean Satisfiability, Constraints, Modeling

AUTOMATIC TEST DATA GENERATION FOR CONTROL FLOW COVERAGE USING BOOLEAN CONSTRAINTS

A Thesis

Presented to

The Graduate Faculty

University of Wisconsin-Platteville

In Partial Fulfillment

Of the Requirement for the Degree

Master of Science

Joint International Master's Program in Computer Science

By

Stefan C. Weiler

2013

AUTOMATIC TEST DATA GENERATION FOR CONTROL FLOW COVERAGE

USING BOOLEAN CONSTRAINTS

Stefan C. Weiler

Under the Supervision of Prof. Dr. Alexander del Pino,

Prof. Dr. Ronald Moore, and Prof. Dr. Yan Shi

Statement of the Problem

To prevent human error in software development to cause failures, thorough testing is necessary. The importance of this task is clear to see in examples like the failed maiden flight of Ariane 5.

Coverage criteria are metrics used to measure the *grade of coverage* of software. *Control flow coverage* comprises sets of such coverage criteria, based on the control flow of a program. The control flow is defined by control structures in a program's source code (e.g. explicit *if*-statements).

To obtain test cases which fulfill a desired grade of coverage, automated generators of test data are needed. A new approach based on *Boolean satisfiability* (SAT) is proposed for this. SAT is a technology based an Boolean formulas and logic reasoning. Using SAT is generally recognized as a very powerful approach to many real world problems.

Methods and Procedures

At the beginning, the fundamentals are given. This comprises a discussion of control flow coverage, as well as constraints and SAT solving technology.

The approach is then first discussed theoretically. The problem of automatically generating test data to achieve control flow coverage is modeled with constraints. This problem is split into two parts: modeling the basic operations relevant to the program's control

flow, and based on this modeling a coverage type.

To show the applicability of the proposed approach, the focus of this thesis is the development of a prototype application. The technique can potentially be applied to many procedural or object oriented programming languages. Exemplarily it has been implemented for functions in a subset of C

To show the capabilities of the implemented solution, examples of its use are given. The implementation is used to generate test cases for a number of examples taken from literature or practice (including one example addressing the Ariane 5 incident). The results of this test case generation get evaluated.

Finally, an evaluation and outlook are given.

Summary of Results

An approach to automatically generate test data for control flow coverage using Boolean constraints has been elaborated.

It has been shown, that – based on a program's source code – a program's control flow, and the problem of deriving test cases covering this control flow can be modeled with Boolean constraints. An *encoder* application has been developed to fulfill this task. Further it has been shown, that the derived constraint system can be solved by one of the existing, highly efficient SAT solvers. A *decoder* has been developed to derive test data based on encoder- and solver-output. A workflow using encoder, solver, and decoder has been created, which can be used to derive test data itterativly.

On several examples from literature and practice it has been tested and documented that the approach can be applied to real world problems. The derived sets of test data for these examples have been proven to fulfill the desired types of control flow coverage.

An outlook has been given on the possible next steps of future work, which could help to move on from the developed prototype towards an application to be used in practice.

Table of Contents

List of Abbreviations

cc Condition coverage (as command line argument)

CC Creative Commons

cdc Condition/decision coverage (as command line argument)

CLP(FD) Constraint Logic Programming (Final Domain)

CNF Conjunctive normal form

dc Decision coverage (as command line argument)

DIMACS Center for Discrete Mathematics and Theoretical Computer Science

ESA European Space Agency

GFDL GNU Free Documentation License

GNU "GNU's Not Unix!" (recursive acronym)

h_da Hochschule Darmstadt (Darmstadt University of Applied Sciences)

LPB Linear pseudo-Boolean

mcdc Modified condition/decision coverage (as command line argument)

MC/DC Modified condition/decision coverage

NASA National Aeronautics and Space Administration

Non-CNF Non conjunctive normal form

Non-LPB Non-linear pseudo-Boolean

PB Pseudo-Boolean

RDoc RubyDoc (Document Generator for Ruby Source)

SA Share-Alike

SAT Boolean satisfiability

SRI Système de Référence Inertielle (Inertial Reference System)

UTC Coordinated Universal Time

YAML YAML Ain't Markup Language / Yet Another Markup Language

List of Figures

List of Tables

I. Introduction

Critical software is responsible for the prevention of damage to persons and property. It is present in various fields, including software in vehicles (land, sea, and aerospace), software involved in handling or monitoring dangerous nuclear, biological, and chemical materials, and in medical devices[1] [WDR09, Lev04].

A failure in such a critical software can potentially lead to huge monetary losses, high numbers of injured, and deaths.

1.1. Excursus

One well known example of a failure is the first flight of Ariane 5, a launcher of the European Space Agency (ESA). This example can especially well be used for research, as ESA shared insight into their findings with the general public However, a report with greater technical detail has been authorized for restricted circulation only [LLF+96, p. iii].

On 4 June 1996, 12:34:06 UTC, Centre Spatial Guyanais, Kourou, French Guiana, ESA launched their Ariane 5 on its maiden flight. Only about 40 seconds later a huge explosion could be seen, debris and potentially toxic remains of rocket fuel were falling down from 3700 meters above ground. The launcher had abruptly left course, and the self-destruction had been triggered automatically. Due to the self-destruction above the cleared area, the launcher was prevented to potentially head towards settled areas, nobody was injured [ESA96].

[1] This is not intended to be a complete lest

An Inquiry Board was given the task to determine the cause of the failure and recommend actions to prevent similar events in future.

> "The origin of the failure was [...] rapidly narrowed down to the flight control system and more particularly to the [active and backup] Inertial Reference Systems [(SRIs)], which [...] ceased to function almost simultaneously [...]" [LLF+96, p. 1].

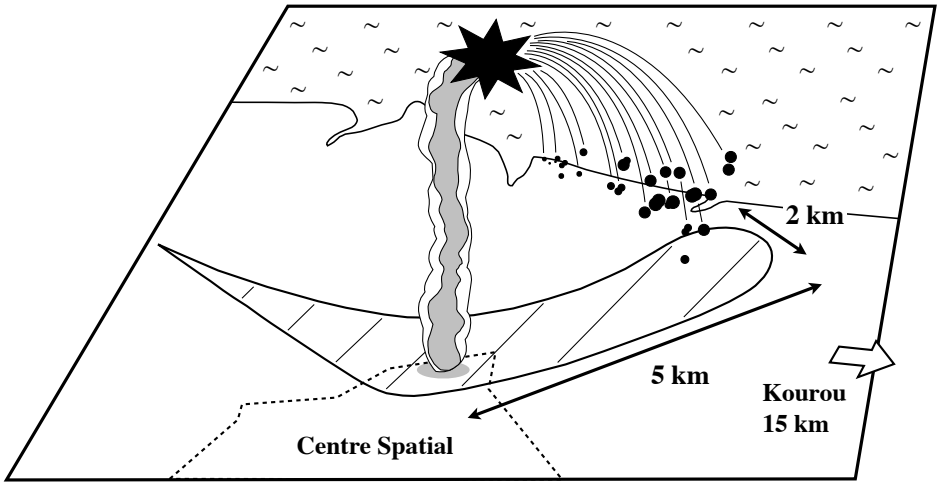


Figure 1.1.: Fragment fallout zone of failed Ariane 501 launch [Phr09b].

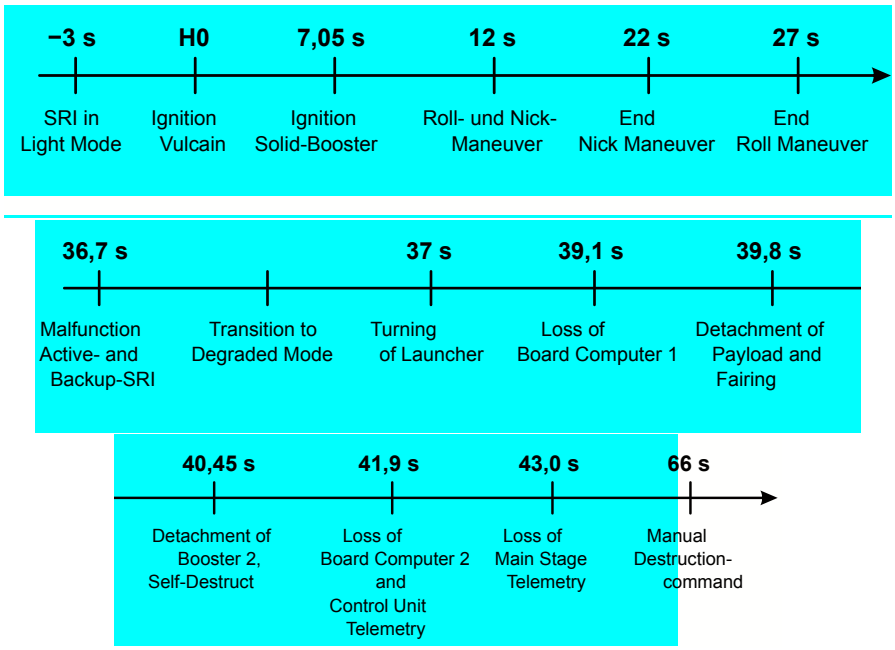


Figure 1.2.: Chronology of Ariane-501 flight [Phr09a].

The actual fault was discovered to be an unprotected conversion of a 64-bit floating point to a 16-bit signed integer value. The software containing the fault had been reused from the predecessor Ariane 4, where it never caused a failure. The much higher acceleration of Ariane 5 was the reason for the value to exceed the range represented by a 16-bit signed integer. An unexpected Operand Error occurred, the active SRI shut down, the backup SRI took over, but experienced the same failure [LLF+96, pp. 4, 12].

The actual code in Ada programming language can be seen below [Lev10]. In Ada this statement can raise an exception, when the number to be converted is out of range of the target type. Appropriate exception handling did not exist.

```
450 P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
451                           ((1.0/C_M_LSB_BH) *
452                           G_M_INFO_DERIVE(T_ALG.E_BH)));
```

Along with the specific fault found in code, the investigation revealed a number of weaknesses in the overall development process. These weaknesses were allowing human error to cause faults which stay undetected.

During the development the persons responsible had been overly confident.

> "[...] the view had been taken that software should be considered correct until it is shown to be at fault. The Board has reason to believe this view is also accepted in other areas of Ariane 5 software design. The Board is in favor of the opposite view, that software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct.
>
> This means that critical software – in the sense that failure of the software puts the mission at risk – must be identified at a very detailed level [...]" [LLF+96, p. 6].

Subsequently the Board found the test coverage to be insufficient.

> "[...] it was not realized, that the test coverage was inadequate to expose such limitations [in the Inertial Reference System]." [LLF+96, p. 9].

A test with trajectory data input that would have triggered a failure had obviously not been performed.

> "The main explanation for the absence of this test [...] [was that] the SRI specification (which is supposed to be a requirements document for the SRI) does not contain the Ariane 5 trajectory data as a functional requirement" [LLF+96, p. 7]

The Board found this to be a general issue.

> "The supplier of the SRI was only following the specification given to it" [LLF+96, p. 6]

This indicates the lack of test coverage criteria beyond requirements coverage or at least the neglect of their importance.

Based on its analyses and conclusions the Boards issued a list of 14 recommendations, two of them addressing coverage:

> "R2 [...] A high test coverage has to be obtained" [LLF+96, p. 13]

> "R11 Review the test coverage of existing equipment and extend it [...]" [LLF+96, p. 14]

Obviously ESA learned the lessons from initial problems. With an overall success rate of about 96%[2] Ariane 5 is the most reliable available launcher in its class[3].

1.2. Goals

To prevent human error in software development to cause failures in critical software, thorough testing is necessary in software verification. *Coverage criteria* are metrics used to measure the *grade of coverage* of software. Depending on the criticality of a software,

[2] 2 failures and 2 partial failures in 70 launches

[3] Heavy lift launch vehicles (20,000 kg and above to low earth orbit): Russian Proton-M 90%, US Delta IV Heavy 92% (see Appendix A)

defined coverage criteria should be fulfilled to a defined grade. The grade of coverage is the quotient of the number of instances where a criterion has been fulfilled divided by the total number of instances the criterion applies to:

$$grade\_of\_fulfillment = \frac{instances_{fullfilled}}{instances_{total}}$$

For critical software it is often stated, that coverage has to be *achieved*, which means its criteria have to be fulfilled to the grade of 100%. Corresponding regulations exist to specify minimum standards on this, for example in aeronautics.

Control flow coverage is an umbrella term for sets of such coverage criteria. A single set defines a *type* of control flow coverage. One or more criteria have to be applied for a defined type of control flow coverage. These types of control flow coverage will be discussed in greater detail later on. As the name indicates, the coverage criteria used are based on the control flow, i.e. the structure of a program (For example: Which decisions are made?, Which statements get executed?). To retrieve the actual structure of a program, its source code has to be analyzed. Tests performed on this basis are therefore white box tests.

Finding test cases to achieve a type of control flow coverage is a complex real world problem, which has to be addressed in some way.

Boolean satisfiability (SAT) solving is a technology based on boolean logic. It can be used on a broad variety of real world problems, which have to be encoded in boolean formulas.

This thesis aims to approach the problem of achieving certain types of control flow coverage with a new method, using SAT solving techniques. It will provide a tool for automatic test data generation. Further on, this work can also be used for the evaluation of existing test cases derived by other means (for example: requirement based test cases) with respect to their coverage of the control flow. With a second metric it is hence providing a more thorough test of the software as requirements coverage alone.

It will therefore be shown that this problem can be expressed in Boolean formulas. SAT solving will be used to find a valid assignment to these formulas. From this assignment a set of test cases can be derived.

Examples will be used to show the applicability of this approach. With the successful implementation of a prototype, this thesis will show a promising approach for the automatic generation of test data.

1.3. Background

This thesis is about the task of automatic test data generation in the field of software verification. It is one of many fields in which Boolean constraints and SAT solving can be applied. In white box testing, sets of test data are needed that allow to test for certain types of control flow coverage. For the automatic test data generation this thesis will focus on *control flow coverage / structural coverage*. The control flow of a program is driven by control structures, including: conditional structures (if/else), iteration structures (loops), selective structures (switch), jump statements, and – as in the Ariane 5 example – exception handling[4]. The selected types of coverage are shown in the following list:

- decision coverage
- condition coverage
- condition/decision coverage
- modified condition/decision coverage (MC/DC)

Other types of coverage are out of scope and will not be discussed in this thesis.

Automatic test data generation, in this context, means to identify input values for a (minimal) set of test cases, leading to the desired control flow coverage of a program when executed.

Finding test data is an important task in software verification. It is important to have generators, which can efficiently provide correct data.

[4]The set of available control structures is depending on the programming language.

Being continuously improved, SAT solving provides state of the art capability to solve problems, which can be encoded as a set of constraints. By expressing the problem of finding the test cases for the selected types of control flow coverage as a constraint system, which has to be solved, this approach makes use of the continuing advances in constraint solving techniques.

1.4. Related Work

Research on *Automatic Test Data Generation using Constraint Solving Techniques* has previously been done by Gotlieb, Botella and Rueher [GBR98]. They developed a system that can derive a single test case covering one branch of a program's structure that has to be specified by selecting a point in the source code. This thesis aims to move further and provide a system giving a set of test cases that will allow to fulfill a selected type of structural coverage.

Moreover, Gotlieb et al. used CLP(FD)[5] provided by a library of Sicstus Prolog [GBR98, p. 60], rather than SAT solving. This library has the disadvantage that a license is required and has to be bought for its use, while SAT solvers are generally available for free and open source.

Another general source will be the collection *Handbook of Satisfiability* [BHvMW09], especially the chapter *Software verification* [Kro09].

The guide *A Practical Tutorial on Modified Condition/Decision Coverage*, also containing detailed information on the other types of coverages in scope of this thesis, has been published by NASA [HVCR01].

An example for the required in- and output format of a SAT solver is given in the manual of the Mini SAT[6] solver [Whe08]. Similar descriptions are given for other solvers too, but they might differ slightly.

[5] CLP(FD): Constraint Logic Programming (Final Domain)
[6] Mini SAT is given as an example here for having a well documented output format

1.5. Method of Approach

To show the applicability of the proposed approach, the focus of this thesis is the development of a prototype application. This application can automatically generate test data for selected types of control flow coverage using Boolean constraints.

The technique can potentially be applied to many procedural or object oriented programming languages. Exemplarily it has been implemented for functions in a subset of C. The decision for C is based on two factors:

1. Its leading position as the most widespread programming language according to the TIOBE index [Tio13a] allows this work to address a wide audience.
2. Being statically typed and staying close to the hardware circuits makes it easier to model C compared to languages with dynamic typing or higher abstraction.

The extend of the supported subset is defined by a grammar. The architecture of the implementation allows expanding the covered subset. A theoretical examination (the approach) is discussed separately from the actual implementation to ease the reuse of the developed approach for other programming languages.

In general for a real world problem to be solved using SAT, three components are needed:

- An *encoder*, encoding an instance of the real world problem, as an instance of the SAT problem.
- A *SAT solver*, solving the instance of the SAT problem, giving one possible assignment for the SAT variables.
- A *decoder*, deriving a solution for the real world problem from the assignment of the SAT variables and (implicit or explicit) information on the encoding.

The implementation follows this scheme consisting of three major parts:

- A command line application taking a C source file and options as input, parsing them, and generating a corresponding set of constraints. It also has to give informa-

tion on mapping the variables of the C source file to the variables of the constraint system.

- One of the existing SAT solvers being used to solve the generated set of constraints.
- A second command line application taking the output file of the SAT solver and the variable mapping information as input, deriving a set of test cases.

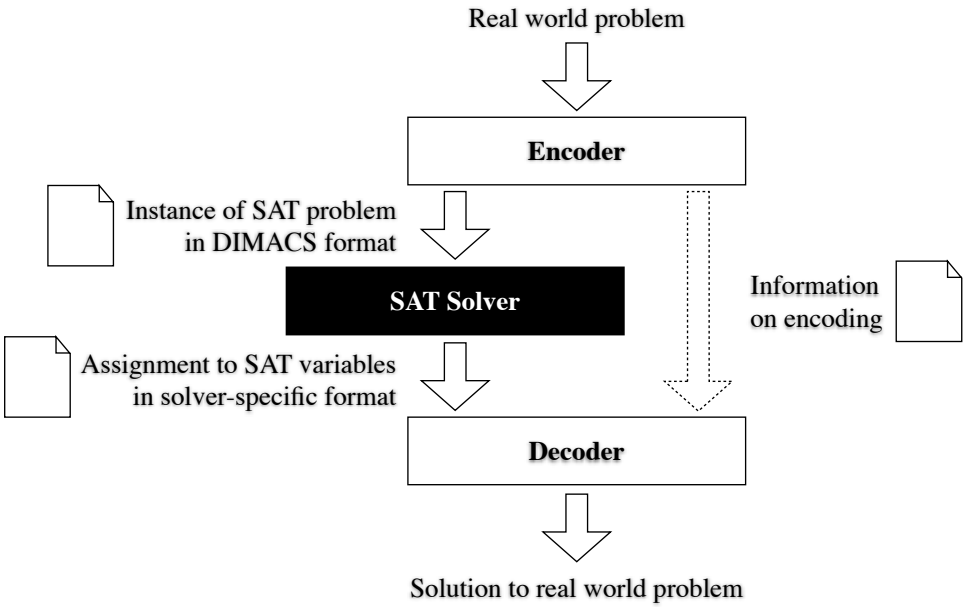


Figure 1.3.: Steps for solving a real world problem with SAT.

The common input format is used to hand over the set of constraints to the SAT solver. "[M]ost SAT solvers [accept] [...] input in a simplified 'DIMACS CNF' format, which is a simple text format" [Whe08]. Output formats are not standardized. The interface can be adapted to a specific SAT solver.

It is possible to chose between decision coverage, condition coverage, condition/decision coverage, and modified condition/decision coverage.

Concerning the order of development, at first parts that are common for all of the coverage types have been implemented. Representations of variable assignments, algebraic and logical operations, and control structures of the source code are represented by equivalent constraints. As condition/decision coverage is based on decision coverage, and condition coverage and condition/decision coverage is itself the basis for modified con-

dition/decision coverage, this gives the order in which the types of control flow coverage were implemented.

To show the capabilities of the implemented solution, examples of its use are given.

In the end the results are evaluated against the goals set in the beginning and a conclusion as well as an outlook is given.

1.6. Motivational Example

A small motivational example showing how test cases can be generated is given below. Further, larger examples, including one related to the Ariane 5 incident, can be found at the end of this thesis in chapter V.

For a better understandability this first example is intentionally kept very small. Unsigned integers, with a reduced number of only four bits, will be used to ease human readability of the intermediate results. The number of bits can for this purpose be changed in the encoders config file. With the setting of only 4 bits for an integer dropping below the usual minimum value, a warning will be given, but execution will continue.

The function "fits_in_one_week" calculates whether a number of required work days for two tasks fits into one week or not, i.e. whether it is theoretically possible to complete both tasks within a single week or not. Two possible outcomes exist: when the sum of days is greater than seven, 0 is returned, otherwise 1 is returned.

This is the functions code as found in the C source file:

```
1  unsigned int fits_in_one_week(unsigned int days_task1, unsigned int
       days_task2){
2      // sum of days
3      unsigned int days;
4      days = days_task1 + days_task2;
5      if(days>7){
6        // sum of days exceeds one week
7          return 0;
8      } else {
9        // sum of days fits in one week
10         return 1;
```

```
11      }
12  }
```

The encoder can then be called, giving it the required input: source file, function identifier, coverage type and number of test cases. The coverage type used is "dc" for *decision coverage*. It will be discussed in greater detail in section 2.1. As of now it should be sufficient to know: it enforces one of the two test cases to take the *if*-branch and the other to take the *else*-branch. The number of test cases has been set to be 2, as it is in this example easy to see this will be the number needed. We do not need to search the optimal number of several iterations.

```
1 ruby coverage-sat-encoder.rb main.c "fits_in_one_week(unsigned int,
      unsigned int)" dc 2
```

A number of intermediate steps is not shown in this example. After tokenization, choosing of the function and reduction to relevant code, the list of tokens is as follows:

```
1 "unsigned", "int", "fits_in_one_week", "(", "unsigned", "int", "
      days_task1", ",", "unsigned", "int", "days_task2", ")", "{", "
      unsigned", "int", "days", ";", "days", "=", "days_task1", "+", "
      days_task2", ";", "if", "(", "days", ">", "7", ")", "{", "}", "else
      ", "{", "}", "}"
```

This tokens will be parsed and an output file will be given:

```
1  c Generated with CoverageSAT Encoder v1 2013
2  c by Stefan Weiler
3  c Darmstadt University of Applied Sciences
4  c  / University of Wisconsin-Platteville
5  c
6  c Input:
7  c Source file:   main.c
8  c Function:      fits_in_one_week(unsignedint,unsignedint)
9  c Coverage type: dc
10 c Test cases:    2
11 c
12 c Mapping:
13 c Parameter mapping:
14 c {days_task1 (4bit unsigned int) -> {1}, days_task2 (4bit unsigned int
       ) -> {9}}
15 c Variable mapping:
16 c {days_task1 (4bit unsigned int) -> {1}, days_task2 (4bit unsigned int
       ) -> {9}, days (4bit unsigned int) -> {17}, 0 (4bit unsigned int)
```

```
    -> {17}, 1 (4bit signed int) -> {83}, 2 (4bit unsigned int) ->
    {93}, 3 (5bit signed int) -> {105}, 4 (5bit signed int) -> {115}, 5
     (5bit signed int) -> {125}}
c
c CNF encoding:
p cnf 230 428
-25 0 -26 0 12 25 -29 0 12 -25 29 0 -12 25 29 0 -12 -25 -29 0 12 -31 0
    -12 25 -31 0 -12 -25 31 0 4 29 -20 0 [...] 230 0
```

This file can be handed to a SAT solver, looking for a solution. The solver used is "PicoSAT". The lines starting with "c" are comments for additional information, which will be ignored by the solver. Most information in these lines is self-explaining. For a detailed discussion on the mapping, see appendix B. Only the line "p cnf 230 428"[7] and the encoded constraints below (the list of constraints is not given here in full length) will be processed. PicoSAT gives this output[8]:

```
s SATISFIABLE
v -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 12 13 -14 -15 -16 -17 -18
v -19 20 21 -22 -23 -24 -25 -26 -27 -28 29 -30 -31 -32 -33 -34
v -35 -36 -37 -38 -39 -40 -41 -42 -43 -44 -45 -46 -47 -48 -49
v -50 -51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -64
v -65 -66 -67 -68 -69 -70 -71 -72 -73 -74 -75 -76 77 -78 -79 -80
v -81 -82 -83 84 85 86 -87 88 89 90 -91 92 -93 94 95 96 -97 98
v 99 100 101 -102 -103 -104 -105 -106 -107 -108 109 -110 111
v -112 -113 -114 -115 -116 117 118 119 -120 -121 122 123 124 125
v 126 -127 128 -129 -130 -131 -132 -133 134 -135 -136 -137 138
v 139 -140 -141 -142 -143 -144 145 -146 -147 -148 149 -150 151
v 152 153 -154 -155 -156 157 -158 -159 -160 161 -162 -163 -164
v 165 166 -167 -168 169 -170 -171 -172 -173 -174 175 -176 -177
v -178 179 -180 181 -182 -183 -184 185 -186 187 -188 -189 -190
v -191 -192 193 -194 195 -196 -197 -198 199 -200 -201 -202 -203
v -204 -205 -206 207 -208 209 -210 211 -212 213 -214 -215 -216
v -217 -218 -219 -220 221 -222 223 -224 225 -226 227 -228 229
v 230 0
```

The first line states that the constraints are satisfiable, i.e. an assignment with two test cases achieving the desired coverage type exists. Below it, the following lines give the assignment of each SAT variable. Assignments for the parameters can be retrieved from this in combination with the parameter mapping. The *decoder* derives the assignments for

[7] To be read as: CNF encoding format, 230 SAT variables, 428 clauses.
[8] line breaks have been modified to better fit in this document

each test case as follows:

```
Satisfiable, parameters for test cases as follows:
Test case 1: days_task1 = 0, days_task2 = 1
Test case 2: days_task1 = 0, days_task2 = 8
```

With these two test cases, it is now possible to run a test on the function. The test covers both possible paths of the function. For test case 1 the *if*-statement evaluates as *true*, for test case 2 it evaluates as *false*. Decision coverage has been achieved.

II. Fundamentals

This chapter introduces the fundamentals required for this thesis. This includes fundamental information on control flow coverage, as well as on constraint systems.

Achieving control flow coverage, more precisely how to achieve a certain type of control flow coverage, is the problem this thesis aims to solve. Constraint systems provide the logic reasoning and solver technology, which are used to find solutions for instances of that problem.

2.1. Control Flow Coverage

Control flow coverage is an umbrella term for sets of coverage criteria. *Coverage criteria* are metrics used to measure the *grade of coverage* of software. The coverage criteria for control flow coverage specifically measure the grade to which the control flow (the structure of a program) is covered when the corresponding tests get executed. The grade of coverage is the quotient of the number of instances where a criterion has been fulfilled divided by the total number of instances the criterion applies to:

$$grade\_of\_fulfillment = \frac{instances_{fullfilled}}{instances_{total}}$$

For critical software, instead of giving a desired grade of coverage, it is often stated, that coverage has to be *achieved*. This means its criteria have to be fulfilled to the grade of 100%.

The control flow of a program is driven by control structures, including: conditional structures (if/else), iteration structures (loops), selective structures (switch), jump statements, and exception handling. The set of available control structures is depending on the programming language.

Documentation on the control flow is – when available – generally too unspecific, sometimes even inconsistent with the source code. The actual structure of a program therefore has to be derived from the source code [SL10, p. 149]. Test cases covering the control flow in a specified way (specified by criteria and grade) can be derived based on this available source code. As these tests are derived from source code, they are *white-box* tests.

A defined coverage criterion or a defined set of coverage criteria is included in a *type* of control flow converage. Types of control flow coverage discussed here are:

- decision coverage
- condition coverage
- condition/decision coverage
- modified condition/decision coverage (MC/DC)

To achieve one of these coverages its criteria have to be fulfilled to 100%. The criteria which have to be fulfilled for each type of coverage can be seen in table 2.1.

To explain the concepts of condition and decision the following example of an *if*-statement shall be used:

```
1 if(a>2 && !b<c)
```

The possible paths taken based on this *if*-statement are shown in figure 2.1.

Coverage Criteria	Decision Coverage	Condition Coverage	Condition/Decision Coverage	Modified Condition/ Decision Coverage
Every decision in the program has taken all possible outcomes at least once	✓		✓	✓
Every condition in a decision in the program has taken all possible outcomes at least once		✓	✓	✓
Every condition in a decision has been shown to independently affect that decision's outcome				✓

Table 2.1.: Comparison of types of control flow coverage [HVCR01, p. 7]

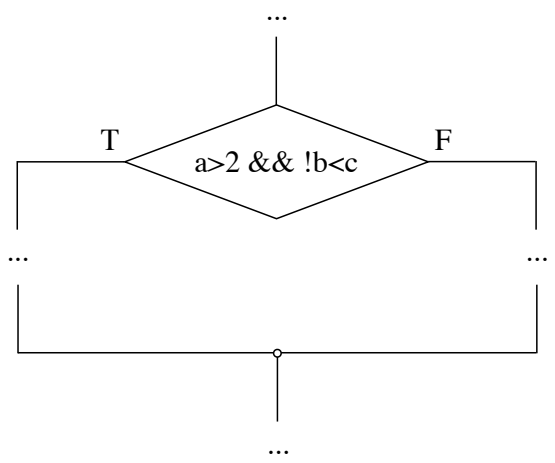


Figure 2.1.: Diagram showing the possible paths based on an example *if*-statement.

Programmers sometimes refer to this whole piece of code as an "*if*-condition". This might cause misunderstandings, as in software test conditions are only specified subparts of this whole statement.

The direct comparison of two variables or constants is a *condition*. For the given example, the conditions are:

- `a>2`,
- `b<c`

The default comparison operators, which might be used in C are <, <=, ==, >=, >, and !=. In other programming languages further ways of comparison may exist, for example "*equals*"-methods in many object oriented languages.

The overall outcome (*true* or *false*) of the *if*-statement is a *decision*. A decision determines (decides) which branch (*if*- or *else*-branch) is being executed. A decision is derived from its contained conditions by combining the conditions with logic operators (In C: *and* (&&), *or* (||), and *not* (!)). In the given example: `([condition1] && ![condition2])`.

Usually, it is not only desired to have just some set of test cases achieving a certain coverage type. It is most efficient to have the optimal (smallest) set of test cases, with the lowest number of test cases.

Obviously, it is hard to derive appropriate test cases manually, even for rather small and simple pieces of code. Working on more complex pieces of code, and deriving not just any, but an optimal set of test cases, is even harder. It should not be approached manually. Computers are used for this in various ways of *automatic / automated test data generation*. The terms *automatic test generation* [ZKVM12, LGR11, YLDM97] and *automated test generation* [Rus05, ZSBE11, BMMS11] are used in the literature interchangeably.

For the example the types of coverages will be shown by the paths of test cases compared side by side as seen in figure 2.2.

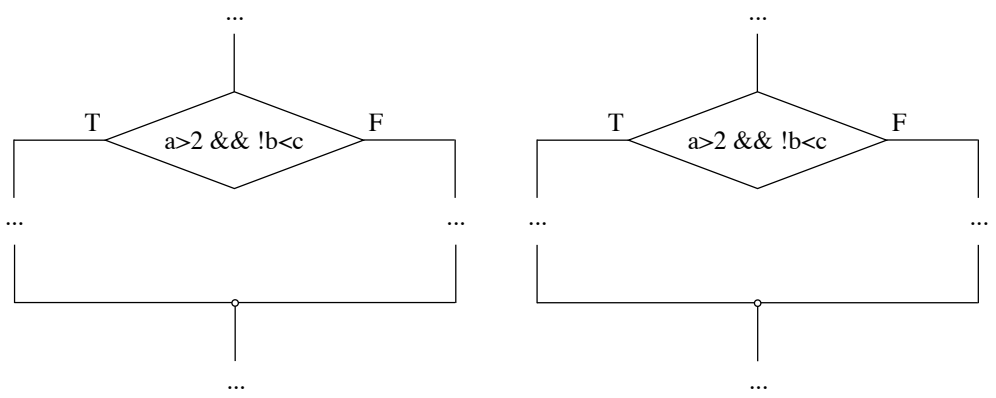


Figure 2.2.: Diagrams showing two test cases parallel

2.1.1. Decision Coverage

With decision coverage both possible outcomes of a decision must be included in the set of test cases. Achieving decision coverage ensures all paths being covered, as seen in figure 2.3. Already the possibility to derive these test cases proves the absence of unreachable paths. Otherwise it indicates their presence.

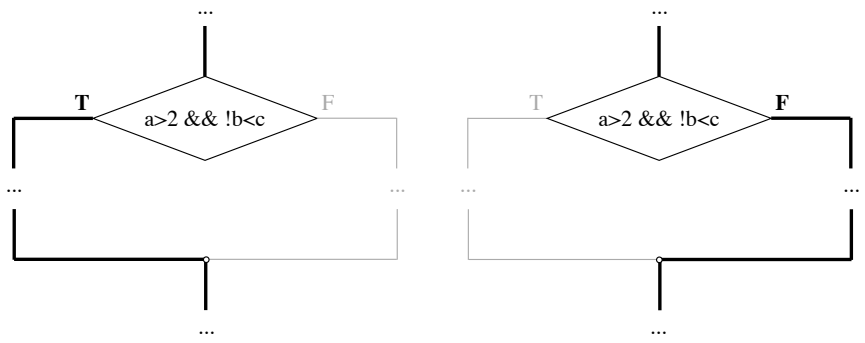


Figure 2.3.: Diagrams showing decision coverage

2.1.2. Condition Coverage

To show condition coverage the decision has to be broken down to its constituting conditions, as seen in figure 2.4. Note that in figure 2.4 & 2.5, the second condition might not be evaluated, due to the diagrams modeling short-circuit evaluation.

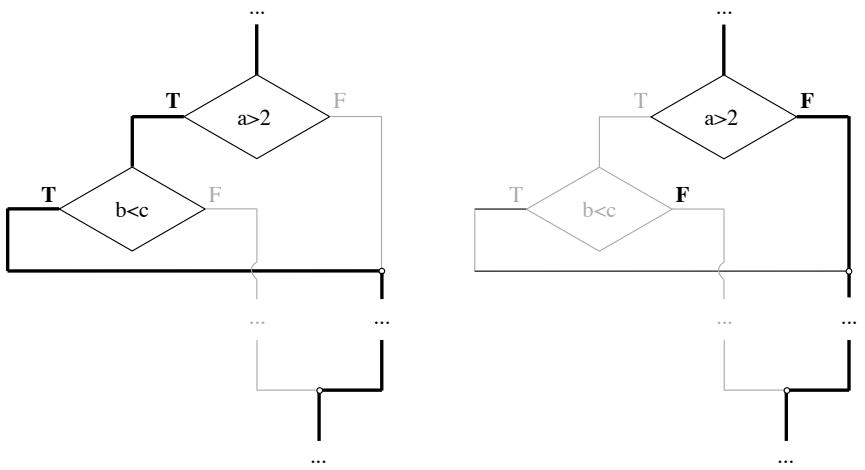


Figure 2.4.: Diagrams showing condition coverage

2.1.3. Condition/Decision Coverage

A set of test cases with full condition coverage does not necessarily cover all branches. This can be the case for example with the following assignment of parameters of two test cases:

- Test case 1 ($a = 3$, $b = 4$, $c = 5$):
  - `condition1` is *true*,
  - `condition2` is *true*;
- Test case 2 ($a = 2$, $b = 1$, $c = 0$):
  - `condition1` is *false*,
  - `condition2` is *false*.

Each condition is once *true* and once *false*, condition coverage is achieved. However, the decision for both test cases is *false*, the code inside the *if*-branch is not covered, decision coverage is not achieved. This can be seen in figure 2.4 This means, that even a set of test cases with full condition coverage might possibly not cover large parts of code. With the combined condition/decision coverage this disadvantage is avoided. One possible set of two test cases is shown in figure 2.5.

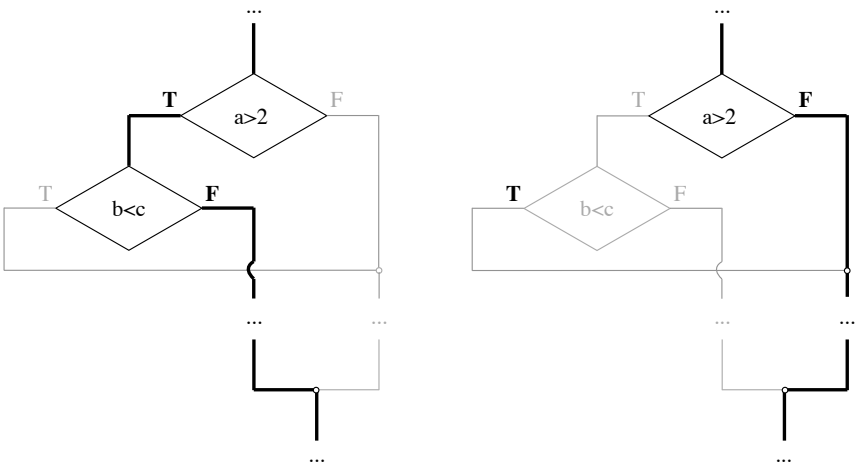


Figure 2.5.: Diagrams showing condition/decision coverage

2.1.4. Modified Condition/Decision Coverage

Modified condition/decision coverage states that, additionally to having condition/decision coverage, each condition must effect its corresponding decision individually. This means, that at least one pair of test cases exists, in which:

1. the condition is once *true* and once *false*,
2. all other conditions are the same in both, and
3. the outcome of the decision is once *true* and once *false*.

The same must be true for all conditions in every decision.

For the given example this can be achieved with three test cases, as shown in figure 2.6. Top and left, respectively top and right show the individual effect.

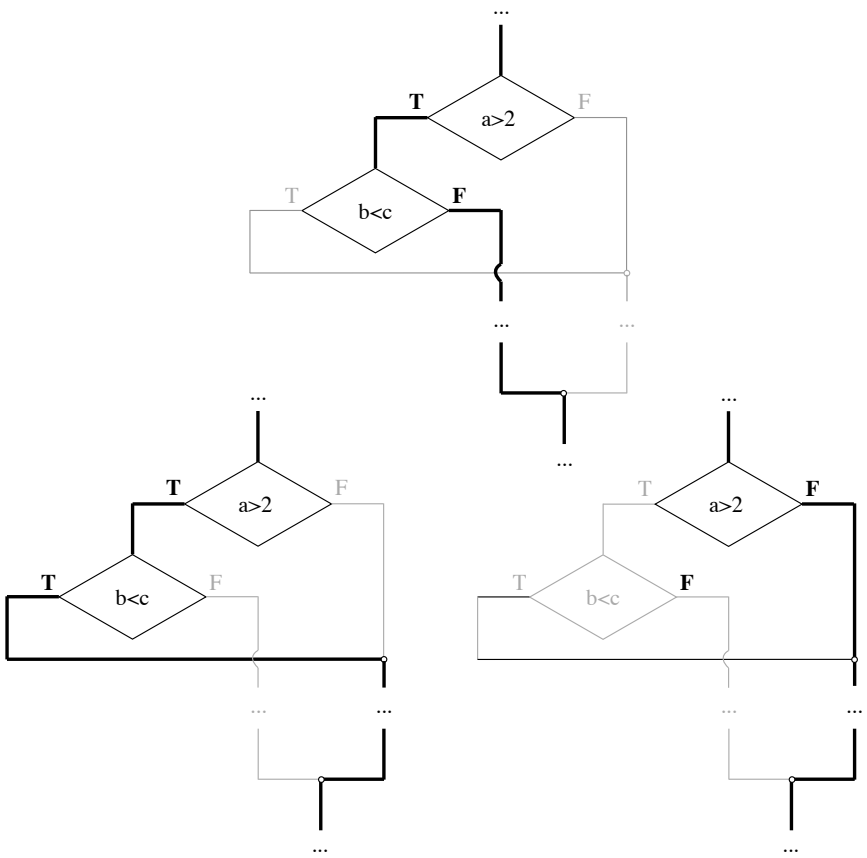


Figure 2.6.: Diagrams showing modified condition/decision coverage

2.2. Constraint Systems

Constraints are formulas derived from functions. A set of constraints makes a *constraint system*. The difference between functions and constraints can be described as follows:

A function F takes some input variable or a set of input variables and gives a result value $F(x)$, e.g.:

$y = F(x)$

A constraint on the other hand is a restriction, stating that the output result of a function shall have a certain value or be within a certain range of values. This is usually done by a relation of the form of an equality or inequality [Soc10]. It can be stated that the output result of a function F shall be *equal*, *less*, *greater*, *less or equal*, or *greater or equal*, or *not equal* to a defined constant value b. This can be denoted in the following way:

$F(x) \triangleright b$

where $\triangleright$ stands for one of $=$, $>$, $<$, $\geq$, $\leq$, or $\neq$ [Wei12, p. 1].

The output of a formula is therefore predefined for constraints. The values which shall be derived from a constraint are those of fitting input variables. One, many, or no sets of input variables might exist, which satisfy the constraint.

The output of every type of formula might be restricted to make a constraint. Here the focus will be on selected special types of constraints, and how to solve them, namely:

- Boolean constraints
- Pseudo-Boolean constraints
- SAT solving

2.2.1. Boolean Constraints

Boolean constraints are based on Boolean functions. Boolean formulas work only on two states, which may be denoted as *false* and *true* or 0 and 1. So there are only two possible output values, which constraints could be restricted to be equal to. Usually, Boolean constrains are restricting the result to be *true*.

$F(x) = 1$

This is not hindering to restrict the result to be *false*. Any constraint can easily be transformed for this by negating it:

$F(x) = 0; \qquad F'(x) = \neg F(x); \qquad F'(x) = 1.$

As the only used form of constraint is $F(x) = 1$, it is usually denoted just as $F(x)$, its restriction to be equal to 1 is implicit in this context.

To solve a given problem with a SAT solver, it is generally necessary to encode it in the *conjunctive normal form (CNF)*. CNF has been the standard form for SAT since first proposed by Davis and Putman in 1958 [FM09, p. 19; Dav01; DP58]. Non-CNF solvers are rarely used [Pre09, pp. 75-76].

A formula in CNF has the form:

$$\bigwedge_i \bigvee_j (\neg) x_{ij}$$

It is a conjunction of i disjunctive terms. The discjunctive terms are made of j literals (j can be a different number for each disjunctive term). Each literal can be either a Boolean variable x_{ij} or its negation $\neg x_{ij}$ [Pre09, p. 75]. For example:

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

A single disjunctive term is called *clause*. A list of clauses is often used instead of explicitly giving the conjunctive normal form. For the given example this is:

1. $a \vee b \vee \neg c$
2. $\neg b \vee c \vee d \vee \neg e$

The CNF of a formula F can be derived from its truth table. Each line of the truth table with a truth value of 0 is contributing as a disjunctive clause to the CNF formula. The literals for one of these clauses can be derived in the following way: If the value of F in the corresponding line is 0, x_{ij} is used as a literal, if not $\neg x_{ij}$ is used [Sch00, p. 28]. The truth table is given in table 2.2.

a	b	c	d	e	$F(a,b,c,d,e)$	Clauses
0	0	0	0	0	1	
0	0	0	0	1	1	
0	0	0	1	0	1	
0	0	0	1	1	1	
0	0	1	0	0	0	$a \vee b \vee \neg c \vee d \vee e$
0	0	1	0	1	0	$a \vee b \vee \neg c \vee d \vee \neg e$
0	0	1	1	0	0	$a \vee b \vee \neg c \vee \neg d \vee e$
0	0	1	1	1	0	$a \vee b \vee \neg c \vee \neg d \vee \neg e$
0	1	0	0	0	1	
0	1	0	0	1	0	$a \vee \neg b \vee c \vee d \vee \neg e$
0	1	0	1	0	1	
					⋮	
1	1	0	0	0	1	
1	1	0	0	1	0	$\neg a \vee \neg b \vee c \vee d \vee \neg e$
1	1	0	1	0	1	
					⋮	

Table 2.2.: Example for deriving CNF from truth table. For all excluded lines F is *true*. For full table see table C.1, appendix C.

The derived six clauses are a valid encoding of the problem. However, they are not an optimal one. The size of an encoding can be measured by the number of clauses, literals, and variables. A smaller encoding is preferable [Pre09, pp. 90-91]. Often, many ways to encode a problem in CNF exist. Using a truth table to derive the encoding is only a feasible approach for very small problems, or dedicated sub-parts of larger problems. A good encoding requires intuition and experimentation [Pre09, p. 93].

For the given example it can be shown that the encoding derived from the truth table is equivalent to the encoding given previously. The following rule can be applied:

$$((x_0 \vee x_1 \vee ... \vee x_n) \wedge (\neg x_0 \vee x_1 \vee ... \vee x_n)) \leftrightarrow (x_1 \vee ... \vee x_n)$$

This transformation is called *resolution*, and is also used internally by SAT solvers. It reduces the number of clauses and literals in a constraint system.

Applying the resolution, the clauses can be merged as follows:

$$\left.\begin{array}{l} \left.\begin{array}{l} a \vee b \vee \neg c \vee d \vee e \\ a \vee b \vee \neg c \vee d \vee \neg e \end{array}\right\} a \vee b \vee \neg c \vee d \\ \left.\begin{array}{l} a \vee b \vee \neg c \vee \neg d \vee e \\ a \vee b \vee \neg c \vee \neg d \vee \neg e \end{array}\right\} a \vee b \vee \neg c \vee \neg d \end{array}\right\} a \vee b \vee \neg c$$

$$\left.\begin{array}{l} a \vee \neg b \vee c \vee d \vee \neg e \\ \neg a \vee \neg b \vee c \vee d \vee \neg e \end{array}\right\} \neg b \vee c \vee d \vee \neg e$$

The metrics (seen in table 2.3) for both encodings clearly show how the constraint system is being optimized by this.

	Clauses	Literals	Variables
As derived from truth table	6	30	5
Optimized encoding	2	7	5

Table 2.3.: Comparison of CNF encodings before and after optimization by size

The clauses can be combined and transformed in several ways, and logic reasoning can be applied on them. From the structure of the conjunctive normal form, some general conclusions can be made:

- For the constraint system to be satisfiable, all clauses have to be satisfiable (logical *and*).
- Correspondingly, if one clause is not satisfiable, the constraint system as a whole is unsatisfiable.
- For a clause to be satisfiable, at least one literal has to be assigned a *true* value (logical *or*).
- Correspondingly, if a clause is not satisfiable, all literals are assigned *false* as value.

2.2.2. Pseudo-Boolean Constraints[1]

Pseudo-Boolean constraints, as the very name already illustrates, are closely related to the concepts of Boolean constraints as discussed previously.

Mathematically the Handbook of Satisfiability describes a pseudo-Boolean function in its broadest sense as: a function that maps n Boolean values to a real number [RM09, p. 695]. So while all input values remain Boolean, real numbers can be used any other place in the formulas and as the result.

$x_1, x_2, x_3, ..., x_n \Rightarrow \mathbb{R}$

Usually, integer numbers are used instead of real numbers as a computational restriction (shown in the formula below on the left side). The unlimited precision real numbers would provide is in general not required.

$x_1, x_2, x_3, ..., x_n \Rightarrow \mathbb{Z} \quad \longrightarrow \quad x_1, x_2, x_3, ..., x_n \Rightarrow \mathbb{Z}_0^+$

All these mappings can be transformed in a way, so that only positive (unsigned) integers are needed (shown in the formula above on the right side).

Pseudo-Boolean constraints can take different forms. Figure 2.7 visualizes these types and subtypes of pseudo-Boolean constraints, as well as the basic operations of linearization and normalization used to transform from one type to another.

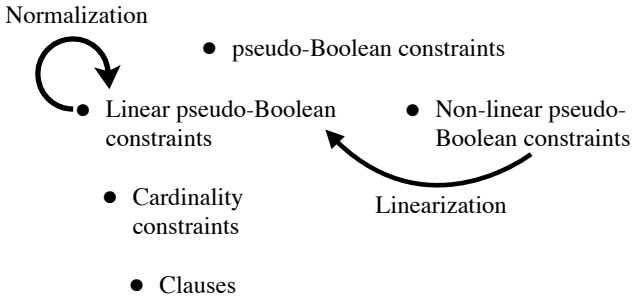


Figure 2.7.: Hierarchy of types of pseudo-Boolean constraints, and basic operations [Wei12, p. 1].

[1] This section includes content based on a seminar paper [Wei12] by the author of this thesis. The original seminar paper can be found in appendix D.

2.2.2.1. Linear Pseudo-Boolean Constraints

A *linear pseudo-Boolean* (LPB) *constraint* has the form:

$\Sigma_j \, a_j l_j \triangleright b$

where a_j and b are integer constants, l_j are literals, and $\triangleright$ is one of $=$, $>$, $<$, $\geq$, or $\leq$ [RM09, p. 696]. Examples:

- $3x_1 + 4x_2 + 5x_3 \geq 7$
- $x_1 + x_2 + x_4 < 3$
- $5 + x_1 = 6 - x_2$
- $8x_4 + 4x_3 + 2x_2 + x_1 \leq 8y_4 + 4y_3 + 2y_2 + y_1$

While the later two examples do not meet the form given above, they obviously could be transformed in a few steps.

For linear pseudo-Boolean constraints there is a *normal form* called *posiform*. The posiform is defined as follows:

$\Sigma_j \, a_j l_j \geq b, \qquad a_j, b \in \mathbb{N} : 0^+$

The comparator must be $\geq$, and all a_j and b must be positive integers or zero [RM09, p. 698].

It is possible to transform any linear pseudo-Boolean constraint. The *normalization* to posiform can be done in two steps, dealing with the two restrictions named above.

1. Getting $\geq$
   - $... > b \;\equiv\; ... \geq b + 1$
   - $... < b \;\equiv\; ... \leq b - 1$
   - $... = b \;\equiv\; ... \leq b, \;\; ... \geq b$
   - $... \leq b \;\equiv\; -1 \times (...) \geq -b$

2. Getting $a_j, b \in \mathbb{N} : 0^+$

   - $(-a_j) \times x_j \geq b$

     $\equiv (-a_j) \times (1 - \neg x_j) \geq b$

     $\equiv -a_j + a_j \neg x_j \geq b$

     $\equiv a_j \neg x_j \geq b + a$

   - $(-a_j) \times \neg x_j \geq b$

     $\equiv (-a_j) \times (1 - x_j) \geq b$

     $\equiv -a_j + a x_j \geq b$

     $\equiv a_j x_j \geq b + a$

From the posiform it is already possible to clearly see some special cases:

- If still $b \leq 0$, the solution is trivial. As left side is at least 0, the constraint is always *true* [RM09, p. 698]. This is called a Tautology [Sch00, p. 19].
- On the opposite extreme, if $\Sigma_j a_j < b$ the constraint is unsatisfiable, as even in case of all literals x_j being 1 their sum remains smaller than b. Moreover, the set of constraints as a whole is unsatisfiable as well, because a set M of formulas is satisfiable if and only if each subset of M is satisfiable [Sch00, p. 34], and a single constraint is the smallest possible subset of M.

2.2.2.2. Non-Linear Pseudo-Boolean Constraints

While being the more general term, *non-linear pseudo-Boolean* (non-LPB) *constraints* are more complex.

A non-linear pseudo-Boolean constraint has the form:

$\Sigma_j \; a_j \Pi_k \; l_{j,k} \triangleright b$

where a_j and b are integer constants, $l_{j,k}$ are literals, and $\triangleright$ is one of $=, >, <, \geq$, or $\leq$ [RM09, p. 697].

The obvious difference to LPB constraints is, that here inside the sum, products can appear. Examples:

- $7x_1x_2 + 3x_1 + x_3 \geq 8$
- $7x_2 + 3x_1x_3 + 2x_4x_2x_1 \geq 8$

It is possible to transform any non-linear pseudo-Boolean constraint into a set of linear pseudo-Boolean constraints. This process is called *linearization*. A method to do this transformation is based on the work of Fortet [For60].

Each product (for being not allowed in linear pseudo-Boolean constraints) is to be substituted by a newly introduced variable.

$l_1 \times l_2 ... \times l_n \Leftrightarrow v$

Two additional constraints are introduced, to assure the new variable behaves equivalent to the product.

$1l_1 + 1l_2 ... + 1l_n - nv \geq 0$

$1l_1 + 1l_2 ... + 1l_n + 1v \geq 1$

These constraints enforce bidirectional:

- one or more $l = 0 \Leftrightarrow v = 0$,
- all $l = 1 \Leftrightarrow v = 1$,

2.2.2.3. Cardinality Constraints

Cardinality constraints can be seen as a subtype of LPB constraints. There are three types of cardinality constraints:

- $atleast(k, \{x_1, x_2, ..., x_n\})$
- $atmost(k, \{x_1, x_2, ..., x_n\})$
- $exactly(k, \{x_1, x_2, ..., x_n\})$

These constraints can be translated to (linear) pseudo-Boolean constraints, where each $a_j = 1$ and $b = k$. So they can be seen as a special case of pseudo-Boolean constraints.

- $atleast(k, \{x_1, x_2, ..., x_n\}) \equiv x_1, x_2, ..., x_n \geq k$
- $atmost(k, \{x_1, x_2, ..., x_n\}) \equiv x_1, x_2, ..., x_n \leq k$
- $exactly(k, \{x_1, x_2, ..., x_n\}) \equiv x_1, x_2, ..., x_n = k$

2.2.2.4. Relation to Clauses

Clauses, as discussed in the previous section, can be seen as special cases of pseudo-Boolean and cardinality constraints. A clause is equivalent to an *atleast* cardinality constraint with $k = 1$, and can further on be translated to a linear pseudo-Boolean constraint:

$x_1 \vee x_2 \vee ... \vee x_n$

$\equiv atleast(1, \{x_1, x_,..., x_n\})$

$\equiv x_1 + x_2 + ... + x_n \geq 1$

In the reverse direction it is also possible to translate a pseudo-Boolean constraint to a set of clauses.

> "Pseudo-Boolean Constraints can be translated to SAT. This is one way among others used to solve Psuedo-Boolean constraint systems [RM09, pp. 710ff] and surprisingly this approach can be quite efficient as demonstrated by the minisat+ solver [ES06] in the several evaluations of Pseudo-Boolean solvers [MR07]. [...] A naive approach is to translate a [pseudo-Boolean] constraint as a set of clauses which is semantically equivalent and uses the same variables" [RM09, p. 726].

A number of methods exist to approach this translation:

- binary adder
- binary decision diagram

- unary notation

For higher efficiency the appropriate encoding is best chosen for each constraint individually [RM09, pp. 726-728].

> "Pseudo-Boolean constraints are more expressive than clauses and a single Pseudo-Boolean constraint may replace an exponential number of clauses. More precisely, the translation of a non-trivial Pseudo-Boolean constraint to a set of clauses which is semantically equivalent (in the usual sense, i.e. without introducing extra variables) has an exponential complexity in the worst case" [RM09, p. 701].

The expressiveness of types of constraints is also shown in figure 2.8. One has to keep in mind, that the clauses of the CNF formula, which were derived in the way described above, are not necessarily the shortest possible representation [Sch00, p. 29].

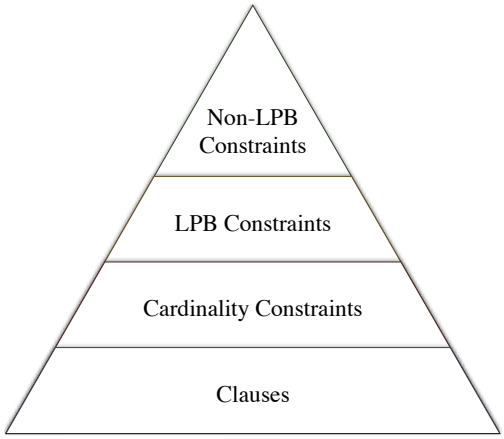


Figure 2.8.: Expressiveness of constraint types (not to scale).

Additionally to the decision problem, pseudo-Boolean constraints can also be used to solve a different problem: the optimization problem.

The decision problem, as already known from Boolean constraints, asks if there exists a solution, so if the constraints are satisfiable at all. The optimization problem on the other hand is to find out what is the one "best" solution. This optimal solution is defined by minimizing a cost function subject to a set of constraints. The cost function is a function, assigning a specified cost to each literal.

This concept is unknown to Boolean constraints in SAT. The optimization problem can only be emulated. To do this, the constraint system has to be encoded and solved iteratively until the lowest cost is found for which it is still satisfiable. Binary search may be used to find this optimal value with a minimal number of iterations.

2.2.3. SAT Solving

SAT solvers are being used to automatically solve SAT problem instances. This means to check whether a constraint system is *satisfiable* (i.e. at least one assignment exists) or not (*unsatisfiable*). When the constraint system is satisfiable, one possible assignment for all Boolean variables is typically provided by the SAT solver. The internal behavior of a SAT solver is very interesting and complex. As here the focus is on the application of SAT, the internal details of solvers will not be discussed. A SAT solver can in this context be seen as a black box. The solver is taking a problem as input giving a solution as output. Different SAT solvers have thereby different characteristics, based on which they might be better suited for one or another kind of problem.

SAT is used to solve many real-world problems in different fields, including for example verification problems in software [BHvMW09, p. v]. This has of course been made possible along with the development of applicable SAT solvers. Thanks to ongoing progress, SAT solvers have become a very efficient way to approach these problems. It has been said that "nowadays more problems are being solved faster by SAT solvers than other means" [FM09, p. 3]. SAT competitions, challengers, and benchmark problems have fostered continuous progress in SAT solving technology [BBH$^+$13; BHvMW09, p. vi; HS00; Pre09, p. 81]. This development is still continuing with significant progress and advancements [BHvMW09, p. v].

While this has become a big advantage for the developers of SAT solvers to evaluate their solutions, it is also a very useful guideline when an efficient solver shall be chosen for practical application on a problem.

To work with Boolean constraint systems on computers, a machine readable standardized

file format is used. It was first introduced for the DIMACS[2] Challenge in 1993, and is therefore referred to as the DIMACS file format [JT96; Pre09, p. 81].

A DIMACS file is a plain text file with a certain structure. It starts optionally with a part called preamble, which contains information about the file. These lines start with a lower case "c" as their first letter, to mark them as comments.

```
1 c [comment]
```

"[comment]" can be replaced with any desired text. There can be any number of comment lines from zero to an arbitrary number of comment lines (as long as each of these lines starts with "c").

The first line actually used by the SAT solver is called problem line, marked with a "p" as first character. The line has the form

```
2 p [encoding format] [parameters]
```

"[encoding format]" might be either "cnf"-encoding for a problem that is in conjunctive normal form, or "sat"-encoding for a problem that can be in any form (including CNF). The parameters are different, depending on the encoding format.

For *CNF files* the problem line has the form:

```
2 p cnf [variables] [clauses]
```

"[variables]" and "[clauses]" are to be replaced by positive integers giving the respective actual numbers of clauses and variables in the file to follow. These parameters are required for initialization, and also work as check sums – when inconsistent with the actual numbers, the solver might abort the execution[3].

The rest of the file contains the clauses. Each literal in a clause has an integer number as identifier of the represented Boolean variable. If the literal represents a negated Boolean

[2]DIMACS – Center for Discrete Mathematics & Computer Science at Rutgers State University of New Jersey

[3]The exact behavior is of course depending on the actual solver used

variable, the variable number will be preceded by a minus sign. Literals are separated by certain whitespace characters (spaces, tabs, or newlines). Between two clauses a zero is used as separator. A common formatting to ease readability is to separate the literals by spaces and use a new line after each 0, so every line contains just a single clause. However, this is not required and not always done. The order of clauses in the file and the order of literals in each clause is irrelevant to the solver.

The DIMACS format requires the literals to be represented by natural numbers. When the naming scheme used is a different one, they have to be mapped. For the example given previously, a possible mapping is $\{a \to 1, b \to 2, c \to 3, d \to 4, e \to 5\}$. A comment line can be used to include the information of this mapping in the file.

A DIMACS CNF file for the example would be:

```
c variable mapping: {a->1, b->2, c->3, d->4, e->5}
p cnf 5 2
1 2 -3 0
-2 3 4 -5 0
```

SAT files use a different format. The problem line has the form

```
p sat [variables]
```

"`[variables]`" is here as well to be replaced by a positive integer giving the respective actual number of variables used. The handling of variable numbering and negation is the same as for CNF-encoding.

A "+"-sign represents an *or*-operation and a "*"-sign represents an *and*-operation. The literals, the operation is to be applied on, follow in brackets after the sign. Both of these operations might be nested in any possible way.

A DIMACS SAT file for the given example would be:

```
c variable mapping: {a->1, b->2, c->3, d->4, e->5}
p sat 5
(*(+(1 2 -3)
   +(-2 3 4 -5)))
```

It is common practice to use "*.cnf" as extensions for files in CNF-encoding, and "*.sat" for files in SAT-encoding [DIM93].

A separate file format exists, which can represent pseudo-Boolean constraints [RM12], along with pseudo-Boolean solvers. Alternatively, pseudo-Boolean constraints can be translated to SAT [RM09, pp. 710ff]. This approach has been proven to be quite efficient [ES06, MR07]. So even when some aspects might be modeled using pseudo-Boolean constraints, neither separate file formats, nor separate solvers will be needed.

While the input format for SAT solvers has been standardized, the output format has not. The output of different SAT solvers contains similar information, but its representation differs.

Below output of the example is shown for PicoSAT [Pic10] and MiniSAT [Whe08] solvers.

PicoSAT encoding:

```
c [Comment]
s SATISFIABLE
v 1 2 -3 4 -5 0
```

MiniSAT encoding:

```
SAT
1 2 -3 4 -5 0
```

Both solvers will state whether the given problem is satisfiable or not, either by "`SAT`" / "`SATISFIABLE`" or "`UNSAT`" / "`UNSATISFIABLE`". If it is satisfiable, a possible assignment of the variables will also be given. The output uses the same enumeration of variables as the input format[4]. A minus sign indicates that the variable has to be assigned the value 0, otherwise the value 1.

Based on the variable mapping these values can be set into $F(a, b, c, d, e)$ as follows: $F(1, 1, 0, 1, 0) = 1$. So this quintuple is an element (possibly one among many) of the solution set $\mathbb{S}$: $(1, 1, 0, 1, 0) \in \mathbb{S}$.

[4] The same assignment of variables is used here to compare both outputs. When actually used, different solvers are likely to generate different solutions

III. Approach on Encoding

The overall-workflow consists of three major parts:

- the *encoder*, generating the constraint system,
- the *SAT solver*, solving the constraint system, and
- the *decoder*, generating the test data.

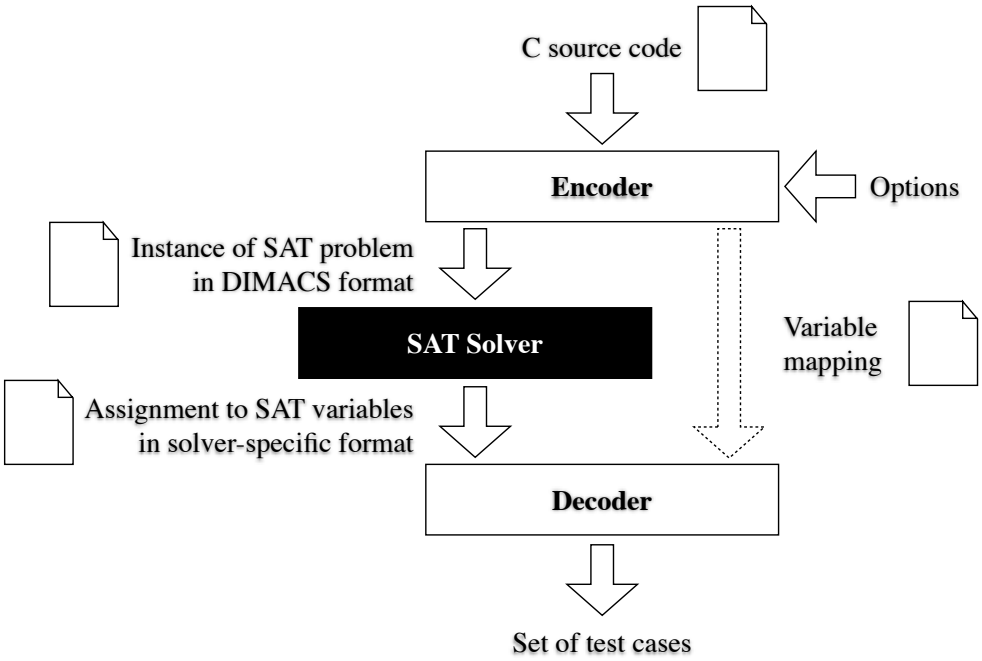


Figure 3.1.: Steps for deriving test cases for a type of coverage with SAT.

The SAT solver can be used out of the box, and the decoder is in comparison rather small and simple. The encoder is by far the most complex part to be developed here. This chapter will therefore address the theoretical approach on encoding aspects of a program-

ming language with SAT. Some of those aspects are specific for C, others could as well be adapted and reused for other programming languages. The practical implementation of the encoder, as presented later, makes use of the theoretical approach discussed here.

3.1. Encoding as a Model

The system of Boolean constraints to be generated can be seen as a model. Stachowiak defined the fundamental properties of a model, which can be seen as a guideline for the generation of any model, as follows:

> "[The] Fundamental Model Properties [are]
>
> 1. Mapping: Models are always models *of something*, i.e. mappings from, representations of natural or artificial originals, that can be models themselves.
> 2. Reduction: Models in general capture *not all* attributes of the original represented by them, but rather only those seeming relevant to their model creators and/ or model users.
> 3. Pragmatism: Models are not uniquely assigned to their originals per se. They fulfill their replacement function a) for *particular* – cognitive and/or acting, model using *subjects*, b) within *particular time intervals* and c) restricted to *particular mental or actual operations*" [Sta73, pp. 131-133; Mod12].

A simplistic approach would be to make a model representing the complete source code of a program. However, this is not necessary. With the reduction to the essential attributes, and the pragmatism to model only those parts of the original relevant for the task, the model can be kept smaller. For the task of finding test cases for types of control flow coverage, two assumptions can be made for the model:

1. All instructions after the last *if*-statement can be ignored, as it can not have influence on previous conditions and decisions.

2. All instructions before an *if*-statement are only relevant, when it changes the value of at least one

   a) *directly relevant* variable used in at least one *if*-statement in code below, or

   b) *indirectly relevant* variable used to change the value of at least one directly or indirectly relevant variable in code below.

The source code can therefore be analyzed from its end to its start, reversing the order of execution. The code can be reduced to considerable extent. This leaves only the relevant parts as a basis for the model.

A number of example instructions will be used to show this process.

Example 1 – relevance of variables and instructions:

```
1  e = ... b ... ;
2  d = ... a ... ;
3  c = ... ;
4  b = ... ;
5  a = ... b ... ;
6  a += ... ; a -= ... ; a /= ... ; a *= ... ; a %= ... ;
7  ... a++ ...
8  ... a-- ...
9  ... ++a ...
10 ... --a ...
11 if ( ... a ... )
12 ...
```

1. Directly relevant instruction (if statement)

   directly relevant variable: a

```
11 if ( ... a ... )
```

2. Indirectly relevant instructions of n-th grade

   indirectly relevant instruction of 1st grade

```
5  a = ... b ... ;
```

   alternatives to “=”

```
6 a += ... ; a -= ... ; a /= ... ; a *= ... ; a %= ... ;
```

increments and decrements

only in-/decrementation is relevant, not the complete instruction

```
7  ... a++ ...
8  ... a-- ...
9  ... ++a ...
10 ... --a ...
```

indirectly relevant variable of 1st grade: `b`

indirectly relevant instruction of 2nd grade

```
4 b = ... ;
```

3. Irrelevant instructions

irrelevant variables: `e, d, c`

```
1 e = ... b ... ;
2 d = ... a ... ;
3 c = ... ;
```

All instructions below the very last if statement are irrelevant

```
12 ...
```

Example 2 – relevance of parameters:

```
1 funct ( w, x, z )
2
3   y = ... x ... ;
4
5   if ( ... z ... y ... )
```

Analogous to the variables, parameters can be differentiated:

- Directly relevant parameter: `z` (used in an *if*-statement)
- Indirectly relevant parameter (1st grade): `x` (changes value of a variable used in an *if*-statement)

- Irrelevant parameter: `w` (has no influence on any *if*-statement)

Example 3 – temporary relevance of a variable:

It is also possible for variables to become irrelevant above a point.

```
1 ...
2 int a;
3 a = 1;
4 a = c;
5 c = 0;
6 if ( ... a ... ) ...
```

The instruction in line 6 makes *a* directly relevant. The use of *c* in line 5 is irrelevant. The instruction in line 4 makes *c* indirectly relevant. As with line 4 the value of *a* is only determined by *c* and not by any previous value of *a*, *a* becomes irrelevant above line 4. In line 3 *a* is therefore not relevant. However, the declaration of *a* in line 2 is relevant. Although it does not have any influence on the program flow, it is necessary to keep the reduced code valid C-code. The declaration of a variable, which is relevant at some point, has to remain in the reduced code.

Example 4 – relevance of variables in alternate paths:

```
1  ...
2  int a;
3  int b;
4  int c;
5  a = 1;
6  b = 2;
7  c = 3;
8  if ( ... ) {
9    a = 1;
10   a = b;
11   b = 0;
12   c = 0;
13 } else {
14   a = 1;
15   a = c;
16   b = 0;
17   c = 0;
18 }
19 if ( ... a ... ) ...
```

At the bottom of the *if*-branch (line 13) the same variables are relevant as at the bottom of the *else*-branch (line 18). Above the *if*-statement in line 8, the set of relevant variables is the union of the relevant sets of variables at the top of the *if*- and *else*-branches (line 9 and 14).

3.2. Basic Operations

To model the behavior of a program according to its given source code, basic operations must in some way be translated to sets of constraints. Modeling this behavior is the basis upon which the different kinds of control flow coverages can be applied.

These operations include:

- input parameters;
- variables
  - declaration,
  - assignment, and
  - use;
- hard-coded (numerical) values;
- arithmetic operations;
- logical operations; and
- conditional behavior

Here conditional behavior models the selective execution of code based on conditions in *if* and *while* statements. In this context it does not yet enforce any type of coverage. This is not part of this step, but will be discussed in section 3.3.

3.2.1. Variable Types

The model must be able to represent the values of several variable types used during the execution of a program. The types covered here will be the *basic types* integer (`short`, `int`, `long`, `long long`) and character (`char`). All of these in signed and unsigned variants.

Composite types like arrays, strings or structs could be assembled from these basic primitive types, but are not yet supported. One area that will be out of scope are floating point numbers, as they are currently under research separately. The guidelines are given by the actual implementation of these types in the programming language.

> With a few exceptions, most programming languages offer basic data types that have bit-vector semantics, i.e., they have a bounded range defined by the number of bits allocated for them. In case of an unsigned number with n bits, this range is $0,...,2^{n-1}$. If this range is exceeded by an arithmetic operation, the result wraps around, which means that the modulo 2^n of the actual result is computed. [Kro09, p. 505]

For C these restrictions hold true, and the datatypes that have to be modeled follow bit-vector semantics. However, it would for example not be impossible to model a programming language making use of unbounded size datatypes. The abstraction from the machine level that such kind of programming languages offer, could (with additional work) be modeled with Boolean constraints as well. With C being closer to the machine operations, the translation to a constraint system can be approached rather straight forward.

> To generate a constraint system "A possible solution is to model programs by closely following the hardware that is used to implement their execution environment: bit-vectors can be modeled as a set of Boolean variables, and bit-vector operators as a set of Boolean functions." [Kro09, p. 506]

This means that a variable of the program can be represented by a set of n Boolean variables, where n is the bitlength of the variable's type. As seen in figure 3.2, the bits of a value are being mapped to Boolean variables left to right, starting with 1 (0 being a

reserved value). This is for example different from an integer variable, whose internal numbering is right to left, starting with 0.

The Boolean SAT variables will always be given in output from lowest (left) to highest number (right). This includes the output of the SAT solver, which is not changeable. The chosen scheme allows the output to be better human-readable. Otherwise numbers would have to be read reverse. For the implementation addressing in this order is not more problematic than in opposite direction.

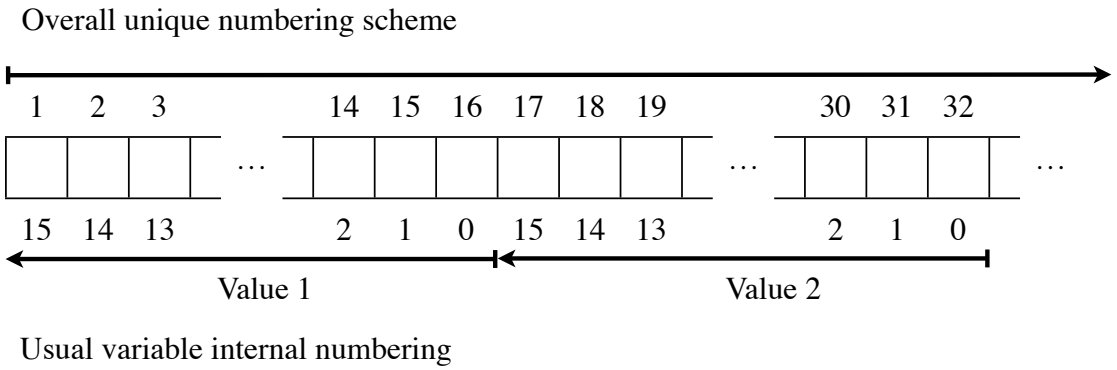


Figure 3.2.: Mapping bits to Boolean variables. Example for two 16-bit values.

> This approach is commonly referred to as 'bit-blasting', as the word-level structure is lost. [Kro09, p. 511]

To compensate for the loss of word-level structure, it is important to generate a list, which is mapping program variables from the source code to their corresponding sets of Boolean variables and vice versa. This can be done in parallel to generating the constraint system, and will later be needed as input for the decoder.

However, a 1 to n mapping is only partially true. The variables of procedural programming languages can get reassigned different values again and again over the course of the program's runtime. A Boolean variable in a constraint system is in contrast not changeable over time but is decided to be a fix value: either *true* or *false*. To model the reassignment of a variable, it is therefore necessary to use a new set of Boolean variables to represent the new value. So a multitude of n Boolean variables is needed to represent the value of a variable in the program over time.

3.2.2. Variable Declaration

Declaration of a variable means to state that a variable exists and to define the type it will have.

In our model this means that we have to add a mapping for the declared program variable to a list of Boolean variables of the constraint set. As with just a declaration without assignment of any value to the new variable, a value is non existent (treated in C as `NULL` value). For the model this non existence of a value means that the program variable is mapped to an empty list. The example below shall demonstrate this.

In the beginning the mapping is empty.

$\{\}$

With the declaration of an integer a,

```
1 int a:
```

the mapping will be

$\{a \rightarrow \{\}\}$

when adding a second integer b

```
2 int b:
```

the mapping would become

$\{a \rightarrow \{\}, b \rightarrow \{\}\}$

and so on.

Additional variables might be introduced. This can for example be necessary to hold intermediate results for some operations. The problem of conflicting names by accidentally choosing a variable name already existing in the input source code (but maybe not yet at that point in the mapping) has to be avoided. A simple approach is to just enumerate theses variables. As variables used in the given C source code are restricted to always

begin with upper/lower case letters or the underscore character, the variable name can not start with any numeric character. A new introduction is therefore possible without checking for collisions by remembering the latest introduced variable's number.

3.2.3. Numeric Constants

Numerical values can appear in two contexts. They can be merely an assignment of a value to a variable.

```
1 a = 3;
```

Or they can be used in arithmetic or logical operations.

```
2 b = a + 4;
```

Beside decimal representation, octal or hexadecimal numbers might be used as well. Character (char) types of variables can be handled the same way as numerics. In fact the letters used are just a different visual representation of a numeric value – the corresponding encoding of the character (mostly ASCII).

All numerics have to be transformed to binary numbers for further handling. When for example the mapped Boolean variables are from 1 to 16, and then the number to be assigned is:

$107F_{16} = 4223_{10} = 10177_8 = 0001000001111111_2$

With a pseudo-Boolean constraint this could be enforced by:

$2^{14}x_1 + 2^{13}x_2 + 2^{12}x_3 + 2^{11}x_4 + 2^1x_5 + 2^{10}x_6 + 2^9x_7 + 2^8x_8 + 2^7x_9 + 2^6x_{10} + 2^5x_{11} + 2^4x_{12} + 2^3x_{13} + 2^2x_{14} + 2^1x_{15} + 2^0x_{16} = 4223$

The corresponding set of constraints to enforce this would be:

$\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \wedge \neg x_5 \wedge \neg x_6 \wedge \neg x_7 \wedge \neg x_8 \wedge \neg x_9 \wedge x_{10} \wedge x_{11} \wedge x_{12} \wedge x_{13} \wedge x_{14} \wedge x_{15} \wedge x_{16}$

A single literal constraint is used for each digit. When being negated it enforces a 0 of the value, without negation a 1.

Only for the assignment use, these constraints could be directly used on the variable. If used inside of some operation, it can be used on an additionally introduced variable as discussed before. This second way can also be used for direct assignments, which will allow for a uniformal handling of numerical values. The very small overhead would only be in the mapping (one additional entry).

For negative numbers the *two's complement* representation has to be applied, which can shortly be described as the opposite of the representation of the number's absolute value decremented by 1 ($|n| - 1$). The following example shows how the corresponding negative number, of same absolute value as in the previous example, would be represented:

$-4223_{10} \triangleq 1110111110000001_2$

While used for an usigned type this decimal encoding would be a different number:

$1110111110000001_2 = 61313_{10}$

The leftmost bit will be 1 for all negative numbers. This can be used to distinguish these in signed integer values.

3.2.4. Variable Assignment

There are two ways to model the assignment of values to a variable that will be used here:

- Handling assignments in the variable mapping
- Introducing constraints to assure identical values.

Handling assignments in the variable mapping comes with the advantage of not introducing any new Boolean variables, clauses or literals in the constraint system.

May this be the initial variable mapping before an assignment:

$\{a \to \{1\}, b \to \{17\}, c \to \{33\}\}$

An assignment operation assigns the value of b to a:

```
1 a = b;
```

After this assignment operation the mapping would become:

$\{a \rightarrow \{1, 17\}, b \rightarrow \{17\}, c \rightarrow \{33\}\}$

With the latest mapping appended at the end of a list of mappings, the information on previous mappings is not lost.

Note that while a and b are being mapped to the same Boolean variables at this point, this should not be confused with a pointer-like concept. The Boolean variables are static. Changing the value of b after this will not affect the value of a.

```
2 b = c;
```

Instead, a new mapping for b is added.

$\{a \rightarrow \{1, 17\}, b \rightarrow \{17, 33\}, c \rightarrow \{33\}\}$

Even though this way to handle assignments is efficient by generating a smaller system of constraints to model a program's behavior, it also has some disadvantages.

As the assignment is modeled outside of the constraint system in the variable mapping, it is not accessible from inside the system to be bound to conditions. When inside an *if*- or an *else*-branch a new value is assigned to a variable, whose value will be accessed at a later point (this excludes variables of only local scope), this assignment is not absolute, but depending on the condition of the if-statement.

An assignment has to be modeled with constraints, enforcing equality in these cases.

$a \leftrightarrow b$

The corresponding truth table and the derived clauses can be seen in table 3.1.

a	b	Valid?	Clauses
0	0	1	
0	1	0	$a \vee \neg b$
1	0	0	$\neg a \vee b$
1	1	1	

Table 3.1.: Derived clauses for the assignment operation

Beside handling conditional assignments (inside if/while) this is also the basis for converting types of shorter bitlength to types of larger bitlength and signed to unsigned types (both vice versa).

3.2.5. Variable Conversion

Type conversions are based on the modeling of assignment with constraints. There are three cases of conversions to be distinguished (see figures 3.3, 3.4).

For conversions to types of smaller bitlenght the same method is used regardless whether the types are signed or unsigned. The equivalency constraints are simply applied to those bits available in both the source and target value, starting from the right. The remaining left bits of the source value are not considered.

For conversions to types of larger bitlength, it has to be differentiated whether the source type is signed or unsigned.

For signed source types, the remaining left bits of the target value, for which no corresponding bit in the source value exists, are to be set equal to the leftmost bit of the source value. The interpretation of a value will change when converting a negative value of a signed type to a value of an unsigned type.

For unsigned source types the remaining bits of the target value are to be assigned to be zero instead.

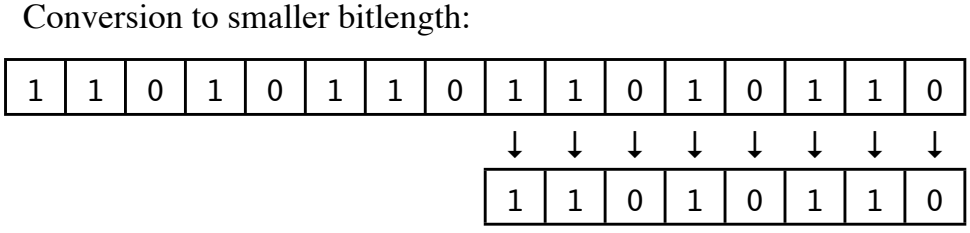


Figure 3.3.: Conversions to smaller bitlength using example values.

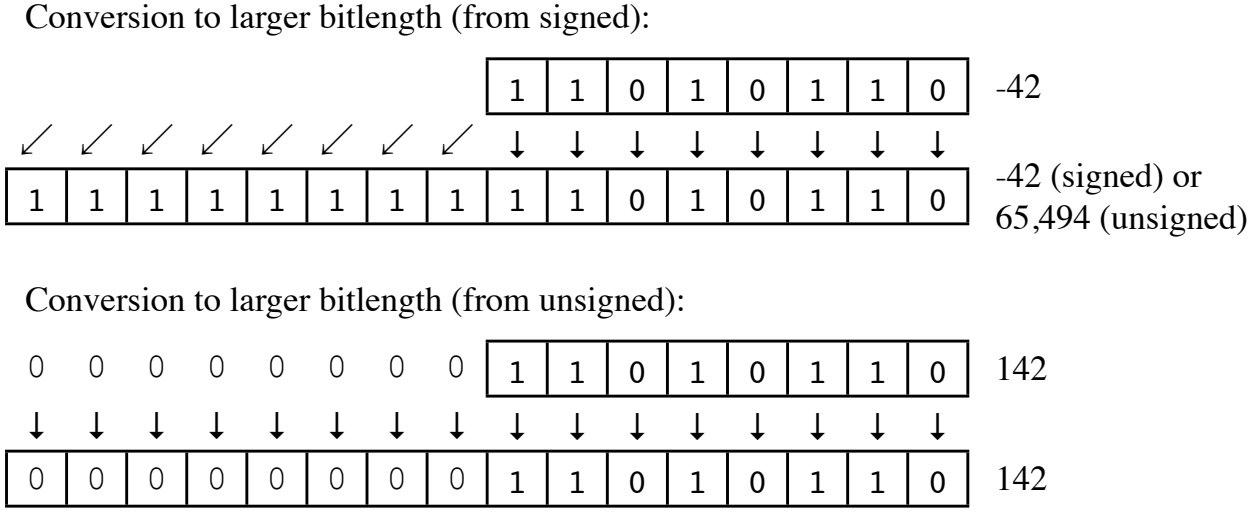


Figure 3.4.: Two cases of type conversions to larger bitlength using example values.

3.2.6. Logical Operations

Of all operations in a programming language, logical operations are the ones to be encoded most naturally in SAT. They are also very important, fundamental operations.

If-statements make direct use of logical operations. Additionally arithmetic operations – as discussed in the following section 3.2.7 – are based on logic operations. Finally, encoding a type of control flow coverage (section 3.3) will require logical relations among conditions and decisions across test cases.

Logical operations include:

- conjunction (*and*)
- disjunction (*or*)
- negation (*not*)

There is no *xor* as a logical operator in C. Only a bitwise *xor*-operation exists. Yet, a model of it is needed as a component of more complex operations, e.g. addition.

C actually uses a whole integer to store the true or false result as a value 1 or 0. However, it is not necessary to model it this way. A single Boolean variable is sufficient, easier to

handle, and keeps the constraint system smaller.

The truth tables and derived clauses for all four used logical operations are given in the following tables 3.2 to 3.9.

The extended truth table lists all possible combinations of the input and output variables. The clauses for CNF have to be derived from those lines which are not valid, i.e. do not appear in the regular truth table.

The clauses are not based on the output column of the truth table. Deriving them from the output column would enforce for example for the *and* table a and b to be true. However, this is not what is intended here, the intention is to have valid assignments for all three variables in the formula $c = a \wedge b$, including those assignments where a, b or c are assigned to be *false*.

Inputs		Output
a	b	$c = a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.2.: Truth table for *and*

Inputs		Output	Valid?	Clauses
a	b	$c = a \wedge b$		
0	0	0	1	
0	0	1	0	$a \vee b \vee \neg c$
0	1	0	1	$\}\ a \vee \neg c$
0	1	1	0	$a \vee \neg b \vee \neg c$
1	0	0	1	
1	0	1	0	$\neg a \vee b \vee \neg c$
1	1	0	0	$\neg a \vee \neg b \vee c$
1	1	1	1	

Table 3.3.: Extended truth table and derived clauses for *and*

Inputs		Output
a	b	$c = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.4.: Truth table for *or*

Inputs		Output	Valid?	Clauses
a	b	$c = a \vee b$		
0	0	0	1	
0	0	1	0	$a \vee b \vee \neg c$
0	1	0	0	$a \vee \neg b \vee c$
0	1	1	1	
1	0	0	0	$\neg a \vee b \vee c$
1	0	1	1	$\}\ \neg a \vee c$
1	1	0	0	$\neg a \vee \neg b \vee c$
1	1	1	1	

Table 3.5.: Extended truth table and derived clauses for *or*

Inputs		Output
a	b	$c = a \veebar b$
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.6.: Truth table for *xor*

Inputs		Output	Valid?	Clauses
a	b	$c = a \veebar b$		
0	0	0	1	
0	0	1	0	$a \vee b \vee \neg c$
0	1	0	0	$a \vee \neg b \vee c$
0	1	1	1	
1	0	0	0	$\neg a \vee b \vee c$
1	0	1	1	
1	1	0	1	
1	1	1	0	$\neg a \vee \neg b \vee \neg c$

Table 3.7.: Extended truth table and derived clauses for *xor*

Input	Output
a	$b = \neg a$
0	1
1	0

Table 3.8.: Truth table for *not*

Input	Output	Valid?	Clauses
a	$b = \neg a$		
0	0	0	$a \vee b$
0	1	1	
1	0	1	
1	1	0	$\neg a \vee \neg b$

Table 3.9.: Extended truth table and derived clauses for *not*

3.2.7. Arithmetic Operations

Arithmetic operation used here include:

- addition
- subtraction
- multiplication
- division
- modulo

And further on as special cases of addition respectively subtraction:

- incrementation
- decrementation

The covered types of instructions are the following (and any valid combination of them):

```
// Addition / Increment
c = a + b;
a += b;
a++;
++a:

// Subtraction / Decrement
c = a - b;
a -= b;
a--;
--a:

// Multiplication
c = a * b;
a *= b;

// Division
c = a / b;
a /= b;

// Modulo
c = a % b;
a %= b;
```

> "The most commonly applied approach to check satisfiability of these formulas is to replace the arithmetic operators by circuit equivalents to obtain a propositional formula" [Kro09, p. 511].

3.2.7.1. Addition

The adder logic used in circuits to sum numbers is made up by n parallel 1-bit full adders, where n is equal to the bit length of the numbers used. The rightmost bits might be added by a half adder (dropping c_0 input).

To model addition for the constraint system, there is more than one possible approach:

1. Treating a 1-bit full adder as a black box and modeling it according to its external behavior (full adder approach).
2. Modeling two half adders and their connecting *or*, which the full adder consists of

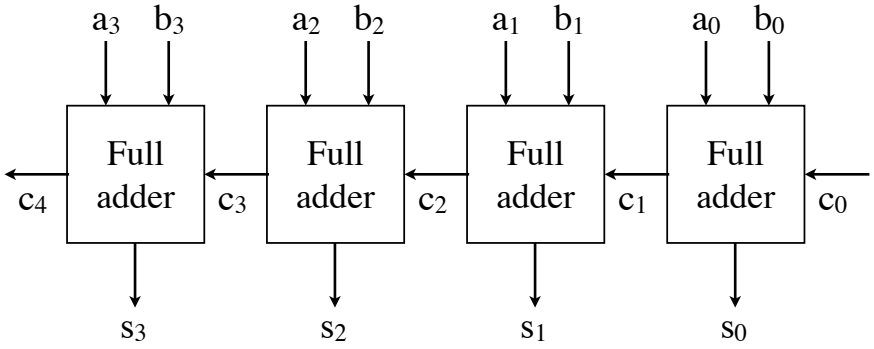


Figure 3.5.: Exemplary composition of a 4-Bit Ripple Carry Adder [Bur06]

(half adder approach).

3. Breaking down the half adders further, modeling the underlying *xor* and *and* (basic logic approach).

A full adder, half adders, and their underlying logic can be seen in figure 3.6. The clauses for each approach can be derived from the respective truth tables. The truth tables of full and half adder are shown in tables 3.10 and 3.11. The required sub-operations of *and*, *or*, and *xor* are being discussed in section 3.2.6 on logical operations.

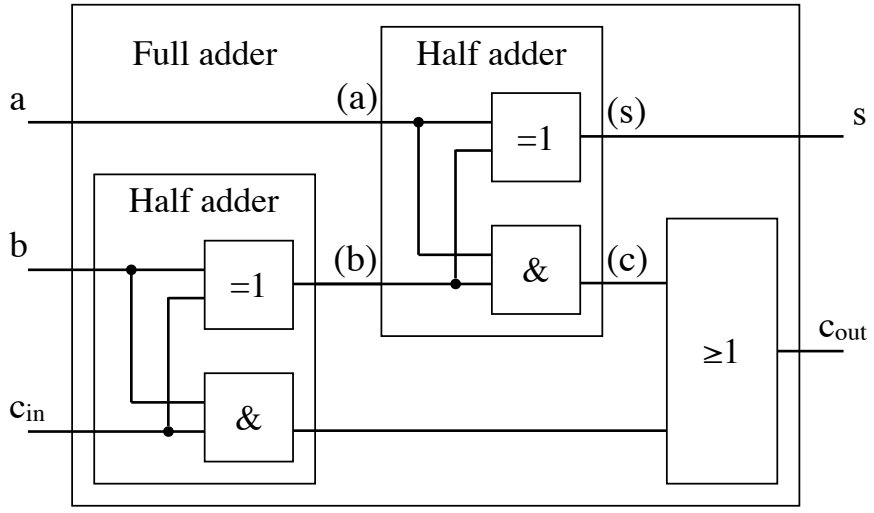


Figure 3.6.: Composition of full and half adders [RP06, pp. 307f]

In the following the three approaches shall be evaluated and one of them shall be chosen to be used.

Inputs			Outputs	
a	b	c_{in}	c_{out}	s
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

Table 3.10.: Truth table for full adder

Inputs		Outputs	
a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 3.11.: Truth table for half adder

Inputs			Outputs		Valid?	Clauses
a	b	c_{in}	c_{out}	s		
0	0	0	0	0	1	
0	0	0	0	1	0	$a \vee b \vee c_{in} \vee c_{out} \vee \neg s$
0	0	0	1	0	0	$a \vee b \vee c_{in} \vee \neg c_{out} \vee s$ } $a \vee b \vee c_{in} \vee \neg c_{out}$
0	0	0	1	1	0	$a \vee b \vee c_{in} \vee \neg c_{out} \vee \neg s$
0	0	1	0	0	0	$a \vee b \vee \neg c_{in} \vee c_{out} \vee s$
0	0	1	0	1	1	

⋮

Table 3.12.: Derived clauses for full adder. For complete version see table E.2. In total: 32 lines of values, 24 derived clauses, reduced set of 16 clauses

Inputs		Outputs		Valid?	Clauses
a	b	s	c		
0	0	0	0	1	
0	0	0	1	0	$a \vee b \vee s \vee \neg c$
0	0	1	0	0	$a \vee b \vee \neg s \vee c$ } $a \vee b \vee \neg s$
0	0	1	1	0	$a \vee b \vee \neg s \vee \neg c$
0	1	0	0	0	$a \vee \neg b \vee s \vee c$ } $a \vee \neg b \vee s$
0	1	0	1	0	$a \vee \neg b \vee s \vee \neg c$

⋮

Table 3.13.: Derived clauses for half adder. For complete version see table E.1. In total: 16 lines of values, 12 derived clauses, reduced set of 8 clauses

The comparison of the approaches, can be done by evaluating the metrics: number of clauses, number of literals, and number of variables. See table 3.14 for the overview of these values. Note that the numbers below refer to the already simplified set of clauses. The variables used by any of the three approaches include a, b, c_{in}/c_{out} and s, which are common to all three approaches, because they are the externally needed variables, representing in- and output.

	Clauses	Literals	Variables
Full adder	**16**	72	**5**
Half adder	19	64	8
Basic logic	17	**50**	8

Table 3.14.: Comparison of the approaches

A summary for the evaluation of each approach can be given as follows:

- With the full adder modeling there are 16 clauses, eight of them having five, and eight having four literals. This sums up to 72 literals in total. No variables are added additionally to c_{in}/c_{out} and s, so the total remains five.
- With the half adder modeling (made from two half-adders and one *or*) there are 19 clauses, eight of them having four, ten having three, and one having two literals. This sums up to 64 literals in total. Three variables (c_1, c_2, and s_1) have to be introduced additionally, so the total is eight.
- With the basic logic modeling (made from two *xor*s, two *and*s, and one *or*) there are 17 clauses, 14 of them having three, and three having two literals. This sums up to 50 literals in total. Three variables (c_1, c_2, and s_1) have to be introduced additionally, so the total is eight.

From these numbers it is possible to make a decision on which approach to use. In general, it can be assumed: as lower the numbers are, as smaller and "*better*" is the encoding. The half-adder approach can be left out of consideration, as it scores worst. While the number of literals is almost half way between those of the full-adder and basic logic approach, its number of variables is the same as the higher one of the other two, and its

number of clauses is even the highest of all. The full-adder approach scores best in two categories: clauses and variables. The basic logic approach scores best only in a single category: literals. However, this category provides the largest reduction in encoding size.

It has been decided to use the basic logic approach.

3.2.7.2. Subtraction

Subtractions can be derived from the handling of additions. The chosen handling differs from the actual machine level implementation, which also reuses addition, but in a different way.

The subtraction:

$a - b = c$

Is the same as the addition:

$c + b = a$

It does not matter that one of the inputs is unknown, but the result is known. The constraints stating that $c + b = a$ enforce this bidirectionally. This is quite similar to the concepts of functional programming languages. Actually the words *result* and *input* are not adequate as they imply a unidirectionality.

3.2.7.3. Multiplication

A multiplication can be handled by doing a number of additions.

$a \times b$ is equivalent to	or interchangeably
$\sum_{i=1}^{b} a$	$\sum_{i=1}^{a} b$
(adding a b-times to itself)	(adding b a-times to itself)

However, this should only be done with at least one of the factors a and b being a rather small number. For larger numbers this approach would be inefficient, replacing the multiplication by far too many additions. It is therefore not generally used.

The method generally used in circuits is similar to the written multiplication. The written multiplication is broadly applied for manual calculations in the decimal system using pen and paper.

12345×67890

can be calculated as:

$$a \times b = \sum_{i=0}^{n-1} a \times b_i \times 10^i$$

where n is the number of digits of b, and b_i is a digit of b at position i from the right.

For the given example this would be:

$12345 \times 0 \times 10^0$

$+12345 \times 9 \times 10^1$

$+12345 \times 8 \times 10^2$

$+12345 \times 7 \times 10^3$

$+12345 \times 6 \times 10^4$

The same approach can also be used with binary numbers, using powers of 2 instead of 10 as factor.

With numbers in the binary system this is even simpler. Each digit b_i of b, which a is to be multiplied with, can either be 0 or 1. Therefore, this multiplication with a single digit number is the same as doing an *and*-operation.

If $b_i = 1$, then $a \times b_i = a$, else $a \times b_i = 0$.

Before adding the intermediate results, each of them is shifted i digits to the left, just as in the written multiplication in the decimal system.

Special handling of negative values is not necessary. Operations with Integers representing negative values will also give a correct result using this approach.

3.2.7.4. Division

Division can not be replaced as directly by multiplication as subtraction can be by addition. The equivalency of

$a \div b = c$

and

$b \times c = a$

is not in all cases true for Integer values.

To make this replacement work, the remainder has to be used in both multiplication and division formulas.

$(a - r) \div b = c$

$b \times c = a - r$

$b \times c + r = a$

may $0 \leq r < |b|$ for $a > 0$; $|b| < r \leq 0$ for $a < 0$; $r = 0$ for $a = 0$

So a division can be replaced by transforming it into a multiplication when the remainder r is introduced as a new temporary variable. The previously shown encoding of multiplications can therefore be reused along with the as well established encoding for the addition.

The sign of the remainder is the same as the sign of the dividend according to C (ISO 1999) [ISO07]. With C (ISO 1990) it was implementation defined, so results might be different.

r has to be restricted to be in the range between including 0 and excluding $|b|$ by appropriate constraints. Correspondingly q has to be restricted to be in the range between

including 0 and including a (it might otherwise take values exceeding this range, which are valid due to integer overflow).

There is no need to avoid q taking the value of zero. Quite the contrary, it is endorsed when a division by zero gets revealed executing the derived test cases. The restricted ranges for r and q can be seen in figure 3.7

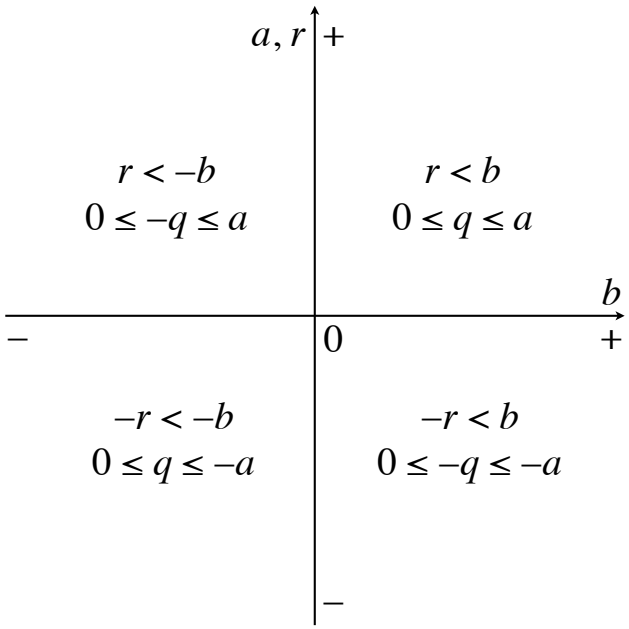


Figure 3.7.: Ranges of remainder r and quotient q for encoding of division and modulo operation.

3.2.7.5. Modulo

With the approach shown for division, it is possible to do modulo computation as well. Instead of the value c, the result in this case is the value r.

3.2.8. Comparisons

Along with logical operations, comparisons are a major component of *if*-statements. They are important for being the basis of conditions and decisions. Two values of variables are being compared, and a Boolean value is given as result. In C, as in most languages, there are several comparison operations:

- less
- less or equal
- equal
- greater or equal
- greater
- not equal

When two values x and y are compared, this is mathematically the same as comparing the difference of x and y to zero:

$x \triangleright y \Leftrightarrow x - y \triangleright 0$

($\triangleright$ standing for any one of the comparisons types $<$, $\leq$, $=$, $\geq$, $>$, or $\neq$).

For integer values this is however not always true, as an overflow might happen. This problem will be addressed later in this section.

How to model a subtraction has previously been discussed, so it can be used here. With this the problem of comparing two numbers to each other is reduced to comparing one number to zero.

It can be differentiated between three types of values the difference of x and y can have.

(1.) $x - y < 0$

(2.) $x - y = 0$

(3.) $x - y > 0$

The three cases can easily be checked for:

(1.) The leftmost bit is 1

(2.) All bits are 0

(3.) By elimination: (1.) and (2.) do not apply, the results of checking them are false: $(\neg(1.) \wedge \neg(2.)) \leftrightarrow (3.)$

These results can not only be used to determine *less*, *equal*, or *greater* comparisons. *Less or equal*, *greater or equal*, and *not equal* can be derived by combinations of the above.

- less or equal: $(1.) \vee (2.)$
- greater or equal: $(3.) \vee (2.)$
- not equal (less or greater): $\neg(2.) \leftrightarrow ((1.) \vee (3.))$

Therefore, all necessary options to compare values in C are being covered by this solution.

It is necessary to use signed types, as it must be possible for the difference of values to be negative. Unsigned variables of bitlength n have to be transformed to signed variables of bitlength $n + 1$ to use this approach. This is basically done by putting an additional zero in front of the leftmost bit.

Signed variables are in risk of giving wrong results, caused by an overflow in the calculation of the subtraction. Here also an increase in bitlength by one additional bit appended left to the leftmost bit solves the problem. The added bit has to be a zero for positive and a 1 for negative values.

In general signed and usigned variables as well have to be converted to a signed type of a bitlength increased by one. Conversions are to be applied as discussed in section 3.2.5.

3.2.9. Conditional Behavior

The main purpose of an implementation of conditional behavior is the correct handling of conditional assignments. Conditional assignment means, that inside an *if*- or an *else*-branch a variable gets assigned a new value. The value the variable has after the *if*-block is therefore dependent on which path has been chosen.

A conditional assignment is given in the following example:

```
1 if ( ... ) {
2   // d_super
3   if ( a > b ) { // d
4     // d_if
5     a = b;
```

```
6     } else {
7       // d_else
8       b = a;
9     }
10  }
```

May d be a Boolean variable holding the true or false result of a decision.

When the *if*-statement is nested, the outcome of the respective previous decision has to be considered. The information whether the *if*-branch has been chosen in that previous decision is given in the Boolean variable d_{super}.

$d_{if} = d \wedge d_{super}$

$d_{else} = \neg d \wedge d_{super}$

Any clause within the *if*-branch is supplemented with the literal $\neg d_{if}$, those in the else branch with $\neg d_{else}$. This will lead to the clauses only enforcing something when the condition is decided accordingly.

Example for a conditional assignment operation in the *if*-branch:

$d_{if} \rightarrow (a_{new} \leftrightarrow b_{old})$

The truth table for this assignment and the derived clauses are given in table 3.15.

This, however, would leave the new value of a to be undefined, when the condition is not evaluated as *true*. Therefore, the equivalency of a with its previous value a_{old} has to be assured when the condition is evaluated as being *false*:

$\neg d_{if} \rightarrow (a_{new} \leftrightarrow a_{old})$

For the else branch corresponding conditional assignments exist:

$d_{else} \rightarrow (b_{new} \leftrightarrow a_{old})$

$\neg d_{else} \rightarrow (b_{new} \leftrightarrow b_{old})$

d_{if}	a_{new}	b_{old}	Valid	Clauses
0	0	0	1	
0	0	1	1	
0	1	0	1	
0	1	1	1	
1	0	0	1	
1	0	1	0	$\neg d_{if} \vee a_{new} \vee \neg b_{old}$
1	1	0	0	$\neg d_{if} \vee \neg a_{new} \vee b_{old}$
1	1	1	1	

Table 3.15.: Modeling a conditional assignment

3.2.10. Loops

Loops can not directly be encoded using Boolean constraints. To handle them, loops must be unwound, i.e. be replaced by a corresponding set of *if*-statements [Kro09, p. 514-516].

In C "`for`", "`while`", and "`do while`" are the keywords for explicit loop statements. Implicit loops created by backwards jump statements or recursion could, in principle, be handled the same way as explicit loops. However, these implicit loops are much harder to recognize.

The number of *if*-statements used to replace a loop (the depth of the unwinding) limits the maximal possible amount of loop iterations. An arbitrary high number of iterations can therefore not be modeled.

The following *while*-loop shall be used as an example:

```
1 while ( ... ) {
2   ...
3 }
```

Every *do-while*- or *for*-loop can be replaced by a semantically equivalent *while*-loop. The shown handling of a *while*-loop can correspondingly be applied to *do-while*- and *for*-loops as well.

The example *while*-loop can be replaced by the following set of *if*-statements:

```
1 if ( ... ) {
```

```
2    ...
3    if ( ... ) {
4      ...
5      if ( ... ) {
6        ...
7        if ( ... ) {
8          ...
9        }
10     }
11   }
12 }
```

The last *if*-statement has to be enforced to be false to prevent overrunning the limited number of *if*-statements. The maximum number of iterations for this example is four.

Annotations added make the introduced *if*-statements distinctive from regular *if*-statements.

```
1  if ( ... ) {
2    ...
3    /*loop*/if ( ... ) {
4      ...
5      /*loop*/if ( ... ) {
6        ...
7        /*loop*/if ( ... ) {
8          ...
9          /*loop-end*/if ( ... ) {
10         }
11       }
12     }
13   }
14 }
```

Adding the annotation within comment delimiters prevents ambiguity with potentially existing functions "`loopif()`" or "`loop-endif()`" of same name. "`/*loop*/if`" would be interpreted by the parser basically the same way as "`if`". It would, however, not add to the condition and decisions listing, and therefore would be ignored in the application of coverage criteria. The result of "`/*loop-end*/if`" would be enforced to be false.

3.3. Control Flow Coverage

The Boolean variables containing information on condition and decision outcomes are provided by the previous modeling of basic operations. For achieving one of the types of control flow coverage, there is the need to have as many copies of the previous modeling as the number of wanted test cases. The constraints applied for the coverage types work across those test cases (e.g. a condition or decision has to be *true* in test case 1 and *false* in test case 2).

3.3.1. Condition Coverage

A scheme of unique numbers can be applied to conditions of the entire model.

Condition enumeration scheme:

$c_{<DecisionNo.>,<ConditionNo.withinDecision>,<TestCaseNo.>}$

Where *decision no.* enumerates all decisions over the whole model, *condition no.* enumerates the conditions within one decision, and *test case no.* identifies which test case it belongs to. All enumerations starting with 1.

Example:

$c_{1,1,1}, c_{1,2,1}, c_{1,3,1}, c_{1,4,1}$

$c_{1,1,2}, c_{1,2,2}, c_{1,3,2}, c_{1,4,2}$

$c_{1,1,3}, c_{1,2,3}, c_{1,3,3}, c_{1,4,3}$

Four conditions of one decision over three test cases.

Conditions have to be already set in relation to their corresponding higher decision d_{super} (the decision one step up from the decision containing them). This is to ensure the conditions to be within the path to be executed. Without this the conditions might not be covered by a derived test case, as the alternate path might be taken.

$c_{true} = c \wedge d_{super}$

$c_{false} = \neg c \wedge d_{super}$

To achieve condition coverage for each condition there must be at least one outcome *true* and at least one outcome *false* across all test cases.

For the previously shown example this leads to:

$atleast(1, \{c_{true1,1,1}, c_{true1,1,2}, c_{true1,1,3}\})$

$atleast(1, \{c_{false1,1,1}, c_{false1,1,2}, c_{false1,1,3}\})$

$atleast(1, \{c_{true1,2,1}, c_{true1,2,2}, c_{true1,2,3}\})$

$atleast(1, \{c_{false1,2,1}, c_{false1,2,2}, c_{false1,2,3}\})$

$atleast(1, \{c_{true1,3,1}, c_{true1,3,2}, c_{true1,3,3}\})$

$atleast(1, \{c_{false1,3,1}, c_{false1,3,2}, c_{false1,3,3}\})$

$atleast(1, \{c_{true1,4,1}, c_{true1,4,2}, c_{true1,4,3}\})$

$atleast(1, \{c_{false1,4,1}, c_{false1,4,2}, c_{false1,4,3}\})$

The cardinality constraints are used for better readability. As they constrain to "at least one" they can be expressed in a single clause as well.

3.3.2. Decision Coverage

As for conditions a corresponding scheme of unique numbers can also be applied to decisions of the entire model.

Decision enumeration scheme:

$d_{<DecisionNo.>,<TestCaseNo.>}$

Where *decision no.* enumerates all decisions over the whole model, and *test case no.* identifies which test case it belongs to. Both enumerations starting with 1.

Example:

$d_{1,1}, d_{1,2}, d_{1,3}$

One decision over three test cases.

Decisions have to be already set in relation to their corresponding higher decision d_{super}, so $d_{if<DecisionNo.>,<TestCaseNo.>}$ and $d_{else<DecisionNo.>,<TestCaseNo.>}$ are available as the decisions for either path.

To achieve decision coverage for each decision there must be at least once chosen the *if* path and at least once chosen the *else* path across all test cases.

For the previously shown example this leads to:

$atleast(1, \{d_{if1,1}, d_{if1,2}, d_{if1,3}\})$

$atleast(1, \{d_{else1,1}, d_{else1,2}, d_{else1,3}\})$

3.3.3. Condition/Decision Coverage

With condition coverage and decision coverage given, condition/decision coverage can be applied straight forward. The derived constraints of both of the previously discussed coverages are to be applied simultaneously. Additional constraints are not necessary.

3.3.4. Modified Condition/Decision Coverage

The modified condition/decision coverage takes over all constraints, which have to be applied for condition/decision coverage. A set of constraints has to be applied to enforce the additional rule that each condition must be able to individually effect the decision it is contained in.

These additional constraints are by far the most complex constraints applied for any of the discussed coverage types, as it can be seen in the example below.

The example below has been simplified. Each expression $(c_{...} \leftrightarrow c_{...})$ would actually need to be replaced by $((c_{true...} \leftrightarrow c_{true...}) \wedge (c_{false...} \leftrightarrow c_{false...}))$, and correspondingly each

expression $(d_{...} \leftrightarrow d_{...})$ by $((d_{if...} \leftrightarrow d_{if...}) \wedge (d_{else...} \leftrightarrow d_{else...}))$. The full formula is given in appendix F.

$$
\begin{aligned}
&\{ \\
&\quad [\\
&\qquad \neg(c_{1,1,1} \leftrightarrow c_{1,1,2}) \wedge (c_{1,2,1} \leftrightarrow c_{1,2,2}) \wedge (c_{1,3,1} \leftrightarrow c_{1,3,2}) \wedge (c_{1,4,1} \leftrightarrow c_{1,4,2}) \\
&\qquad \wedge \neg(d_{1,1} \leftrightarrow d_{1,2}) \\
&\quad] \vee [\\
&\qquad \neg(c_{1,1,1} \leftrightarrow c_{1,1,3}) \wedge (c_{1,2,1} \leftrightarrow c_{1,2,3}) \wedge (c_{1,3,1} \leftrightarrow c_{1,3,3}) \wedge (c_{1,4,1} \leftrightarrow c_{1,4,3}) \\
&\qquad \wedge \neg(d_{1,1} \leftrightarrow d_{1,3}) \\
&\quad] \vee [\\
&\qquad \neg(c_{1,1,2} \leftrightarrow c_{1,1,3}) \wedge (c_{1,2,2} \leftrightarrow c_{1,2,3}) \wedge (c_{1,3,2} \leftrightarrow c_{1,3,3}) \wedge (c_{1,4,2} \leftrightarrow c_{1,4,3}) \\
&\qquad \wedge \neg(d_{1,2} \leftrightarrow d_{1,3}) \\
&\quad] \\
&\} \wedge \{ \\
&\quad [\\
&\qquad (c_{1,1,1} \leftrightarrow c_{1,1,2}) \wedge \neg(c_{1,2,1} \leftrightarrow c_{1,2,2}) \wedge (c_{1,3,1} \leftrightarrow c_{1,3,2}) \wedge (c_{1,4,1} \leftrightarrow c_{1,4,2}) \\
&\qquad \wedge \neg(d_{1,1} \leftrightarrow d_{1,2}) \\
&\quad] \vee [\\
&\qquad (c_{1,1,1} \leftrightarrow c_{1,1,3}) \wedge \neg(c_{1,2,1} \leftrightarrow c_{1,2,3}) \wedge (c_{1,3,1} \leftrightarrow c_{1,3,3}) \wedge (c_{1,4,1} \leftrightarrow c_{1,4,3}) \\
&\qquad \wedge \neg(d_{1,1} \leftrightarrow d_{1,3}) \\
&\quad] \vee [\\
&\qquad (c_{1,1,2} \leftrightarrow c_{1,1,3}) \wedge \neg(c_{1,2,2} \leftrightarrow c_{1,2,3}) \wedge (c_{1,3,2} \leftrightarrow c_{1,3,3}) \wedge (c_{1,4,2} \leftrightarrow c_{1,4,3}) \\
&\qquad \wedge \neg(d_{1,2} \leftrightarrow d_{1,3}) \\
&\quad] \\
&\} \wedge \{ \\
&\quad [\\
&\qquad (c_{1,1,1} \leftrightarrow c_{1,1,2}) \wedge (c_{1,2,1} \leftrightarrow c_{1,2,2}) \wedge \neg(c_{1,3,1} \leftrightarrow c_{1,3,2}) \wedge (c_{1,4,1} \leftrightarrow c_{1,4,2}) \\
&\qquad \wedge \neg(d_{1,1} \leftrightarrow d_{1,2}) \\
&\quad] \vee [\\
&\qquad (c_{1,1,1} \leftrightarrow c_{1,1,3}) \wedge (c_{1,2,1} \leftrightarrow c_{1,2,3}) \wedge \neg(c_{1,3,1} \leftrightarrow c_{1,3,3}) \wedge (c_{1,4,1} \leftrightarrow c_{1,4,3})
\end{aligned}
$$

$$
\begin{aligned}
&\quad \wedge\neg(d_{1,1} \leftrightarrow d_{1,3}) \\
&\quad] \vee [\\
&\quad (c_{1,1,2} \leftrightarrow c_{1,1,3}) \wedge (c_{1,2,2} \leftrightarrow c_{1,2,3}) \wedge \neg(c_{1,3,2} \leftrightarrow c_{1,3,3}) \wedge (c_{1,4,2} \leftrightarrow c_{1,4,3}) \\
&\quad \wedge\neg(d_{1,2} \leftrightarrow d_{1,3}) \\
&\quad] \\
\} \wedge \{ \\
&\quad [\\
&\quad (c_{1,1,1} \leftrightarrow c_{1,1,2}) \wedge (c_{1,2,1} \leftrightarrow c_{1,2,2}) \wedge (c_{1,3,1} \leftrightarrow c_{1,3,2}) \wedge \neg(c_{1,4,1} \leftrightarrow c_{1,4,2}) \\
&\quad \wedge\neg(d_{1,1} \leftrightarrow d_{1,2}) \\
&\quad] \vee [\\
&\quad (c_{1,1,1} \leftrightarrow c_{1,1,3}) \wedge (c_{1,2,1} \leftrightarrow c_{1,2,3}) \wedge (c_{1,3,1} \leftrightarrow c_{1,3,3}) \wedge \neg(c_{1,4,1} \leftrightarrow c_{1,4,3}) \\
&\quad \wedge\neg(d_{1,1} \leftrightarrow d_{1,3}) \\
&\quad] \vee [\\
&\quad (c_{1,1,2} \leftrightarrow c_{1,1,3}) \wedge (c_{1,2,2} \leftrightarrow c_{1,2,3}) \wedge (c_{1,3,2} \leftrightarrow c_{1,3,3}) \wedge \neg(c_{1,4,2} \leftrightarrow c_{1,4,3}) \\
&\quad \wedge\neg(d_{1,2} \leftrightarrow d_{1,3}) \\
&\quad] \\
\}
\end{aligned}
$$

Every section in curly brackets takes care of one condition, effecting the decision individually. The condition taken care of is the one whose equality operation in round brackets is negated.

Every section in square brackets handles one possible combination across the number of test cases (1 and 2, 1 and 3, 2 and 3). Any combination of two test cases can be the one combination showing the individual effect of a condition on a decision. The number of combinations and therefore the number of sections in square brackets growth exponentially with the number of test cases. But as the target is a minimal number of test cases this should not be a big problem.

In the round brackets, the corresponding conditions and decisions are connected with a logical equality operation. A single pair of conditions and the decision are set to be not equal (thereby enforcing individual influence of the condition on the decision).

IV. Implementation

Two command line applications have been implemented, one for the encoding and one for the decoding functionality. The SAT-solver used is one of the existing SAT-solvers, so no own implementation was necessary for this step. The SAT-solver's in- and output formats, however, take an important role in implementation. They are provided by the encoder, respectively used by the decoder.

The encoder as well as the decoder are implemented using the Ruby programming language. There is no exceptional requirement on the programming language to be used for the implementation. It has to be object oriented and it has to support hash tables.

Ruby has been designed to follow the principle of "least surprise". This philosophy is an advantage when developing a prototype application. It allows the programmer to focus on the application to be developed even when doing something new, rather than worrying about how things work in the specific programming language.

`ruby-lang.org`, a website advocating and supporting usage of the Ruby programming language explains Ruby the following way:

> "Ruby is a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write" [rub13c].

Ruby includes influences from Ada, C++, CLU, Dylan, Eiffel, Lisp, Perl, Python, and Smalltalk [Bin07, p. 3; Coo09, p. 101; rub13a].

It is one of the top-ten programming languages[1] according to the TIOBE Programming Community Index for August 2013 [Tio13b].

It is an interpreted language, to run it a Ruby interpreter is therefore required to be installed. Using an interpreted language for encoder and decoder is not performance critical. The solving, which is computationally the "hardest" step of the overall workflow, is not done with Ruby, but with an highly efficient SAT-solver.

Ruby is pre-installed on Mac OS X (v10.4 and up) and many Linux distributions. It is also available for Windows. To check whether Ruby is already installed or not the following can be entered in command line:

```
1 ruby -v
```

With Ruby available on the system, information on the currently installed version will be output. Otherwise the system will give some error message stating that the command "ruby" does not exist.

Ruby version 1.8.7 was used for development along this thesis. Using a previous version of the Ruby interpreter might work fine, but please note that the code has not been tested for downward compatibility. Using a later version should in general be unproblematic.

If installation or update is necessary, it can be done according to the download information on `ruby-lang.org` [rub13b], or the "installing ruby" section of wikibooks on ruby programming [Wik04]. For Windows there is a package available on `rubyinstaller.org`, "a self-contained Windows-based installer that includes the Ruby language, an execution environment, important documentation, and more" [BBB+13].

A detailed documentation of all classes and functions of encoder and decoder is provided with the program code. When it is not available it can be produced using RDoc [rdo13] which is part of the Ruby bundle. The following command is to be used:

```
1 rdoc ./src
```

[1] Java, C, C++, Objective-C, PHP, C#, (Visual) Basic, Python, JavaScript, and Ruby

A small ruby script to run the three steps of the workflow (encoder, solver, and decoder) is also included. It is intended for unixoid Systems as it is running system commands. An adaption to support Windows would be possible.

4.1. Implementation of the Encoder

The encoder has been structured into major components:

- main; file: "`coverage-sat-encoder.rb`"
- class models
  - condition/decision class model;
    file: "`coverage-sat-encoder/condition_decision_classesl.rb`"
  - variable class model;
    file: "`coverage-sat-encoder/variable_classes.rb`"
- assistance functions
  - global data; file: "`coverage-sat-encoder/global.rb`"
  - initializations; file: "`coverage-sat-encoder/initializations.rb`"
- workflow functions
  - input and preprocessing;
    file: "`coverage-sat-encoder/input_preprocessing.rb`"
  - parsing; file: "`coverage-sat-encoder/parsing.rb`"
  - encoding
    * general encoding; file: "`coverage-sat-encoder/encoding.rb`"
    * encoding of coverage;
      file: "`coverage-sat-encoder/encoding-coverage.rb`"
  - DIMACS-output; file: "`coverage-sat-encoder/dimacs-output.rb`"

When it comes to the points of tokenization and parsing, the aim is to serve the specific purpose and enable the encoding. It is not necessary to use the most modern and efficient way of implementation, as it could for example be found in a state of the art interpreter. The software developed here is a prototype application, the tokenizer and parser components are required for it to work, but they are not the main focus of it.

The visualization of the encoder components and their relationships can be seen in figure 4.1. Each of them will be explained in greater detail in the following sections.

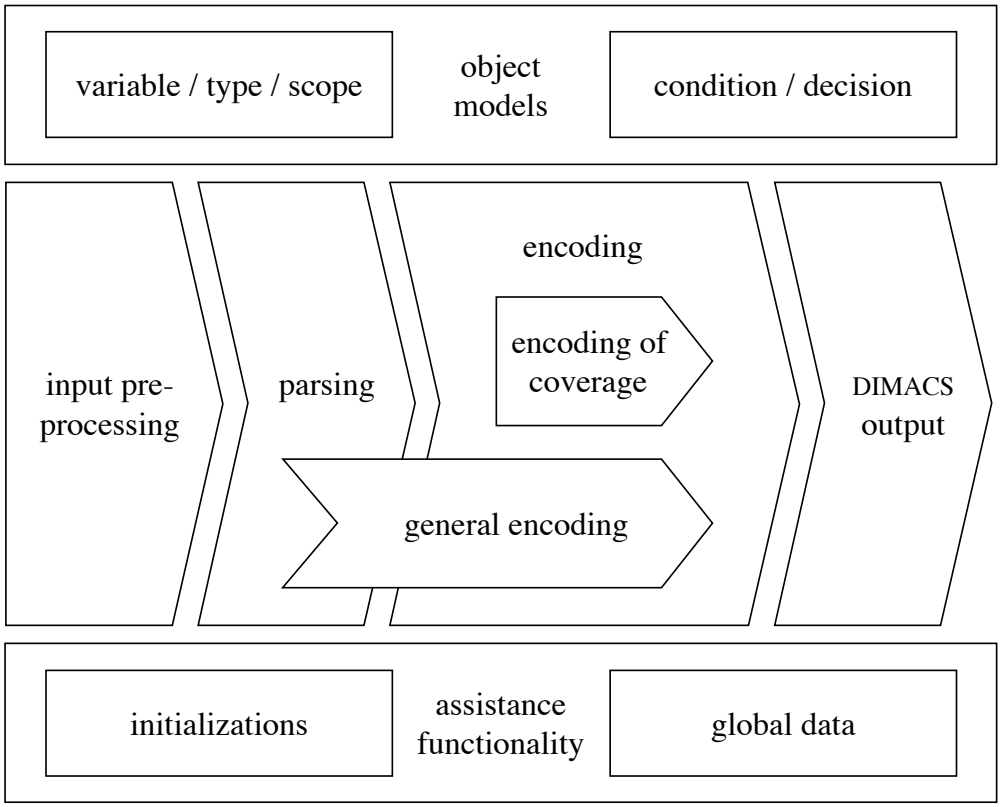


Figure 4.1.: Major components of the encoder.

In the middle of the figure the major workflow components can be seen. They are ordered from left to right. The input first gets read in and preprocessed. The relevant tokenized part of code gets handed to the parsing component. The general encoding for each parsed statement is done in parallel while parsing it. After parsing is finished, encoding continues. Encoding of the coverage is added, which itself makes use of the general encoding functionality. When the encoding is finished, the DIMACS output file can be generated.

Above the workflow functions the generated object models can be seen. This contains related objects for variables, types, and scopes, as well as related objects for conditions and decisions. A generated object remains accessible to each workflow function until the end of execution.

Below the workflow functions – correspondingly to the object model above – non-object-oriented assistance functionality can be seen. It can be differentiated into global data (for example counters like for the number of SAT variables used can be found there) and functions which are used for initialization.

4.1.1. External Behavior

The external behavior of the encoder is defined by its in- and outputs. The in- and outputs of the encoder, along with their sources, respectively recipients, can be seen in figure 4.2.

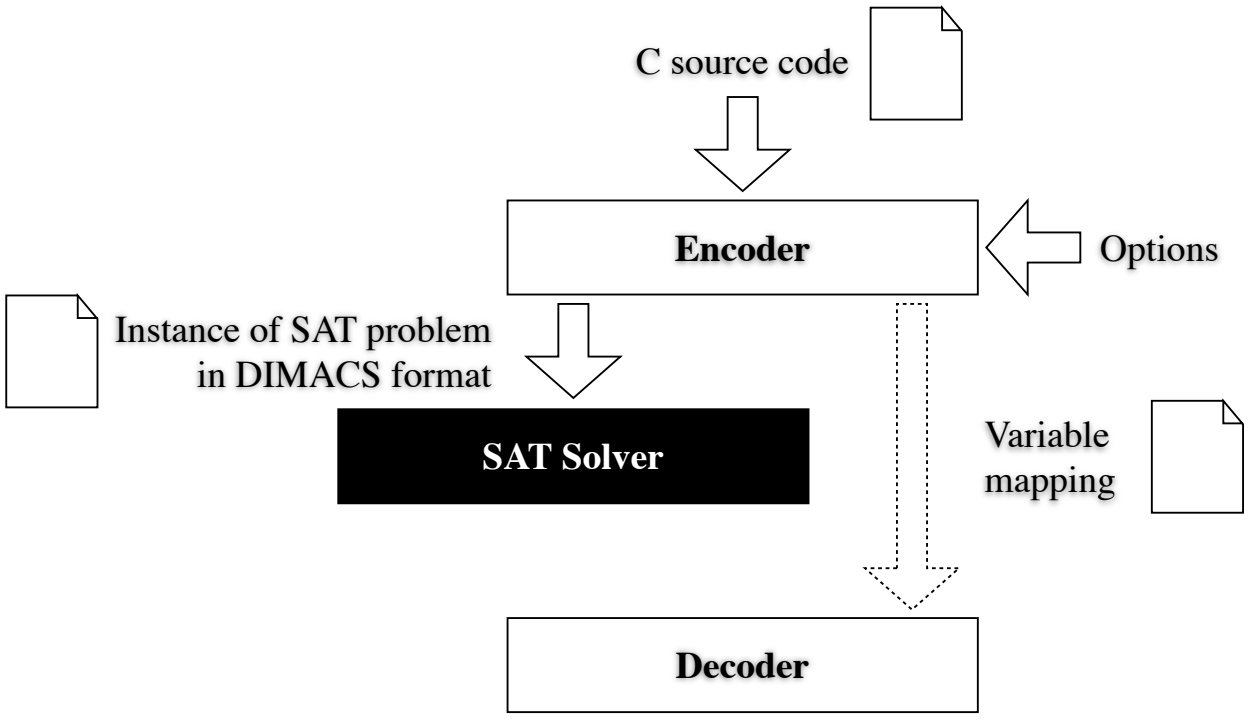


Figure 4.2.: In- and output of the encoder

4.1.1.1. Input

The encoder requires the following information as input:

- source file name, possibly including path (if in different directory)
- function signature of format "`function_name(parameter_1_type, paramameter_2_type, ...)`", identifying an individual function from the source file to be used. Spaces can be placed freely.
- coverage type to be encoded, named by one of the following short identifiers (not case sensitive):
  - "`cc`" for condition coverage
  - "`dc`" for decision coverage
  - "`cdc`" for condition/decision coverage
  - "`mcdc`" for modified condition/decision coverage
- number of test cases: positive natural number
- filename for the output file (optional; default: "output.cnf")
- configuration of variable types according to compiler/system to be used
  - character type has to be defined to be either signed or unsigned
  - bitlength for the types:
    * "`char`"
    * "`short`"
    * "`int`"
    * "`long`"
    * "`long long`"

The settings expected to change frequently when calling the encoder (source file name / path, function signature, coverage type, number of test cases, and output filename) are to be given as command line arguments.

The following example shows an invocation of the encoder with those command line arguments:

```
ruby coverage-sat-encoder.rb example.c "do_something(int, unsigned long
     int, int)" mcdc 2 "test.cnf"
```

The settings expected to remain unchanged for most of the time (configuration of the properties of variable types) are set in a configuration file. YAML (YAML Ain't Markup Language / Yet Another Markup Language) is used as format for this config file. "YAML is a human friendly data serialization standard for all programming languages" [Eva13]. It was chosen as its content can be most easily read into objects through a library function.

The file "type-config.yaml" has the following format:

```
# Variable Type Specification
# for CoverageSAT Encoder v1 2013
# by Stefan Weiler
# Darmstadt University of Applied Sciences
#  / University of Wisconsin-Platteville
---
- !ruby/object:TypeSpec
  char_is_signed:      false
  char_bitlength:      8  # Min. 8
  short_bitlength:     16 # Min. 16
  int_bitlength:       16 # Min. 16
  long_bitlength:      32 # Min. 32
  long_long_bitlength: 64 # Min. 64
```

It defines an object "TypeSpec" with its respective properties.

When read in, a warning will be given, when the bitlengths are set below their respective minimum bitlengths. A warning will also be given when the bitlength of an expectedly larger type is not equal or greater than the expectedly next smaller type. Despite the warning, such settings will however be accepted.

4.1.1.2. Output

There are two recipients for the output data of the encoder:

- SAT-solver
- decoder

Along with being used by these successive components, the output format also has to be as human readable as possible (for example for debugging).

The output to be used by the SAT-solver has to be in DIMACS format, as described earlier in section 2.2.3 on "SAT Solving".

Information on the specific encoding of a given input is included in the DIMACS formatted output file. The information is included in the comment section of the file, making use of the DIMACS format allowing any content in lines marked with "c" as their first character. A second output file for information on the encoding is therefore not needed. The decoder has to get handed over the output file of the encoder along with the output file of the SAT solver to decode the result.

The structure of an output file can be seen in the following example:

```
1  c Generated with CoverageSAT Encoder v1 2013
2  c by Stefan Weiler
3  c Darmstadt University of Applied Sciences
4  c  / University of Wisconsin-Platteville
5  c
6  c Input:
7  c Source file:   main.c
8  c Function:      example(int,unsignedlongint,int)
9  c Coverage type: mcdc
10 c Test cases:    2
11 c
12 c Mapping:
13 c Parameter mapping:
14 c {w (16bit signed int) -> {1}, x (32bit unsigned int) -> {33}, z (16
       bit signed int) -> {97}}
15 c Variable mapping:
16 c {w (16bit signed int) -> {1}, x (32bit unsigned int) -> {33}, z (16
       bit signed int) -> {97}, b (16bit signed int) -> {129, 193}, 0 (16
       bit signed int) -> {129}, 1 (16bit signed int) -> {161}, 2 (16bit
       signed int) -> {193}, a (16bit signed int) -> {193}}
```

```
17 c
18 c CNF encoding:
19 p cnf 450 610
20 -129 0 -145 0 -130 0 -146 0 -131 0 -147 0 -132 0 -148 0 -133 0 ...
```

The parameter respectively variable mapping has the following format:

```
1 {<VariableIdentifier> (<NumberOfBits>bit <SignedOrUnsigned> <Category>)
      -> {<LeftMostSatVairable>,<LeftMostSatVairable>}}, ...}
```

In the following the variable mapping will be discussed. The information applies correspondingly to the parameter mapping.

The mapping within curly brackets contains a list of variables, with the variables separated by commas. The order of the list is the order in which the variables have been generated. `<VariableIdentifier>` gives the name of the variable, which is either the name of the variable used in source code, or a consecutive number for auxiliary variables. After the identifier, a definition of the variables characteristics is given in round brackets. `<NumberOfBits>` gives the bitlength of the variable. `<SignedOrUnsigned>` defines wether it the variable is of a "`signed`" or an "`unsigned`" type. `<Category>` is either "`int`" or "`char`", it defines whether the value should be interpreted as a natural number or as ASCII encoding of a character.

After the arrow ("->") the mapped SAT variables are specified in a list surrounded by curly brackets and separated by commas. `<LeftMostSatVariable>` is the SAT variable representing the highest bit of a value, to its right follow the remaining bit, and the values for the remaining test cases. As a variable can take several values, several `<LeftMostSatVariable>` can be found. They are ordered left to right with the leftmost of them being the first assigned value, and the rightmost being the last assigned value.

In contrast to the variable mapping, the list of SAT variables in the parameter mapping contains only one element. All parameters are also included in the variable mapping. The parameter mapping is intended to ease the identification of the parameters amongst the other variables.

After the "comment" section, the actual encoding of clauses as discussed in section 2.2.3 follows. The encoded clauses are here not given in full length.

Further output (information, warnings, error messages) may be given.

4.1.2. Class Model

While most of the implementation has been done by functions and basic data types, object-oriented representation has been used to model data and functionality where necessary.

This includes:

- variable, type, and scope;
- condition, decision, and determination; and
- types specification.

4.1.2.1. Variable, Type, and Scope Class Model

An object of type *Variable* represents a program-variable. A variable has an *identifier* of type String, which is either the variables name used in the source code for regular variables, or a unique number for auxiliary variables introduced additionally. An array of integer values is used for the mapping of the program-variable's successive values to SAT-variables. Each integer in the array is the number of the SAT-variable representing the leftmost bit of a value. The most recent mapping comes last in the array. New mappings get appended at the end. An empty mapping can be seen for uninitialized variables.

An existing SAT-variable (named by an integer) can be assigned as newest mapping to a Variable object using *assign_sat*. The latest mapping can be retrieved with *get_sat*. A new mapping can be added with *new_sat*, also returning the naming of the new SAT-variables as integer. The *inspect* method returns a string with human-readable information on the

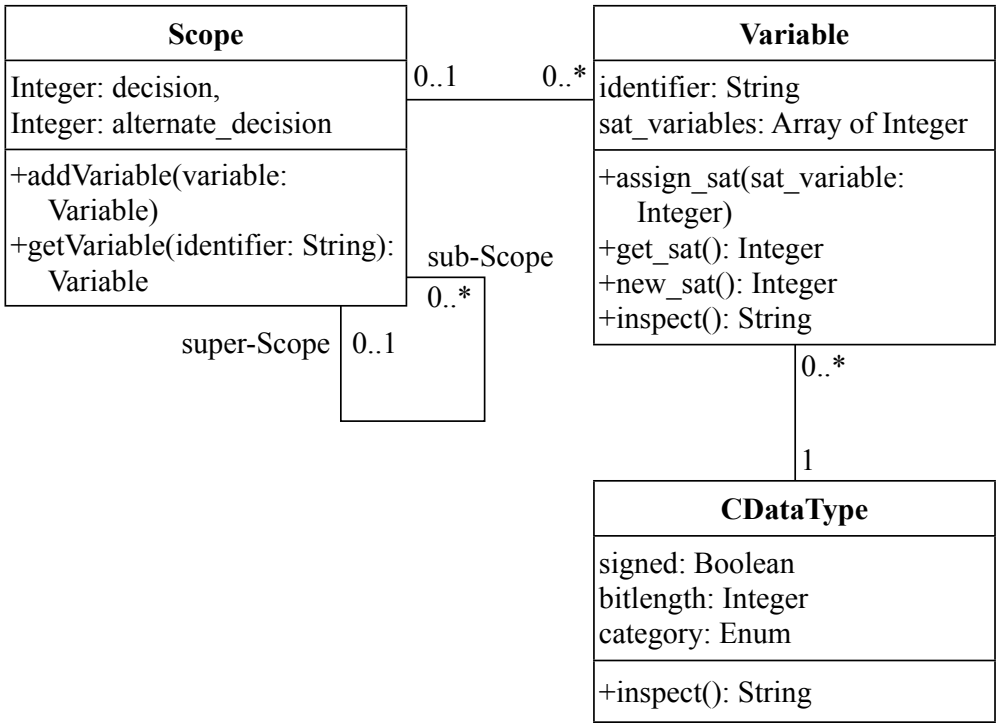


Figure 4.3.: Class diagram of variables, types, and scopes.

objects content. It is used for the output of the variable mapping.

CDataType represents data types used in C (integer and character types being used). A type is either signed (signed is *true*) or unsigned (signed is *false*). It has a *bitlength* of type integer, representing the number of bits to be used. The *category* can be either character or integer.

The *inspect* method here serves the same purpose as the corresponding method does for the variable.

A variable has a single unchangeable type. An arbitrary number of variables can have the same type.

The *Scope* is primarily concerned with the scoping of the variable, i.e. where in code it is visible. It is here also used to handle conditional parts of code. *decision* is an integer value naming the SAT-variable containing the outcome of the decision under which the code in this scope is to be executed. *alternate_decision* is an integer value naming the SAT-variable containing the decision for the alternative path. When directly in row a scope is

closed, the *else*-keyword is given, and a new scope is opened, the alternate-decision of the previous scope becomes the decision of the new scope. The *decision* and *alternate_decision* are to be "*nil*" when the execution of code shall be unconditional.

Variables can be added to the scope using *addVariable*. A variable corresponding to an identifier string can be retrieved using *getVariable*. This method will also retrieve variables from super-scopes.

Regular variables have a single scope, auxiliary variables do not need a scope. An arbitrary number of variables can have the same scope. All scopes except the uppermost scope have a *super-scope* and are therefore *sub-scopes* of these. The number of sub-scopes a scope can have is arbitrary.

4.1.2.2. Condition/Decision Class Model

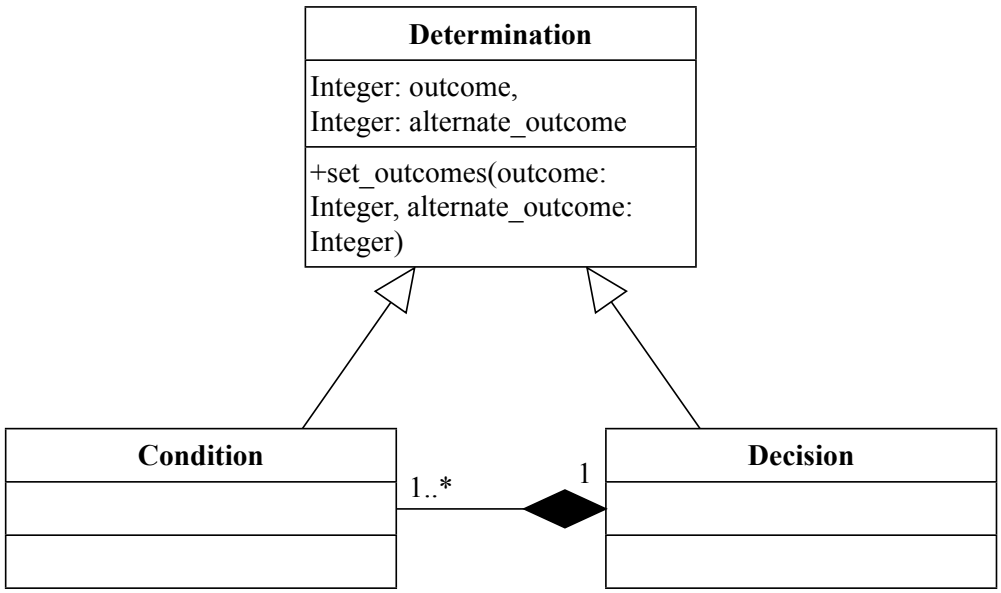


Figure 4.4.: Class diagram of conditions and decisions.

A super-class *Determination* has been introduced, as *Condition* and *Decision* share most of their attributes. Each Determination has an *outcome* and an *alternate_outcome*, both being integers and naming the respective SAT-variable. *outcome* and *alternate_outcome* are to be already related to the super-decision, as discussed in section 3.3. The values can be set using the *set_outcomes* function.

Condition and Decision do not implement further attributes or methods other than theses attributes and methods handling their relation to each other. The main difference is that a Condition is a leaf node of a Decision. This means a Condition always belongs to one single Decision. A Decision on the other hand can have any number of Conditions, yet there must be at least one (composition).

4.1.2.3. Type Specification Class

There is one additional class for the type specification: TypeSpec. It is used for deserialization of type specification information from a YAML file.

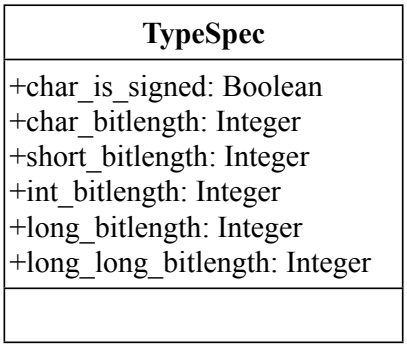


Figure 4.5.: Class diagram of TypeSpec.

4.1.3. Global Data

Several arrangements are possible when ordering the SAT-variables. A good ordering should be both suited to be implemented for the machine to generate and interpret them, and human-readable. Three arrangements which have been considered are shown in figure 4.6

Each square is a bit of a variable represented by a SAT variable. The first number in the index refers to the number of the variable, the second number refers to the number of the test case.

In the first variant a bit of the first test case is always followed by the bits of the following test cases. It is not required to know the bitlength of a variable to find the bits for the

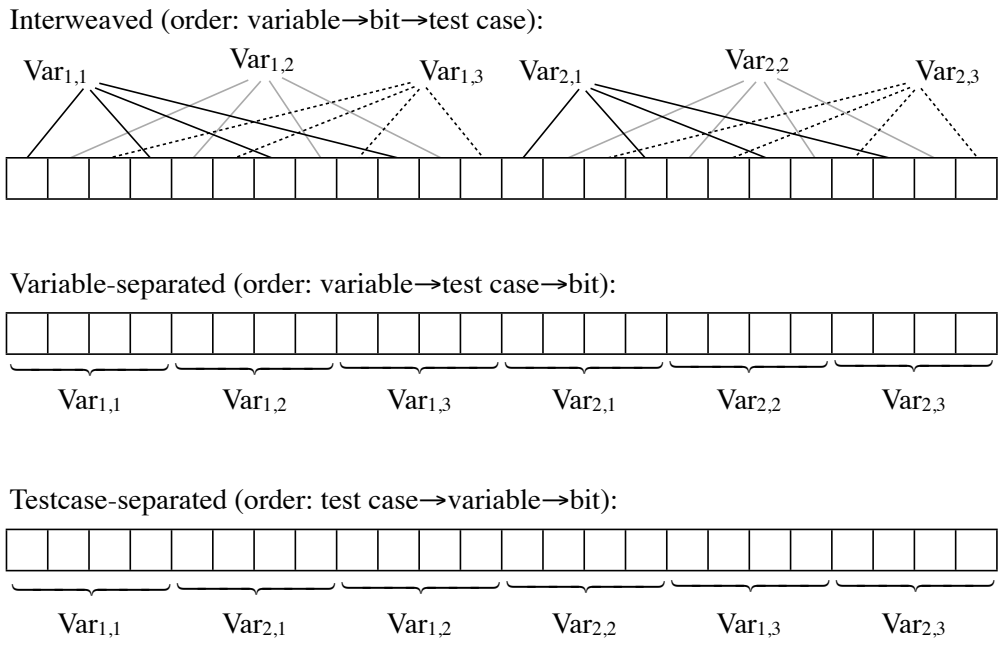


Figure 4.6.: Three possible ways to implement the program-variable to SAT-variable mapping. Example with two variables and three test cases.

second, third, and possibly further test cases. This makes it the easiest arrangement for implementation. On the other hand it has the worst human readability.

The second variant keeps the bits of each variable together and provides a better human readability. The values for the different test cases of a variable are kept together, so the individual test cases are not separated but distributed across the whole mapping. Little additional effort is required for the implementation of this approach. When encoding it always has to be kept track of each variables different length, which influences the position of mapped bits for the second and all following test cases.

In the third arrangement first all variables for the first test case are given, they are followed by all variables of the second test case, then the third and so on. It has a good human readability and a clear separation of the test cases. It however makes it hard to implement, as for the first variable of the second test case to start, it is already required to have information where the last variable of the first test case will end. This makes it impossible to encode all test cases in parallel.

The second approach has been chosen for providing the best combination of rather simple

implementation and human readability.

4.1.4. Input and Preprocessing

Input and preprocessing includes all steps that have to be done before parsing, starting with reading in the source file. The steps of preparation from tokenization, over choosing a function, to reducing to relevant code can be seen in figure 4.7.

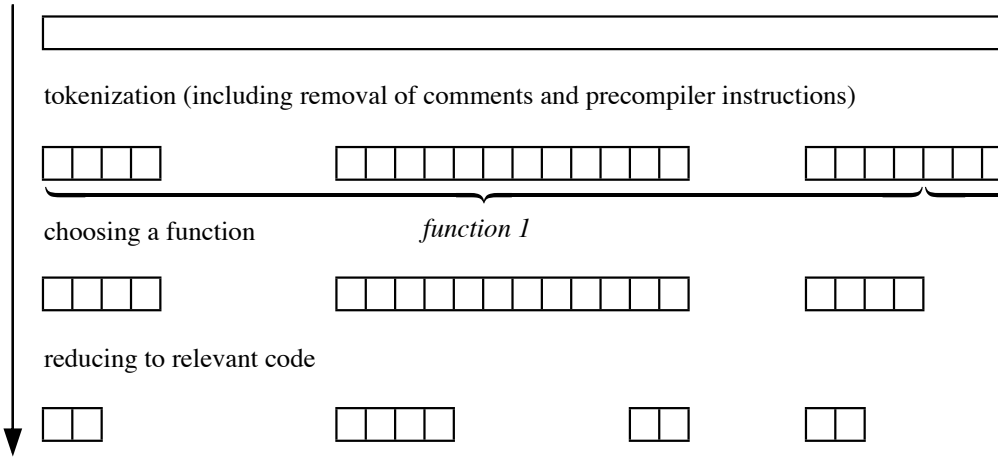


Figure 4.7.: Several steps of code reduction. Illustration of the concept.

The bar in the uppermost row represents the input source code as a whole. This source code is broken down by a tokenizer into its smallest parts – the tokens. These tokens are represented in the second row by the small squares. Only parts containing actual program code (not comments and pre compiler instruction) become part of the list of tokens. This is represented in the figure by gaps, where the not included parts would have been.

From the list of tokens a function is chosen for which test cases shall be derived. With choosing function 1, the list of tokens only has to contain tokens belonging to this function. Tokens of other functions following to the right can be dropped.

The list of tokens of the chosen function is then reduced to the code relevant to encode a type of control flow coverage. The result is a final list of tokens, which is subsequently handed to the parser.

4.1.4.1. Tokenization

The *tokenizer* is used to break down the C source code into smallest segments, the tokens. It is the first step in the handling of the input source code by the encoder.

The tokenizer has been designed to give only those tokens needed by the parser. Segments of the source codes containing comments and precompiler instructions are ignored and do not become part of the output. When precompilation is needed on the source code, it has to be performed externally before handing it to the encoder.

A special focus was given on tokenizing in an intelligent way. This means:

- for operators consisting of multiply characters (comparison operators like "<=", compound assignment operators like "+=", logical operators like "&&") to keep them together as a single token.
- for negative numbers to stay one token as well ("-123", instead of {"-", "123"})

Doing it this way adds a little more work to the tokenizer, but makes the following handling of tokens by the parser simpler.

4.1.4.2. Division and Selection of Functions

A C source file can (and in practice likely will) contain more than just one function. In order to not having the overhead of copying a single function to a separate file, the function that will be used has to be selected automatically from a source file. The list of tokens, which has been derived from the source file, is therefore split into several separate lists of tokens for each function.

To organize the split up functions, and finally being able to choose one of them, a unique identifier is needed. The function that shall be used can on this basis be selected according to a function identifier taken from user input.

A function consists of a function header – in C easily to be recognized as the part before the curly brackets open – and a function body – the remaining part containing the

instructions.

The following shows the structure of function header and body:

Function header:

```
return_type function_name(parameter_1_type parameter_1_name,
    parameter_2_type parameter_2_name, ... )
```

Function body:

```
{
   \\ do something
}
```

From the function header a function signature can be derived. A function signature is a unique identifier of a function. For this purpose, the return type, and the parameter names do not need to be included, as they are not required for unique identification.

From the function header given above, the corresponding function signature can be derived as:

```
function_name(parameter_1_type,parameter_2_type ... )
```

The functions will be stored in an hash table. The function signature is the key. The complete function (also including the header) is the value.

The function to be used will be chosen based on user input. Whitespace in the function signature given by the user will be ignored. This is done to avoid problems for example with either set or not set spaces after a comma in the parameter list.

4.1.4.3. Reduction

The list of tokens can be reduced based on the rules derived in the chapter on the approach, section 3.1.

The steps of reduction, which have been shown there in example 1 and 2, have been fully implemented. As of now, the more advanced reduction shown in example 3 and 4 have

not been implemented. This removes the majority of tokens non-relevant code, only a small amount is left.

The output is a list of tokens shorter or equal in length compared to the list provided by the tokenizer. It is generally expected to be shorter.

4.1.5. Parsing

The list of tokens is then analyzed according to a grammar that represents the structure of the supported C subset. This grammar is given in its full length in appendix G. The functions implementing the parsing stay closely to this grammar.

For each statement in code, the parser selects and executes a number of encoding functions to model it.

To explain this with an analogy: it is like constructing a model using building blocks. For each component in the construction manual (which has been derived from the original blueprint), a number of blocks is taken from the set of block types and added to the model. A block type can be used multiple times for different purposes.

4.1.6. Encoding

Giving the encoder its namer, the actual encoding is done by a number of encoding functions, i.e. functions adding clauses to the list of clauses. These functions implement the encoding of operations and coverage types as shown in in chapter III.

The encoding functions take program-variables as Variable objects or integers identifying SAT-variables as input. An exception is the function encoding a numeric value. It requires along with a target variable the respective numeric value as input. These functions encode a relation between their input variables. They do not return a value.

An encoding function can make use of other encoding functions. For example the encoding function for a full adder makes use of the encoding function for a half adder twice.

4.1.6.1. Auxiliary Encoding Functions

Along with the regular encoding functions, there are three auxiliary functions:

- `getTmpVar()` – providing an identifier for an auxiliary variable object.
- `getID()` – get a numeric identifier for a single new SAT-variable (internally uses `getIDs(1)`)
- `getIDs(size)` – get `size` number of numeric identifiers for new SAT-variables
- `getSingleID()` – provides a single ID (i.e. not one for each test case). This is only used for encoding the coverage criteria.

4.1.6.2. Encoding of Basic Operations

The encoding of basic operations follows closely the approach discussed in section 3.2. For the individual operations, please refer to that section.

There is an encoding function for each operation discussed in this section. Additional encoding functions exist, were an operation is handled by several functions, or for shared functionality. Especially for the logic operations, different functions can be used when an operation shall apply to all test cases or only to specific SAT variables.

4.1.6.3. Encoding of Coverage Criteria

Encoding of the coverage criteria is done after the source has been completely parsed, and the basic operations have been encoded. The encoding of the coverage makes use of the same logical, comparison, and assignment functions created for the encoding of basic operations.

For the individual types of coverage, and how their criteria get encoded, please refer to section 3.3.

The decisions and conditions and their respective relations, will be retrieved from the

corresponding lists of objects created during parsing. Which type of coverage is to be applied, is decided based on the users initial input.

4.1.7. Export to DIMACS

The collected list of clauses has to be assembled to a text file in DIMACS format, along with additional information on the number of clauses and the number of variables used. While the number of variables had ben counted along with the creation of the clauses, the number of clauses is simply derived from the length of an array that was used to store the clauses until this point.

Additional information is gathered and added as comment lines to the file. This includes:

- file name of the input source code
- name of the function for which the test cases are to be derived
- selected coverage type
- number of test cases
- mapping of the variables and parameters

The format, in which this information will be included in the DIMACS output file, has been discussed in section 4.1.1.2 on the encoder output.

4.2. Implementation of the Decoder

The second command line application implemented is the decoder. It decodes the test cases from the output of the SAT solver.

4.2.1. External Behavior

The external behavior of the decoder is defined by its in- and outputs. The inputs along with their sources, and the output of the decoder can be seen in figure 4.8.

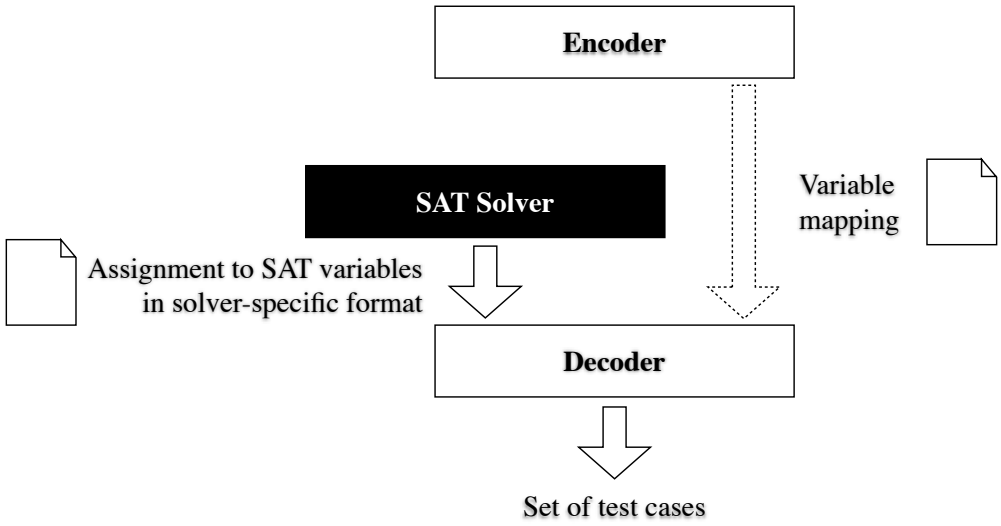


Figure 4.8.: In- and output of the decoder

4.2.1.1. Input

Input comes from two sources: SAT-solver, and encoder. The files containing this input have to be defined as command line arguments as follows:

```
1 ruby coverage-sat-decoder.rb solver-output.txt encoder-output.cnf
```

Defining the encoder output file is optional. The standard output file name which is used by the encoder ("output.cnf") will be used otherwise.

The format of the SAT-solver output is not standardized, but specific for each SAT-solver. PicoSAT [Bie08, Bie13a] has been used as a solver, so its output format will be used here. The decoder can be extended to support output formats for other solvers..

When the SAT-solver found an assignment, the SAT-solver output format is similar to the following example:

```
s SATISFIABLE
v -1 2 -3 -4 5 -6 -7 8 9 10 -11 12 13 -14 15 -16 -17 -18 -19 20
v  21 22 -23 -24 25 26 27 28 -29 -30 31 32 -33 34 -35 36 37 -38
```

⋮

```
v ... 0
```

The user might not check the solver result before handing it to the decoder. Therefore, an input file stating unsatisfiability must also be accepted and properly handled by the decoder. Using PicoSAT it will only contain the following line.

```
s UNSATISFIABLE
```

From the encoder output file, information on the number of test cases and the mapping of parameters is needed. They are contained in the comment section of the encoder's output file, as shown in the following example:

```
c ...
c Test cases:    2
c
c Mapping:
c Parameter mapping:
c {w (16bit signed int) -> {1}, x (32bit unsigned int) -> {33}, z (16
    bit signed int) -> {97}}
c ...
```

4.2.1.2. Output

There are two cases to be distinguished for the output of the decoder. They are based on the SAT-solver's output file used as input for the decoder.

When the SAT-solver has given the result as "UNSATISFIABLE", no test cases can be derived from it. The following output is given:

```
Unsatisfiable, no test cases derived.
```

When the SAT-solver has given the result as "SATISFIABLE", the assignment of parameters for each test case is given like in this example:

```
Satisfiable, parameters for test cases as follows:
Test case 1: a = 1, b = 2, c = 'x' ...
Test case 2: a = 3, b = 4, c = 'y' ...
```

⋮

```
Test case n: ...
```

The output is given to standard output. When it is desired to have the output written to a file, standard output can be redirected. In unixoid systems this can be done without changing the program simply by redirecting standard output to a file. Example:

```
ruby coverage-sat-decoder.rb solver-output.txt > test-cases.txt
```

This writes the output to the file "test-cases.txt" in the same directory.

4.2.2. Deriving Test Cases

From the given input, the values of the parameters for each test case can be derived.

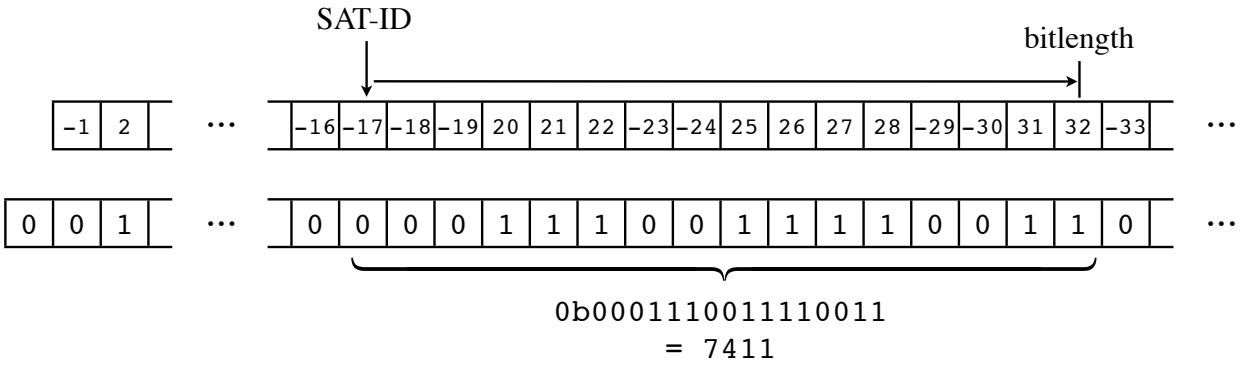


Figure 4.9.: Decoding from SAT-solver output file.

As seen in figure 4.9 the leftmost bit is given by the SAT-ID of a variable, the number of bits is given by the bitlength. The values for additional test cases are following just on the right.

Each SAT variable which is given with a minus sign makes a zero, each SAT variable without a minus sign makes a one. The zeros and ones are being stored in an array. An additional zero is added as first element of the array, as the array index starts with 0, but the SAT variables start with 1. The easiest way to derive a value is to join the sub-range of an array which contains a value to a string. The string can then be parsed to an integer value using Ruby's parsing functionality ("`.to_i(2)`").

When the type of a parameter is signed, values with "1" as their leftmost bits stand for negative numbers. For these parameters, the values have to be transformed to negative numbers according to the two's complement representation.

For parameters of type char, the corresponding ASCII character will be given in output instead of a number. Non-printable characters will be denoted with a two to three letter abbreviation followed by the ASCII value in brackets. Character values exceeding the range which can be represented in ASCII (128 and above) will be output similarly to non-printable characters, but with three question marks instead of an abbreviation. Examples: $0 \rightarrow$ `NUL (0)`; $65 \rightarrow$ `'A'`; $128 \rightarrow$ `??? (128)`.

V. Examples

The capabilities of the implemented solution are shown in this chapter on a number of examples. A minimal motivational example is already included in the introduction (section 1.6). This chapter comprises a small example, a larger example, and an example, which is analogous to the failure of the SRI discussed in the excursus on Ariane 5 (section 1.1)

For all examples the following steps are necessary

Encoder call:

```
ruby coverage-sat-encoder.rb <source-file> <function-identifier> <
    coverage-type> <number-of-test-cases>
```

Solver call:

```
./picosat output.cnf > picosat-output.txt
```

Decoder call:

```
ruby coverage-sat-decoder.rb picosat-output.txt > decoder-output.txt
```

Instead of doing these three steps manually, it is also possible to run them in a row, by using the script. It is called the following way:

```
ruby coverage-sat.rb main.c <source-file> <function-identifier> <
    coverage-type> <number-of-test-cases>
```

The output files of encoder and solver will not be given in this chapter in full length, as they are too long. All in- and output files of the shown examples are, however, included

on the enclosed CD.

5.1. Small Example

In the introduction (section 1.4), related work by Gotlieb et al. has been discussed. The following small example has therefore been taken from that source [GBR98, p. 58, figure 4]. The example code is already truncated in the source. Below an equivalent implementation in C is given.

```
1  int g(int x, int y) {
2    int z;
3    int t;
4    z = x * y;
5    t = 2 * x;
6    if (z <= 8) {
7      t = t - y;
8      if (t == 1 && x > 1) {
9        // ...
10     }
11   }
12   return 0;
13 }
```

5.1.1. Decision Coverage

Decision coverage can be achieved using three test cases. The values for each parameter in each test case can be seen in the decoder output given below:

```
1  Satisfiable, parameters for test cases as follows:
2  Test case 1: x = 1, y = 1
3  Test case 2: x = 1, y = 9
4  Test case 3: x = 2, y = 3
```

In the following the execution of each test case will be discussed step by step (the enumeration gives the line number in source code). This is to prove the derived test cases are correct, i.e. their execution achieves decision coverage. The same will be applied to other types of coverage later on.

Test case 1:

4. z takes the value 1.
5. t takes the value 2.
6. The first *if*-statement ($t <= 8$) evaluates *true* for 1 being less or equal 8.
7. t takes the value 1.
8. The second *if*-statement evaluates *false* for the conjunctive condition $x > 1$ evaluating *false*, as 1 is not greater 1. The other conjunctive condition $t == 1$ evaluates *true*, as 1 is equal 1.

Test case 2:

4. z takes the value 9.
5. t takes the value 18.
6. The first *if*-statement ($t <= 8$) evaluates *false* for 9 not being less or equal 8. The *if*-path (containing the second *if*-statement) is not executed.

Test case 3:

4. z takes the value 6.
5. t takes the value 4.
6. The first *if*-statement ($t <= 8$) evaluates *true* for 6 being less or equal 8.
7. t takes the value 1.
8. The second *if*-statement evaluates *true* for both conjunctive conditions $t == 1$ and $x > 1$ evaluating *true*, as 1 is equal 1 and 2 is greater 1.

From this it can be seen, that the decision for each *if*-statement takes each of the two possible outcomes at least once. The test cases provided are correct.

5.1.2. Condition Coverage

Condition coverage can also be achieved using three test cases. The decoder gives the following output:

```
Satisfiable, parameters for test cases as follows:
Test case 1: x = 1, y = 1
Test case 2: x = 1, y = 9
Test case 3: x = 2, y = 1
```

Test case 1 and 2:

These test cases are the same as for decision coverage discussed above.

Test case 3:

4. z takes the value 1.
5. t takes the value 4.
6. The first *if*-statement ($t <= 8$) evaluates *true* for 1 being less or equal 8.
7. t takes the value 3.
8. The second *if*-statement evaluates *false* for the conjunctive condition $t == 1$ evaluating *false*, as 3 is not equal 1. The other conjunctive condition $x > 1$ evaluates *true*, as 2 is greater 1.

From this it can be seen, that each condition takes each of the two possible outcomes at least once. The test cases provided are correct.

The second *if*-statement's decision evaluates *false* in the two test cases reaching this part of code. Therefore, with the derived test cases condition coverage can be achieved, but not decision coverage. The combined condition/decision coverage is derived in the next section

5.1.3. Condition/Decision Coverage

Condition/decision coverage does as well not require a higher number of test cases. It can be achieved using three test cases. The decoder gives the following output:

```
1 Satisfiable, parameters for test cases as follows:
2 Test case 1: x = 1, y = 0
3 Test case 2: x = 1, y = 9
4 Test case 3: x = 2, y = 3
```

Test case 1:

4. z takes the value 0.
5. t takes the value 2.
6. The first *if*-statement ($t <= 8$) evaluates *true* for 2 being less or equal 8.
7. t takes the value 2.
8. The second *if*-statement evaluates *false* for both conjunctive conditions $t == 1$ and $x > 1$ evaluating *false*, as 2 is not equal 1 and 1 is not greater 1.

Test case 2 and 3:

These test cases are the same as for decision coverage discussed above.

From this it can be seen, that each condition takes each of the two possible outcomes at least once. The same is true for each decision. The test cases provided are correct.

5.1.4. Modified Condition/Decision Coverage

Modified condition/decision coverage can not be achieved with only three test cases. Four test cases are needed, as given in the following decoder output:

```
1 Satisfiable, parameters for test cases as follows:
2 Test case 1: x = 1, y = 9
3 Test case 2: x = 1, y = 1
4 Test case 3: x = 2, y = 3
5 Test case 4: x = 2, y = 1
```

Test case 1 to 3:

These test cases are identical to the test cases from decision coverage, with 1 and 2 switching positions.

Test case 4:

This test case is identical to test case 3 from condition coverage.

In these test cases there is a pair of test cases for each condition, which shows the individual condition's influence on the decision's outcome. Condition and decision coverage is also achieved when executing the test cases. The test cases provided are therefore correct.

5.2. Larger Example

This example allows to meaningfully apply all types of coverage on a larger piece of code and over a higher number of test cases.

When executed the function checks a given date (consisting of year, month and day of month) is valid. To make it more complex it works for Gregorian and Julian calendar, i.e. any date since 9 AD.

In the following its source code is given:

```
1  int check_date_validity(int year, int month, int day_of_month) {
2      int valid = 1;
3      int max_days;
4
5      // 31 day month
6      if (month == 1 || month == 3 || month == 5 || month == 7 || month ==
           8 || month == 10 || month == 12) {
7          // January 1, March 3, May 5, July 7, August 8, October 10,
               Deecember 12
8          max_days = 31;
9      } else {
10       // 30 day month
11         if (month == 4 || month == 6 || month == 9 || month == 11) {
12             // April 4, June 6, September 9, November 11
13             max_days = 30;
14         } else {
15             // February 2
```

```
16              max_days = 28;
17              // Year after 1582
18              if (year > 1582) {
19                  // using Gregorian calendar
20                  if ( year % 4 == 0 && (year % 100 != 0 || year % 400 ==
                        0) ){
21                      //leap year - 29 days
22                      max_days = 29;
23                  }
24              } else {
25                  // using Julian calendar
26                  if ( year % 4 == 0 ){
27                      //leap year - 29 days
28                      max_days = 29;
29                  }
30              }
31          }
32      }
33
34      // Out of range check
35      if (year < 5 || month > 12 || month < 1 || day_of_month > max_days
            || day_of_month < 1 ) {
36          // Julian calendar not applicable before 5 AD
37          // OR month out of range OR day of month below minimum
38          valid = 0;
39      }
40
41      // Julian/Gregorian gap check
42      if (year == 1582 && month == 10 && day_of_month > 4 && day_of_month
            < 15) {
43        // non-existing days due to julian/gregorian gap (Oct. 5 - 14,
              1582)
44          valid = 0;
45      }
46
47      return valid;
48  }
```

5.2.1. Decision Coverage

```
1  Satisfiable, parameters for test cases as follows:
2  Test case 1: year = 1582, month = 10, day_of_month = 5
3  Test case 2: year = 4672, month = 26, day_of_month = 0
4  Test case 3: year = 9303, month = 11, day_of_month = 0
5  Test case 4: year = -16848, month = 13, day_of_month = 5
```

For decision coverage each *if*-statement should be decided at least once to be *true* and at least once to be *false*. In the following this is checked for each *if*-statement:

- 31 day month *true* for test case 1, *false* for all other test cases.
- 30 day month *true* for test case 3, not reached by test case 1, *false* for all other test cases.
- Year after 1582 *true* for test case 2, not reached by test case 1 and 3, *false* for remaining test case 4.
- Out of range check *true* for test case 2 and 4, *false* for all other test cases.
- Julian/Gregorian gap check *true* for test case 1, *false* for all other test cases.

5.2.2. Further Coverage Types

Test cases to achieve condition coverage, condition/decision coverage (13 test cases each), and MC/DC (19 test cases) are given in appendix H.

5.3. Ariane SRI Analogous

This is an example analogous to the Ariane SRI code given in the introduction.

The problem has been adapted, due to limitations of the encoder implementation. It is not converting a floating point number to an integer, but an integer of longer bitlength to one of shorter bitlength. This operation has basically the same "out-of-range"-problematic.

It has also been changed for differences between Ada and C. Ada has implicit range checks for assignments among different types. C has no such checks, so an explicit check for the range has been implemented instead. Whether such a check is implicit or explicit should however not influence the resulting encoding. An encoder adapted for Ada would have to support implicit condition and decisions from these checks, as they as well contribute to the control flow. The unhanded Ada exception is emulated in C by an "abort()"

statement. This statement causes an abnormal program termination, similar behavior would be expected to be caused by an unhanded exception in Ada.

The source code is given below:

```
int ariane_sri_analogy(long measured_value){
    int derived_value;
    // Ada checks range, emulated by explicit "if"
    // valid range of 16 bit int is -32768 to 32767
    if (measured_value > 32767 || measured_value < -32768){
        // Ada has an unhandled eception, emulated by "abort()"
        abort();
    }else{
        // "measured_value" is in range, regular assignment
        derived_value = measured_value;
    }
    return derived_value;
}
```

5.3.1. Decision Coverage

Decision coverage can be achieved with two test cases. The test cases from the decoder output are shown below.

```
Satisfiable, parameters for test cases as follows:
Test case 1: measured_value = 0
Test case 2: measured_value = 32768
```

When executing the derived test cases, the first test case executes normally, but the second test case aborts abnormally. The failure in handling too large values has been revealed.

5.3.2. Condition Coverage

It is also possible to try to achieve condition coverage with two test cases. The following set of test cases given in the decoder output achieves it:

```
Satisfiable, parameters for test cases as follows:
Test case 1: measured_value = 32768
Test case 2: measured_value = -2147483648
```

For test case 1, the first condition is *true*, the second is *false*, vice versa for test case 2. These test cases comprise the possible outcomes of the conditions, but not of the decisions. Condition/decision coverage is needed for this.

5.3.3. Condition/Decision Coverage

With only two test cases, condition/decision coverage can not be achieved. The solver states "`UNSATISFIABLE`". With three test cases the minimal set of test cases has been found:

```
1 Satisfiable, parameters for test cases as follows:
2 Test case 1: measured_value = 0
3 Test case 2: measured_value = 32768
4 Test case 3: measured_value = -2147483648
```

The next step would be to give modified condition/decision coverage, but in this case it is not necessary. Based on the structure of the *if*-statement MC/DC is identical with condition/decision coverage as given above. This is because of the two conditions excluding each other, which gives them always individual influence on the outcome of the decision.

VI. Evaluation

In this chapter several aspects of capabilities and limitations of the introduced approach are being evaluated. This includes:

- applicability of the approach to other programming languages,
- characteristics of solvers, and
- limits of length and complexity.

6.1. Applicability of the Approach to other Programming Languages

The approach has been implemented for a subset of C. A subsequent question would of course refer to the portability of the approach. That is to say: which other languages the approach could be applied to. Expansions and modifications of the approach may be required for this.

It is not feasible to evaluate this question here for every language. Therefore certain properties of languages will be discussed.

6.1.1. Typing

Typing refers to the two opposing ways programming languages can handle their variables: statically typed or dynamically typed. The approach has been applied to C – a

statically typed language.

A variable's type is of high importance for the encoding in SAT. Values of variables are comprised of bits, these bits are in the encoding represented by SAT variables. The number of bits is defined by the type of a variable. Therefore, it is important to know the type of a variable to encode it using the proposed approach. A variable's type also gives information which operations can be applied on a variable (and if so, in which way).

For statically typed programming languages this is not much of a problem, as each variable (including function parameters) is explicitly typed and this typing remains unchanged. In C there are only some points where a variables type has to be inferred in some way. For example in the statement $a = a + 1$, the approach assigns the value "1" to an auxiliary variable. This auxiliary variable needs a type – this type is implicitly given by the known type of variable a.

For dynamically typed programming languages this is much more complicated. The approach would need major modifications to be applied there. In many situations the type could be inferred as similar to the example shown above: assigning an integer value makes a variable an integer, assigning a character value makes a variable a character, and so on. However, this type does not necessarily stay the same over a program's runtime. A type would have to be mapped to each value, instead of a single type being mapped to a variable as of now. There is one point – unfortunately an important one – where inferring types does not work: the parameters. Two alternatives are possible to handle this problem:

1. try every possible combination of type for input – problematic, because of an exponential number of combinations c from number of types t and number of parameters p. It gives: $c = p^t$,
2. assign types for each parameter in each test case manually – undesirable, because the aim was to have automation.

For the difficulties with dynamically typed languages, the approach is rather suited for statically typed languages. The problems with dynamically typed languages are not only

present for this new approach. Statically typed programming languages have been the focus of previous research on automatically generation of test cases. Recent development has begun to tackle this problem [MFT11, p. 1859].

6.1.2. Pointers

Pointers have not been part of the chosen subset of C. This is because pointers are a major obstacle for modeling a piece of code using SAT. One approach would be to dissolve pointers, i.e. to replace pointers by the actual variable they point to.

6.1.3. Object Orientation

No statement was made regarding the applicability to objects in object oriented languages. This thesis discussed the application to functions in C, a procedural programming language, so object orientation was out of scope.

In procedural programming languages the approach applies to functions. Correspondingly in object oriented languages it would be applied to methods. Methods of objects only operating with basic or composed data types, can basically be handled the same way as functions. Some additional modeling would be required to correctly reflect the relation of methods to an object's attributes.

Another point would be handling models as variable types. An objects class can be seen as a composite type (discussed as a possible extension in section 7.1.2).

6.1.4. In- and Output

A user input within a function can be handled like it was an additional parameter to the function. User input and parameters are very similar. Both are unknown values, potentially contributing to the control flow.

In C, inputs are straight forward. The `scanf` function call includes information on how

to interpret the input, and which variable to assign it. This is shown in the following example:

```
1 int x;
2 scanf("%d", &x);
```

Regardless of what the user types in, C will interpret its content as if it was of the desired type. In other programming languages it can be more complicated.

A common style to be seen is treating a user input as string first, then parse it to whatever type desired. Parsing is of course often associated with the possibility of raising an exception, when the input is incompatible. So these parsing operations contribute implicit conditions and decisions to the control flow. This means, that for example decision coverage would have to cover the exception path, induced by erroneous user input.

Output is in general not relevant for the encoding, as it does not contribute to the control flow. This is of course only true unless some operation of relevance is performed within the output. One example is an incrementation changing a relevant variable:

```
1 printf("%d", x++);
```

But reduced to the relevant parts, it would be only:

```
1 x++;
```

6.2. Characteristics of Solvers

PicoSAT – the solver used on the examples – is deterministic, i.e. solving the same input file several times will give the same output again. From own experience, PicoSAT also seems to prefer to assign a SAT variable to be *false*, rather than *true*. For the parameters in test cases derided with the approach using PicoSAT this means their values will likely be round numbers (round in the binary system, not in the decimal system[1]). This often leads

[1] e.g. 0, 1, 2, 4, 8, 16, ... 1024, ...

to positive numbers of low value, or negative numbers of high absolute value (among the lowest values in range of a variable type).

A simple CNF formula was created to show this behavior. It enforces at least one of 10 variables to be true and at least one of them to be false:

$1 \vee 2 \vee 3 \vee 4 \vee 5 \vee 6 \vee 7 \vee 8 \vee 9 \vee 10$

$\neg 1 \vee \neg 2 \vee \neg 3 \vee \neg 4 \vee \neg 5 \vee \neg 6 \vee \neg 7 \vee \neg 8 \vee \neg 9 \vee \neg 10$

The formula was encoded in DIMACS format and handed to the solver ten times. PicoSAT[2] produced all ten times the identical following result:

-1 -2 -3 -4 -5 -6 -7 -8 -9 10

There is no factual basis for preferring this assignment. It is an implementation defined behavior of the solver.

But there are other solvers, which based on their different implementation give results with inherent randomness. One of those solvers is Walksat [SK12, SKC93]. When given the same input, Walksat[3] produced the following results:

-1 -2 3 4 5 -6 -7 -8 9 10
-1 -2 3 -4 5 -6 7 -8 9 10
-1 2 -3 -4 -5 6 7 -8 9 10
-1 2 3 -4 -5 -6 7 -8 -9 10
-1 2 3 4 5 -6 -7 8 9 -10
-1 -2 3 -4 5 -6 7 -8 -9 10
1 -2 -3 -4 5 -6 -7 8 9 10
1 -2 3 -4 -5 -6 -7 8 9 -10
-1 -2 -3 -4 5 -6 -7 -8 -9 -10
-1 -2 -3 4 -5 6 -7 -8 9 10

To evaluate Walksat, it has been used as an alternative solver for the simple motivational

[2]Version: Picosat-957
[3]Version: Walksat_v50

example. Experimental support for the Walksat output format has been implemented in the decoder (a switch in source code exists). The same DIMACS output of the encoder, which has previously been used with PicoSAT, was handed to Walksat several times. When solved and decoded, from all possible sets of test cases, which might fulfill the encoded coverage type, a (pseudo-)random set of test cases is given.

These are two examples of derived test cases using Walksat based on the same CNF file:

```
1 Test case 1: days_task1 = 10, days_task2 = 13
2 Test case 2: days_task1 = 15, days_task2 = 12
```

```
1 Test case 1: days_task1 = 14, days_task2 = 9
2 Test case 2: days_task1 = 2, days_task2 = 12
```

When executing the test cases derived from Walksat's solution, the parameters and return value may seem inconsistent, but they are not. As mentioned in the description of this example, unsigned integers of 4 bits length are used for demonstration purpose. This variable type can represent natural numbers from 0 to 15. An integer overflow – a perfectly legal event in C – happens when exceeding this range. Walksat's solution therefore leads to a correct coverage. Executing the test cases has not only covered the code, it also revealed a weakness in the implementation: missing checks for overflow.

The discovered problem could then be fixed (one possibility is given in the following piece of code), new test cases generated, and then retested. A retest with the previous test cases is not sufficient, as the control flow is changed.

```
5     if(days>7 || days_task1 > 7 || days_task2 > 7){
6           // sum of days exceeds one week
```

Walksat's results show (pseudo-)randomness as opposed to PicoSAT being predictable. This randomness would be a good feature for the problem of deriving test cases. Without a remaining factual basis for selection (each alternative fulfills the coverage criteria), random selection is the most reasonable way to go. Values for the parameters of course can not be completely random numbers. Only certain sets of assignments fulfill the constraints encoded. But from these sets one will be picked in a non-deterministic way.

The disadvantage of the use of Walksat for the given problem is that it is an incomplete solver. It also is not as capable regarding larger constraint systems. This means when it can not find a solution in a specific number of iterative trials it will stop. This does not mean unsatisfiability of the problem, it just remains undecided. Unsatisfiability, however, is an important feature for the approach. Unsatisfiability (when the number of test cases should in fact be sufficient) reveals unreachable paths. To prove unsatisfiability a complete solver like PicoSAT is required.

Complete solvers do not stop and state inconclusiveness. In some cases the solver might find an assignment for n test cases in a time t_n, but for $n-1$ test cases keeps running considerably longer ($t_{n-1} \gg t_n$) without giving a result. This is likely to be caused by no existing solution for $n-1$ test cases. Yet, the solver keeps running until the constraint system has been *proven* to be unsatisfiable. Proving unsatisfiability can be hard for the solver. In this case it is reasonable to abort the solver execution and use the result for n test cases instead.

6.3. Limits of Length and Complexity

The limits of problem instances the approach can be applied to are related to length and complexity of code, and the number of test cases used. The overall limitations are composed of the limitation of encoder, solver, and decoder. SAT solvers have proven to be tough tools. Encoder and decoder on the other hand are prototype applications, developed as a proof of concept. As of now limitations are therefore rather to be expected within the implementation of encoder and decoder than in the capabilities of the solver. This is, however, depending on the actual solver used.

A problem with the number of test cases was found, when trying to use MC/DC. MC/DC applies constraints across all combinations of conditions over all combinations of test cases. Obviously this can become a very large number of constraints. In the Boolean formula generated for MC/DC (see section 3.3.4, and appendix F) there are blocks in curly brackets, containing blocks in square brackets, containing statements in round brackets.

The following table 6.1 gives an overview of the size when applied to the large example with 19 test cases as discussed previously (section 5.2).

Curly bracket blocks	25
Square bracket blocks	4,275
Statements in round bracket	24,282

Table 6.1.: Number of appearances of blocks and statements for MC/DC encoding.

PicoSAT was not capable of solving the generated constraints to derive test cases. A different solver, "lingeling" [Bie13b] had to be used to solve this example. lingeling was chosen because of being a winner in the 2013 SAT competition **??**. Its output format is compatible to PicoSAT, so the decoder can use it without being adapted.

A full evaluation regarding the length and complexity of code the approach can be applied to is still pending. But it should be noted, that overly complex and long functions should be avoided in the first place. Long functions and functions comprising a too high complexity of functionality (so called *god functions*[4]) are *code smells* [CP07, CP11].

"A code smell is a hint that something has gone wrong somewhere in your code" [CP13]. It does not mean the code has to be faulty, but there is a suspicion it could be, based on the experience that it is more likely to be.

"There are two major approaches to programming: [...]

- Pragmatic: CodeSmells should be considered on a case by case basis
- Purist: all CodeSmells should be avoided, no exceptions" [CP13].

The purist approach applies especially to critical software.

When the way software is developed counteracts the goal of producing correct software, even toughest testing can hardly compensate for this. It would likely result in finding large amounts of faults, having to correct, and retest them. This would lead to exceeding time, and cost constraints, and ultimately a failed project.

[4]The source – having a strictly object oriented perspective – discusses *god methods*, but the same principle applies as well to functions

So in this respect the limits of the automated generation of test cases might be only of subordinated practical relevance. Only support for a certain degree of length and complexity would be required. This grade is to be defined.

VII. Conclusion and Outlook

In the previous chapters the approach has already been tested (by applying it to examples) and evaluated. It can be concluded, that the proposed approach is applicable within the implemented domain and worthy of further research.

While having achieved considerable progress, the implemented program is – as intended – a prototype with a certain chosen set of functionality. An overview shall be given here on possible enhancements which would have exceeded the scope and time frame of this thesis. Several areas have been identified, in which future work could further improve the achieved status or build upon it. This includes:

- expanding the supported subset of C
- improving the workflow
- providing more options for coverage criteria

7.1. Expanding the Supported Subset

The implementation as of now covers only a defined subset of C. To be applicable to a larger number of real life C code samples, this subset needs to be expanded.

To expand the supported subset of C, obvious next steps would be implementing additional basic types, implementing composite types, and implementing function calls.

7.1.1. Additional Basic Types

The implementation of additional basic types would address primarily floating point numbers (`float`, `double`, `long double`). Floating point types have intentionally been postponed, as they are more difficult to implement compared to integer and character types. Their bits could be mapped to SAT variables as shown in figure 7.1. Their behavior could be modeled by appropriate constraints. With their implementation support for the most important basic types would be complete.

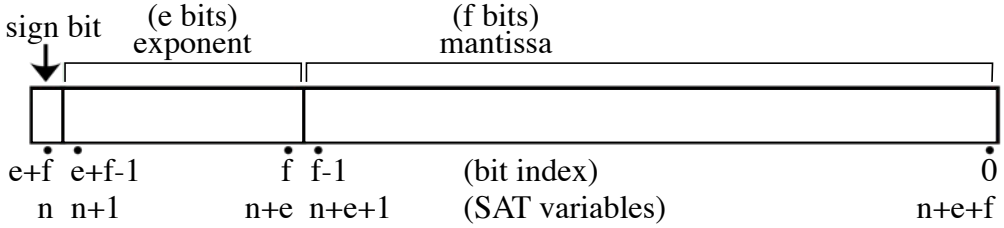


Figure 7.1.: Representing floating point numbers using SAT variables [Sta07].

7.1.2. Composite Types

Composite basically are assembled from a number of basic types. Building upon the implemented basic types, composite types could be implemented.

The composite types array, string, and struct could be implemented. An array is basically a number of variables of the same type in a row. The same is true for a string, being composed of a number of characters. Structs also consist of a number of variables in a row, but these variables can be of different types.

The implementation of composite types would be straight forward with the implementation of required basic types already existing.

7.1.3. Function Calls

The encoding as of now works only on a single function. Yet all other functions from the input source file are already stored in a hash table with their function identifier as key

too. They could be accessed to be used in function calls. This functionality could be used for function stubs as they are common in software testing. The function calls should be limited to non-recursive calls. Recursive calls would create implicit loops, which are hard to model. It might be helpful to read in more than just one source file as it is now, because the functions used might be spread across several files.

7.2. Improving the Workflow

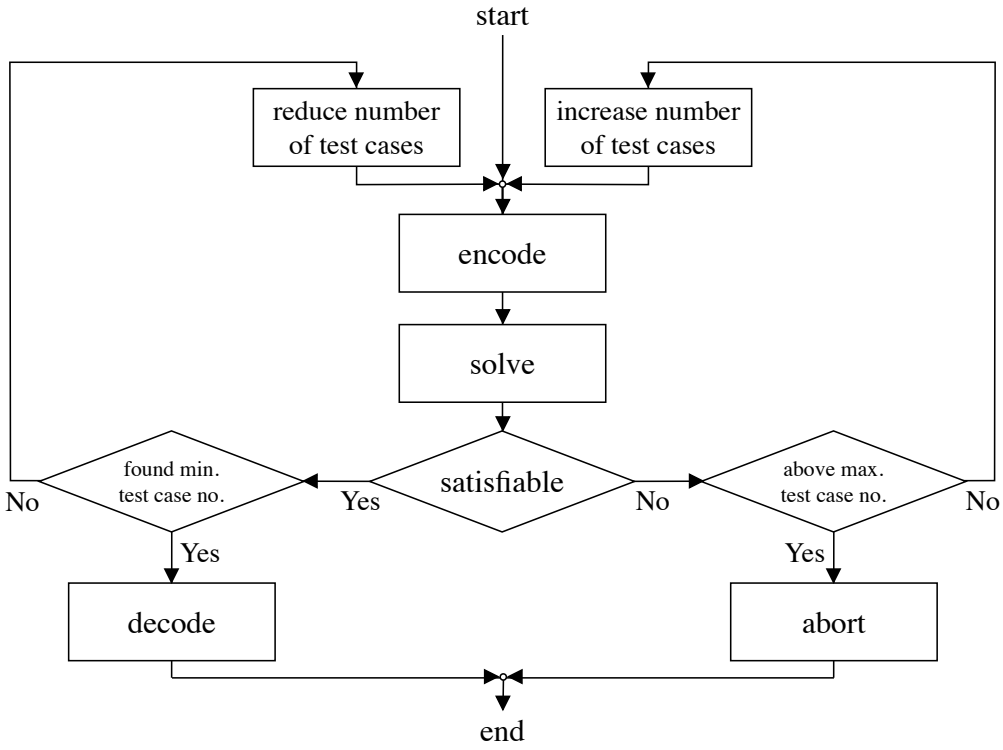


Figure 7.2.: Workflow for achieving a minimum test case number. Choosing the number of test cases with "reduce number of test cases" and "increase number of test cases" should be implemented following a binary search pattern.

Automatizing the current workflow would benefit the user. To find the minimum number of test cases needed to achieve a certain type of coverage, more than one iteration of encoding and solving is necessary. As of now this has to be done manually. The number of test cases is generally not too high, so this is feasible. However, it takes some additional time and effort from the user. This workflow could be automatized as seen in figure 7.2.

The figure basically shows the same workflow which as of now has to be done manually when looking for the minimum number of test cases. Increase and reduction of the number of test cases could be done in the way of a binary search. This would lead to a small number of necessary iterations until the minimum number of test cases would be found.

7.3. Enhancing Coverage Criteria

Enhancements are also possible regarding the implementation of coverage types. Individual points in this area could be:

- evaluating existing test cases
- supporting grades of coverage
- implementing additional coverage criteria and types of coverage

7.3.1. Evaluating Existing Test Cases

As of now, new test cases are derived fulfilling the coverage criteria. Alternatively, existing test cases could be checked whether they fulfill the coverage criteria or not. For example requirement based test cases could be evaluated in this way. The parameters would have to be constraint to take certain values. This would require only little additional work. Existing encoding functions could be reused for it. It would also be possible to pre-set the parameter just for some test cases, but leave one or more test cases open. This would allow for existing test cases not fulfilling the coverage criteria alone to be supplemented by one or more new test cases, so they can fulfill the coverage criteria together.

7.3.2. Support for Grades of Coverage

Instead of searching for lowest number of test cases for 100% fulfillment of coverage criteria, it would also be possible to search for the highest possible grade of fulfillment of

coverage criteria with a given fix number of test cases.

Pseudo-Boolean constraints with a cost-function could be used to achieve this. They could be solved by a pseudo-Boolean solver or translated to SAT and solved iteratively using a SAT solver.

For condition coverage and decision coverage there is only a single coverage criterion. Its fulfillment could be easily measured by the respective ratio. Condition/decision coverage and modified condition/decision coverage have more than one coverage criterion. Each coverage criterion would have to be weighted to give an overall percentage of fulfillment of the coverage criteria.

7.3.3. Additional Coverage Criteria and Types

Additionally to improving the handling of already supported types as discussed above, further types of coverage could be implemented. This applies to any type of coverage on condition that its metrics are based on aspects of the source code (i.e. requirements coverage or similar coverage types are as a matter of course not possible). Several further coverage types of that ilk exist. Some of them are given in the following list:

- control flow coverage:
  - statement coverage
  - all-path coverage
- data flow coverage [RW82, p. 276]:
  - all-paths
  - all-du-paths
  - all-uses
  - all-defs
  - all-c-uses / some-p-uses
  - all-p-uses / some-c-uses
  - all-p-uses
  - all-edges
  - all-nodes

The necessary effort of implementation would depend on the characteristic of the new coverage type and its coverage criteria.

References

[BBB+13] Justin Baker, Daniel Berger, Simon Bohlin, Alexey Borzenkov, Rodolfo Budeguer, Lars Christensen, Huw Collingbourne, Timothy Elliott, Park Heesob, Curt Hibbs, Martin Hrdlička, Bosko Ivanisevic, Luis Lavena, Pavel Maček, Jon Maken, Roger Pack, Charles Roper, Hiroshi Shirosaki, and Gordon Thiesfeld. *RubyInstaller for Windows*. `http://rubyinstaller.org`, last checked: 2013-08-27.

[BBH+13] Adrian Balint, Anton Belov, Marjin Hele, Matti Järvisalo, Daniel Le Berre, Oliver Roussel, Laurent Simon, and Edward A. Hirsch. *The international SAT Competitions web page*. `http://www.satcompetition.org`, 2013, last checked 2013-09-29.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[Bie08] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[Bie13a] Armin Biere. *PicoSAT*. `http://fmv.jku.at/picosat/`, last checked 2013-09-24.

[Bie13b] Armin Biere. *Lingeling, Plingeling and Treengeling*. `http://fmv.jku.at/lingeling/`, last checked 2013-09-29.

[Bin07] Ola Bini. *Practical JRuby on Rails Web 2.0 Projects: Bringing Ruby on Rails to Java*. Berkeley: APress, 2007.

[Bla13a] Patrick Blau. *Ariane 5-ECA Launch Vehicle*. http://www.spaceflight101.com/ariane-5-eca.html, last checked 2013-07-30.

[Bla13b] Patrick Blau. *Delta IV Heavy - RS-68A*. http://www.spaceflight101.com/delta-iv-heavy.html, last checked 2013-07-30.

[Bla13c] Patrick Blau. *Proton-M/BrizM - Launch Vehicle*. http://www.spaceflight101.com/proton-m-briz-m.html, last checked 2013-07-30.

[BMMS11] Domagoj Babić, Lorenzo Martigoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 12–22, July 2011.

[Bur06] Colin M. L. Burnett. *4-Bit Ripple Carry Adder*. Wikimedia Commons, http://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg, (GFDL, CC BY-SA 3.0), 2006-12-19, last checked 2013-06-17.

[Com12] Sylvain Comte. *General scheme of the Ariane 5 rocket, based on multiple sources [Schéma général de la fusée Ariane 5, basé sur des sources multiples]*. Wikimedia Commons, http://commons.wikimedia.org/wiki/File:Schema_Ariane_5.svg, (CC BY-SA 3.0), 2012-07-30, last checked 2013-07-31.

[Coo09] Peter Cooper. *Beginning Ruby: From Novice to Professional. Beginning from Novice to Professional*. Berkeley: APress, 2 edition, 2009.

[CP07] Cunningham & Cunningham Inc. and Portland Pattern Repository Contributers. *Long Functions*. Portland Pattern Repository, http://c2.com/cgi/wiki?LongFunctions, last edited 2007-05-17, last checked 2013-09-30.

[CP11] Cunningham & Cunningham Inc. and Portland Pattern Repository Contributers. *God Method*. Portland Pattern Repository, http://c2.com/cgi/wiki?GodMethod, last edited 2011-07-14, last checked 2013-09-30.

[CP13] Cunningham & Cunningham Inc. and Portland Pattern Repository Contributers. *Code Smell*. Portland Pattern Repository, http://c2.com/cgi/

wiki?CodeSmell, last edited 2013-05-19, last checked 2013-09-30.

[Dav01] Martin Davis. The early history of automated deduction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Deduction*. MIT Press, 2001.

[dDG97] Juan de Dalmau and Jacques Gigou. Ariane-5: Learning from flight 501 and preparing for 502. *ESA Bulletin*, (89):38–47, February 1997.

[DIM93] *Satisfiability Suggested Format*. DIMACS (Center for Descrete Mathematics & Theoretical Computer Science), Rutgers State University of New Jersey, 1993.

[DP58] Martin Davis and Hilary Putnam. *Computational methods in the propositional calculus*. unpublished report, Rensselaer Polytechnic Institute, 1958.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:226, 2006.

[ESA96] European Space Agency (ESA). *Ariane 501 – Presentation of Inquiry Board Report*. esa plain text press releases № 33–1996, http://www.esa.int/For_Media/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report, 1996-07-23, last checked 2013-07-30.

[ESA13a] European Space Agency (ESA). *Launches Archive*. http://www.esa.int/Our_Activities/Launchers/Launches_archive, last update 2013-03-14, last checked 2013-07-30.

[ESA13b] European Space Agency (ESA). *Ariane 5*. http://www.esa.int/Our_Activities/Launchers/Launch_vehicles/Ariane_5, last update 2013-06-20, last checked 2013-07-30.

[ESA13c] European Space Agency (ESA). *Previous Launches*. http://www.esa.int/Our_Activities/Launchers/Previous_launches, last update 2013-07-26, last checked 2013-07-30.

[Eva13] Clark C. Evans. *The Official YAML Web Site*. `http://www.yaml.org`, last checked: 2013-08-28.

[FM09] John Franco and John Martin. A history of satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 1, pages 3–74. IOS Press, 2009.

[For60] Robert Fortet. Applications of boolean algebra in operations research [applications de l'algèbre de boole en recherche opérationnelle]. *French Journal of Operational Research [Revue Française de Recherche Opérationelle]*, 4:17–26, 1960.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving rechniques. *ISSTA '98 Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, 1998.

[Gra08] W. D. Graham. *Proton-DM – Colour drawing of a Proton rocket with a Block DM upper stage*. Wikimedia Commons, `http://commons.wikimedia.org/wiki/File:Proton-DM.svg`, © GW_Simulations[1], 2008-06-27, last checked 2013-07-31.

[Hru08] Tomáš Hruška. *Delta IV family*. Wikimedia Commons, `http://commons.wikimedia.org/wiki/File:Delta_IV_family.png`, public domain, 2008-12-22, last checked 2013-07-31.

[HS00] Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on sat. In Ian Philip Gent, Hans van Maaren, and Toby Walsh, editors, *SAT 2000*, pages 283–292. IOS Press, 2000.

[HVCR01] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. *A Practical Tutorial on Modified Condition / Decision Coverage*. National

[1]The copyright holder of this file, GW_ Simulations, allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.

Aeronautics and Space Administration (NASA), Langley Research Center, Hampton, Virginia, May 2001.

[ISO07] ISO/IEC. *ISO/IEC 9899:TC3 Committee Draft*. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`, September 2007, last checked 2013-05-26.

[JT96] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, volume 26. American Mathematical Society, 1996.

[Kol11] Pavel Kolotilov. *Model of Proton-M launcher with 5m fairing during the Paris Air Show 2011*. Wikimedia Commons, `http://commons.wikimedia.org/wiki/File:Model_of_Proton-M_launcher_with_5m_fairing.jpg`, (CC BY-SA 3.0), 2011-06-24, last checked 2013-07-31.

[Kro09] Daniel Kroening. Software verification. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 16, pages 505–532. IOS Press, 2009.

[Kyl13] Ed Kyle. *Space Launch Report*. `http://www.spacelaunchreport.com/logsum.html`, last updated 2013-01-24, last checked 2013-07-30.

[Lev04] Nancy G. Leveson. The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets*, 41(4), July 2004.

[Lev10] Jean-Jacques Levy. *Un petit bogue, un brand boum !* Presentation at Ecole Normale Supérieure, Paris, `http://moscova.inria.fr/~levy/talks/10enslongo/enslongo.pdf`, 2010-01-26, last checked 2013-08-09.

[LGR11] Guodong Li, Indradep Ghosh, and Sreeranga P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. *23rd International Conference on Computer Aided Verification (CAV)*, pages 609–615, 2011.

[LLF+96] Jaque-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn,

Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. *ARIANE 5 – Flight 501 Failure – Report by the Inquiry Board*. http://http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf, 1996-07-19, last checked 2013-07-30.

[McC09] Carey McCleskey. *Launch Vehicle Reliability*. NASA, http://science.ksc.nasa.gov/shuttle/nexgen/Bayesian_launcher_reliability.htm, 2009-05-04, last checked 2013-07-30.

[MFT11] Stefan Mairhofer, Robert Feldt, and Richard Torkar. Search-based software testing and test data generation for a dynamic programming language. *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1859–1866, 2011.

[Mod12] Modelpractice.wordpress.com. *General Model Theory by Stachowiak*. http://modelpractice.wordpress.com/2012/07/04/model-stachowiak/, 2012-07-04, last checked 2013-04-20.

[MR07] Vasco Manquinho and Olivier Roussel. *Pseudo-Boolean Evaluation 2007*. http://www.cril.univ-artois.fr/PB07, 2007, last checked 2013-06-10.

[Phr09a] Phrd. *Ariane 501 Chronology [Ariane 501 Verlauf]*. Wikimedia Commons, http://commons.wikimedia.org/wiki/File:Ariane_501_Verlauf.svg, public domain, after [LLF+96], 2009-03-07, last updated 2009-03-30, last checked 2013-08-09.

[Phr09b] Phrd. *Ariane 501 Fallout Zone*. Wikimedia Commons, http://commons.wikimedia.org/wiki/File:Ariane_501_Fallout_Zone.svg, (CC BY 3.0), after [dDG97, p. 43], 2009-03-30, last checked 2013-08-09.

[Pic10] Picosat. *PicoSAT man page*. http://www.linuxcertif.com/man/1/picosat/, 2010, last checked 2013-06-11.

[Pre09] Steven Prestwich. Cnf encodings. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 2, pages

75–98. IOS Press, 2009.

[rdo13] *RDoc - Documentation from Ruby Source Files*. `http://rdoc.sourceforge.net`, last checked 2013-09-05.

[RM09] Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 22, pages 695–733. IOS Press, 2009.

[RM12] Olivier Roussel and Vasco Manquinho. *Input/Output Format and Solver Requirements for the Competitions of Pseudo-Boolean Solvers*. `http://www.cril.univ-artois.fr/PB10/format.pdf`, version: 44 , last modification: 2012-11-12, last checked: 2013-06-10.

[RP06] Peter Rechenberg and Gustav Pomberger. *Computer Science Handbook [Informatik Handbuch]*. Hanser, 4th edition, 2006.

[rub13a] ruby-lang.org. *About Ruby*. `http://www.ruby-lang.org/en/about/`, last checked: 2013-08-27.

[rub13b] ruby-lang.org. *Download Ruby*. `http://www.ruby-lang.org/en/downloads/`, last checked: 2013-08-27.

[rub13c] ruby-lang.org. *Ruby Programming Language*. `http://www.ruby-lang.org/en/`, last checked: 2013-08-27.

[Rus05] John Rusby. Automated test generation and verified software. *Verified Software: Theories, Tools, Experiments, Zurich, Switzerland*, October 2005.

[RW82] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. *ICSE '82 Proceedings ot hte 6th international conference on Software*, pages 272–278, 1982.

[Sch00] Uwe Schöning. *Logic for Computer Scientists [Logik für Informatiker]*. Spektrum Heidelberg-Berlin, 2000.

[SK12] Bart Selman and Henry Kautz. *Walksat Home Page*. http://www.cs.rochester.edu/u/kautz/walksat/, last updated 2012-03-31, last checked 2013-09-23.

[SKC93] Bart Selman, Henry Kautz, and Bram Cohan. Local search strategies for satisfiability testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, October 1993.

[SL10] Andreas Spillner and Tilo Linz. *Basic Knowledge Software Test [Basiswissen Softwaretest]*. dpunkt.verlag Heidelberg, 4th edition, 2010.

[Soc10] INFORMS Computing Society. *Mathematical Programming Glossary – Constraint*. http://glossary.computing.society.informs.org/index.php?page=C.html, 2010, last checked 2013-06-26.

[Sta73] Herbert Stachowiak. *General Model Theory [Allgemeine Modelltheorie]*. Springer, 1973.

[Sta07] Ed Stanner. *General floating point*. Wikimedia Commons, http://commons.wikimedia.org/wiki/File:General_floating_point.svg, (GFDL, CC BY-SA 3.0), 2007-03-22, last checked 2013-09-07.

[Tio13a] Tiobe Software. *TIOBE Programming Community Index for July 2013*. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2013-07, last checked 2013-08-01.

[Tio13b] Tiobe Software. *TIOBE Programming Community Index for August 2013*. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2013-08, last checked 2013-08-27.

[WDR09] W. Eric Wong, Vidroha Debroy, and Andrew Restrepo. The role of software in recent catastrophic accidents. *IEEE Reliability Society Annual Technology Report*, 2009.

[Wei12] Stefan Weiler. *Pseudo-Boolean Constraints*. Darmstadt University of Applied Sciences, Department of Computer Science, Seminar (JIM/Master):

Boolean Satisfiability w/ Alexander del Pino, 2012.

[Whe08] David A. Wheeler. *MiniSAT User Guide: How to use the MiniSAT SAT Solver*. `http://www.dwheeler.com/essays/minisat-user-guide.html`, June 2008, last checked: 2013-03-11.

[Wik04] Wikibooks Contributors. *Ruby Programming/Installing Ruby — Wikibooks, The Free Textbook Project*. `http://en.wikibooks.org/w/index.php?title=Ruby_Programming/Installing_Ruby&oldid=2362762`, 2004-09-25, last updated: 2012-06-12, last checked: 2013-08-27.

[YLDM97] Huifang Yin, Zemen Lebne-Dengel, and Yashwant K. Malaiya. Automatic test generation using checkpoint encoding and antirandom testing. *Int. Symp. on Software Reliability Engineering*, pages 84–95, 1997.

[ZKVM12] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. *The 8th International Conference on emerging Networking Experiments and Technologies (CoNEXT 2012), Nice, France*, 2012.

[ZSBE11] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. *ISSTA 2011, Proceedings of the 2011 International Symposium on Software Testing and Analysis, Toronto, Canada*, pages 353–363, July 2011.

A. Launcher Reliability

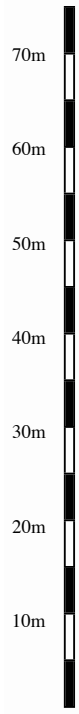
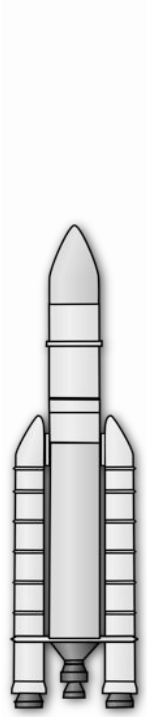
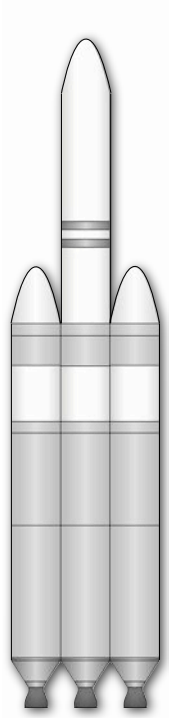
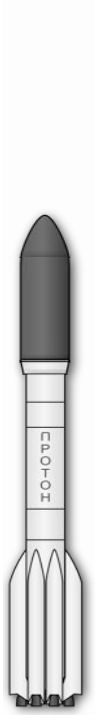
 (aprox.)	 [Com12] Ariane 5	 [Hru08] Delta IV Heavy	 [Gra08, Kol11] Proton M
Launches	70	6	70
Success	66	5	62
Partial Failure	2	1	6
Failure	2	0	2
Reliability Rate	95.7%	91.7%	90.0%

Table A.1.: Launcher reliability as of July 2013 (partial failures counted as ½) [Bla13a, Bla13b, Bla13c, ESA13a, ESA13b, ESA13c, Kyl13, McC09].

B. Explanation of Variables in the Motivational Example

The variable mapping of the motivational example (section 1.6) is discussed here. The following is the relevant excerpt of the CNF output file.

```
14 c Variable mapping:
15 c {days_task1 (4bit unsigned int) -> {1}, days_task2 (4bit unsigned int
     ) -> {9}, days (4bit unsigned int) -> {17}, 0 (4bit unsigned int)
     -> {17}, 1 (4bit signed int) -> {83}, 2 (4bit unsigned int) ->
     {93}, 3 (5bit signed int) -> {105}, 4 (5bit signed int) -> {115}, 5
      (5bit signed int) -> {125}}
```

`days_task1`, `days_task2`, and `days` are the parameters and variables as seen in source code. The remaining variables are auxiliary variables.

`0` holds the result of the operation `days_task1 + days_task2` – the result which gets consecutively assigned to days.

`1` holds the value of the numeric 7 given in the *if*-statement. Based on the bitlength required for 7 in binary, the type of the variable gets chosen as signed int (here 4bit).

But the days variable which it is to be compared to is a *signed* integer. Therefore the variable `2` gets introduced, it hold the value of 1 converted to signed integer.

The comparison is done by subtracting 7 from `days` and check if this is greater 0. To be able to take values below zero (instead of causing an overflow), variable `3` and `4` are being introduced as 5bit signed integers. `days` and `2` get converted to these types.

`5` holds the result of the subtraction of `3` and `4`.

C. Example Truth Table for Boolean Constraints

See next page.

a	b	c	d	e	$F(a,b,c,d,e)$	Clauses
0	0	0	0	0	1	
0	0	0	0	1	1	
0	0	0	1	0	1	
0	0	0	1	1	1	
0	0	1	0	0	0	$a \vee b \vee \neg c \vee d \vee e$
0	0	1	0	1	0	$a \vee b \vee \neg c \vee d \vee \neg e$
0	0	1	1	0	0	$a \vee b \vee \neg c \vee \neg d \vee e$
0	0	1	1	1	0	$a \vee b \vee \neg c \vee \neg d \vee \neg e$
0	1	0	0	0	1	
0	1	0	0	1	0	$a \vee \neg b \vee c \vee d \vee \neg e$
0	1	0	1	0	1	
0	1	0	1	1	1	
0	1	1	0	0	1	
0	1	1	0	1	1	
0	1	1	1	0	1	
0	1	1	1	1	1	
1	0	0	0	0	1	
1	0	0	0	1	1	
1	0	0	1	0	1	
1	0	0	1	1	1	
1	0	1	0	0	1	
1	0	1	0	1	1	
1	0	1	1	0	1	
1	0	1	1	1	1	
1	1	0	0	0	1	
1	1	0	0	1	0	$\neg a \vee \neg b \vee c \vee d \vee \neg e$
1	1	0	1	0	1	
1	1	0	1	1	1	
1	1	1	0	0	1	
1	1	1	0	1	1	
1	1	1	1	0	1	
1	1	1	1	1	1	

Table C.1.: Example for deriving CNF from truth table. Full version of table 2.2, section 2.2.1.

D. Seminar Paper on Pseudo-Boolean Constraints

See next page.

Pseudo-Boolean Constraints

Stefan Weiler

***Abstract*—This paper focusses on Pseudo-Boolean Constraints in the context of Boolean Satisfiability / SAT Solving. The Paper will introduce to Pseudo-Boolean Functions and Constraints, show the relation to SAT with its similarities, differences and ways to translate them in both directions. It is also intended to view the usefulness and application of Pseudo-Boolean Constraints.**

At first basic types of Pseudo-Boolean Constraints and the two basic operations of Linearization and Normalization are explained. Then it is shown how Pseudo-Boolean Constraints are handled in Computer Systems. For solving Pseudo-Boolean Constraints a special focus is given to the Translation to SAT, according to the SAT context of this paper. A model of a Production Planning Process encoded as an example in Pseudo-Boolean Constraints will at the end show how real life problems can easily be solved with this method.

I. Introduction

PSEUDO-BOOLEAN – this term already describes well the nature of Pseudo-Boolean functions. While not being Boolean functions, they are quite similar. Remaining close to Boolean functions, Pseudo-Boolean function do benefit from the advances achieved in SAT solving. On the other hand they are said to be a more expressive and natural way to work on real-life problems.

Pseudo-Boolean functions are studied since the mid 1960s. Today they are used in many fields, such as Operations Research, Graph Theory, Combinatorial Mathematics, Computer Science, VLSI Design, Economics, and Manufacturing, as numerous problems in those fields can be expressed as Pseudo-Boolean functions.

Darmstadt
June 29, 2012

II. Basic Definitions

Mathematically the Handbook of Satisfiability describes a Pseudo-Boolean functions in its broadest sense as: a function that maps n Boolean values to a real number[5, p. 695].

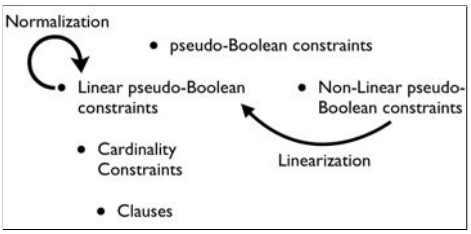
$$x_1, x_2, x_3, ..., x_n \Rightarrow \mathbb{R}$$

Integer numbers are used instead of Real numbers, as a computational restriction, but unlimited precision of Real numbers would not be required.

$$x_1, x_2, x_3, ..., x_n \Rightarrow \mathbb{Z} \quad \longrightarrow \quad x_1, x_2, x_3, ..., x_n \Rightarrow \mathbb{Z}_0^+$$

We will later see, that those mappings can be transformed in a way, so that only positive (unsigned) Integers are needed.

In the coming subsections Basic definitions shall be introduced. The following graphic is meant to visualize types and subtypes of Pseudo-Boolean Constraints, as well as the basic operations of Linearization and Normalization.



At first it has to be differentiated between Pseudo-Boolean Constraints and Pseudo-Boolean Functions in general. A Function takes an input variable or a set of input variables and gives a result value, e.g.:

$$y = f(x)$$

A Constraint on the other hand is a condition, stating that the output result of a function shall have a defined value. It can be defined that the output result of the function should be *equal*, *less*, *greater*, *less or equal*, or *greater or equal* to a defined constant value b. This can be denoted in the following way:

$f(x) \triangleright b$
where $\triangleright$ stands for one of $=$, $>$, $<$, $\geq$, or $\leq$

A. Linear Pseudo-Boolean Constraints

A Linear Pseudo-Boolean (LPB) Constraint has the form:

$$\Sigma_j \; a_j l_j \triangleright b$$

where a_j and b are integer constants, l_j are literals, and $\triangleright$ is one of $=$, $>$, $<$, $\geq$, or $\leq$ [5, p. 696]. Examples:

- $3x_1 + 4x_2 + 5x_3 \geq 7$
- $x_1 + x_2 + x_4 < 3$
- $5 + x_1 = 6 - x_2$
- $8x_4 + 4x_3 + 2x_2 + x_1 \leq 8y_4 + 4y_3 + 2y_2 + y_1$

While the later two examples do not meet the form given above, they obviously could be transformed in a few steps.

B. Non-Linear Pseudo-Boolean Constraints

While being the more general term, Non-Linear Pseudo-Boolean Constraints come second in this definitions, because Linear Pseudo-Boolean Constraints are easier to understand as an introduction to Pseudo-Boolean Constraints.

A Non-Linear Pseudo-Boolean Constraint has the form:

$$\Sigma_j \; a_j \Pi_k \; l_{j,k} \triangleright b$$

where a_j and b are integer constants, $l_{j,k}$ are literals, and $\triangleright$ is one of $=$, $>$, $<$, $\geq$, or $\leq$ [5, p. 697].

The obvious difference to LPB constraints is, that now inside the Sum, Products can appear. Examples:

- $7x_1x_2 + 3x_1 + x_3 \geq 8$
- $7x_2 + 3x_1x_3 + 2x_4x_2x_1 \geq 8$

It is possible to transform any Non-Linear Pseudo-Boolean Constraint into a Set of Linear Pseudo-Boolean Constraints. This process called *Linearization* will be described later.

C. Cardinality Constraints

There are three types of Cardinality Constraints:

- $atleast(k, \{x_1, x_2, ..., x_n\})$
- $atmost(k, \{x_1, x_2, ..., x_n\})$
- $exactly(k, \{x_1, x_2, ..., x_n\})$

These constraints can be translated to (Linear) Pseudo-Boolean Constraints, where each $a_j = 1$ and $b = k$. So they can be seen as a special case of Pseudo-Boolean Constraints.

- $atleast(k, \{x_1, x_2, ..., x_n\}) \equiv x_1, x_2, ..., x_n \geq k$
- $atmost(k, \{x_1, x_2, ..., x_n\}) \equiv x_1, x_2, ..., x_n \leq k$
- $exactly(k, \{x_1, x_2, ..., x_n\}) \equiv x_1, x_2, ..., x_n = k$

D. Clauses

A Clause as it may be derived from the Conjunctive Normal Form is like the following:

$$x_1 \vee x_2 \vee ... \vee x_n$$

A Clause is equivalent to an $atleast$ Cardinality Constraint with $k = 1$, and can therefore easily be translated to a Pseudo-Boolean Constraint.

$$x_1 \vee x_2 \vee ... \vee x_n$$
$$\equiv atleast(1, \{x_1, x, ..., x_n\})$$
$$\equiv x_1 + x_2 + ... + x_n \geq 1$$

E. Linearization

Linearization is the process of translating a Non-Linear Pseudo-Boolean Constraint into a Set of Linear Pseudo-Boolean Constraints. A method to do this translation is based on the work of R. Fortet[2].

Each product is to be substituted by a newly introduced variable.

$$l_1 \times l_2 ... \times l_n \Leftrightarrow v$$

Two additional Constraints are introduced, to asure the new variable behaves equivalent to the product.

$$1l_1 + 1l_2 ... + 1l_n - nv \geq 0$$
$$1l_1 + 1l_2 ... + 1l_n + 1v \geq 1$$

These constraints enforce bidirectional:

- one or more $l = 0 \Leftrightarrow v = 0$,
- all $l = 1 \Leftrightarrow v = 1$,

F. Normalization

The task of Normalization is it to bring a Linear Pseudo-Boolean Constraint to a Normal Form, called the Posiform. The Posiform is the form used as the input form for further steps. To be in Posiform, a Constraint has to meet the following form:

$$\Sigma_j \; a_j l_j \geq b, \qquad a_j, b \in \mathbb{N}_0^+$$

In difference to a Linear Pseudo-Boolean Constraint in general, the comparator must be $\geq$, and all a_j and b must be positive Integers or zero[5, p. 698].

The Transformation to Posiform can be done in two steps, dealing with the two restrictions named above.

1) Getting $\geq$
   - $... > b \;\equiv\; ... \geq b + 1$
   - $... < b \;\equiv\; ... \leq b - 1$
   - $... = b \;\equiv\; ... \leq b, \;\; ... \geq b$
   - $... \leq b \;\equiv\; -1 \times (...) \geq -b$
2) Getting $a_j, b \in \mathbb{N}_0^+$
   - $(-a_j) \times x_j \geq b$
     $\equiv (-a_j) \times (1 - \neg x_j) \geq b$
     $\equiv -a_j + a_j \neg x_j \geq b$
     $\equiv a_j \neg x_j \geq b + a$
   - $(-a_j) \times \neg x_j \geq b$
     $\equiv (-a_j) \times (1 - x_j) \geq b$
     $\equiv -a_j + a x_j \geq b$
     $\equiv a_j x_j \geq b + a$

From the Posiform it is already possible to clearly see some special cases:

- If still $b \leq 0$, the solution is trivial, as left-hand side is at least 0[5, p. 698].This is called a Tautology [6, p. 19].
- On the opposite extreme, if $\Sigma_j a_j > b$ the constraint is unsatisfiable, as even in case of all literals x_j being 1 their sum remains smaller than b. Moreover the set of constraints as a whole is unsatisfiable as well, because a set M of formulas is satisfiable if and only if each subset of M is satisfiable [6, p. 34], and a single constraint is the smallest possible subset of M.

III. Implementing Pseudo-Boolean Constraints

To handle Pseudo-Boolean Constraints with a computer, a standard notation is used. A header contains the information on the number of variables and constraints. For Non-Linear Pseudo-Boolean Constraints, the number of products and the over all size of products is also given. The notation looks like shown in this example:

Listing 1. Beispielcode

```
1 * #variable= 3 #constraint= 1 #product= 1
2                              #sizeproduct= 2
3 -6 x1 x2 7 ~x1 >= 2 ;
```

Two literals without a number between them are a product. A tilde denotes a negation.

A command line application was written during the work on this paper, using this in- and output format. It provides the functionality of Linearization and Normalization, as well as the basic checks that can be done with a constraint in Posiform as described earlier.

As the Posiform is the input format for many following operations, the application was of very good use.

IV. Solving Pseudo-Boolean Constraints

For solving Pseudo Boolean Constraints, two problems are to be differentiated:

- Decision Problem
- Optimization Problem

The Decision Problem asks if there exists a solution, so if the constraints are at all satisfiable. The Optimization Problem is to find out what is the one best solution. This optimal solution is defined by minimizing a Cost Function subject to a set of constraints. The Cost Function is a function, assigning a specified cost to each literal.

The Handbook of Satisfiability[5, p. 710ff] discusses several approaches to solving Pseudo-Boolean Constraints:

- Boolean Constraint Propagation
- Conflict Analysis
- Unique Implication Points
- Generalizing Conflict Analysis
- Optimization and Lower Bounding
- Cutting Planes
- Translation to SAT

As this paper is set in the context of Boolean Satisfiability (SAT), we will have a closer look on the Translation to SAT approach.

A. Translation to SAT

> "Pseudo-Boolean Constraints can be translated to SAT and surprisingly this approach can be quite efficient as demonstrated by the minisat+ solver [1] in the several evaluations of Pseudo-Boolean solvers [3]. [...] A naive approach is to translate a PB constraint as a set of clauses which is semantically equivalent and uses the same variables"[5, p. 726].

1) Truth Table: It is easily to be seen that a representation in clauses must exist for a given Pseudo-Boolean Constraint, by looking at the truth table. For a Pseudo-Boolean constraint formula F, a set of equivalent Clauses can be obtained by deriving the Conjunctive Normal Form (CNF) from the truth table of F. In this case the CNF formula can, in a manner of speaking, be directly read from the table. To get a CNF formula equivalent to F, one has to do the following steps: Each line of the truth table with a truth value of 0 is contributing as a disjunctive clause to the CNF formula. The literals of one of these clauses can be derived in the following way: If the Value of A_i in the corresponding Line is 0, a_i is used as a Literal, if not $\neg a_i$ is used [6, p. 28].

As an example for this method, let $3x_1 + 4\neg x_2 + 5x_3 \geq 7$ denote the constraint F. The truth table is given in Table 1.

x_1	x_2	x_3	$f(x_1, x_2, x_3)$	$f(x_1, x_2, x_3) \geq 7$
0	0	0	$0+4+0=4$	0
0	0	1	$0+4+5=9$	1
0	1	0	$0+0+0=0$	0
0	1	1	$0+0+5=5$	0
1	0	0	$3+4+0=7$	1
1	0	1	$3+4+5=12$	1
1	1	0	$3+0+0=3$	0
1	1	1	$3+0+5=8$	1

TABLE I
EXAMPLE TRUTH TABLE FOR $3x_1 + 4\neg x_2 + 5x_3 \geq 7$

The CNF can be derived as following:
$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$

Then the set of equivalent clauses can be denoted as:
$x_1 \vee x_2 \vee x_3$
$x_1 \vee \neg x_2 \vee x_3$
$x_1 \vee \neg x_2 \vee \neg x_3$
$\neg x_1 \vee \neg x_2 \vee x_3$

From the example it can be seen, that using the shown method a single Pseudo-Boolean constraint in the worst case (all lines of the truth table have to be used) leads to a set of 2^n equivalent clauses with n variables each (letting n denote the number of variables in the Pseudo-Boolean constraint). In fact it can be reduced to $2^n - 1$, as a Pseudo-Boolean Constraint with n zeros and without a single 1 as truth values is known to be unsatisfiable. But for large n this does not really make a difference.

> "Pseudo-Boolean Constraints are more expressive than clauses and a single Pseudo-Boolean constraint may replace an exponential number of clauses. More precisely, the translation of a non-trivial Pseudo-Boolean constraint to a set of clauses which is semantically equivalent (in the usual sense, i.e. without introducing extra variables) has an exponential complexity in the worst case"[5, p. 701].

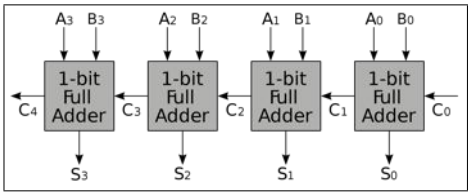
One has to keep in mind, that the clauses of the CNF formula, which were derived in the way described above, are not necessarily the shortest possible representation [6, p. 29].

2) Binary Adder Method: As using the truth table can hardly be regarded a practible approach, other methods have to be used. There are different approaches to encode Pseudo-Boolean Constraints in Clauses.

- Binary Adder
- Binary Decision Diagram
- Unary Notation

Efficient solvers choose appropriate encoding for each constraint individually Exemplarily the approach using binary adders[7] to linear-time transformation into clauses will be explained here.

To use this method, the constraint needs to be in Posiform.



A binary adder, as seen above[8], can be described with the following formulas[9]:

$c_{n+1} = (a_n \wedge c_n) \vee (b_n \wedge c_n) \vee (a_n \wedge b_n)$
$s_n = a_n \ XOR \ b_n \ XOR \ c_n$

In the binary adder method these formulas are being used. The binary representation of all a_j is taken. Instead ones, the corresponding literal x_j is used, while zeros remain zeros. Adding is done iteratively using the previous partial sum and the next number. The final sum is compared to b.

The following example shall help to explain the process:

Pseudo-Boolean Constraint given:
$3x_1 + 4\neg x_2 + 5x_3 \geq 7$

The binary representations of the numbers are:
$a_1 = 3_{10} = 0011_2$
$a_2 = 4_{10} = 0100_2$
$a_3 = 5_{10} = 0101_2$
$b = 7_{10} = 0111_2$

Results of the addition (simplified):
$S_1(a_1, a_2) = 0, \neg x_2, x_1, x_1$
$S_1(S_1, a_3) = (\neg x_2 \wedge x_3) \vee (x_1 \wedge x_3), (\neg x_2 \ XOR \ x_3) \ XOR \ (x_1 \wedge x_3), x_1 \ XOR \ (x_1 \wedge x_3), x_1 \ XOR \ x_3$

To achieve a linear runtime, additional variables equivalent to the given bit-representation would be introduced in real systems.

The final sum can now be compared to $b = 0111_2$. To be greater or equal to b, either the first digit or the 3 later digits must be 1. The constraint can therefore be written as:

$((\neg x_2 \wedge x_3) \vee (x_1 \wedge x_3)) \vee (((\neg x_2 \ XOR \ x_3) \ XOR \ (x_1 \wedge x_3)) \wedge (x_1 \ XOR \ (x_1 \wedge x_3)) \wedge (x_1 \ XOR \ x_3))$

3) Optimization Problem with SAT: The cost function as already explained, is something special when working Pseudo-Boolean Constraint. When translating the Constraints to SAT, it is necessary to find some corresponding approach to the Cost Function expressed in SAT.

Basically this is done by introducing additional constraints iteratively, and translating them to SAT. The options available are linear search or binary search.

- For linear search the cost function is replaced by one new constraint setting the const function $F \leq b$, starting with a maximal value for b and reducing it iteratively by 1.
- For binary search two constraints are additionally introduced $F \leq U$ (upper limit) and $F \geq L$ (lower limit).

Binary search is faster in giving the optimal result, while linear search is giving a (not optimal) result earlier.

V. Expressing a (simplified) real-life example problem as Pseudo-Boolean Constraints

The Handbook of Satisfiability[5, p. 695] names the following as some fields of application:

- Operations Research
- Graph Theory
- Combinatorial Mathematics
- Computer Science
- VLSI Design
- Economics
- Manufacturing

To show an example we will use the field of manufacturing. The Example is from the lecture on Logistical Applications and Optimizations [4, p. 30]. Its a simplified production planning problem, and a simliar task has to be solved manually by Master level students in exams. Here it shall be solved by using Pseudo-Boolean Constraints.

A. Task

Production optimization:
You produce 3 materials: B (Black) Y (Yellow) W (White) on only one machine.

- Material B is needed: 200 pc on 1.7., 100 pc on 1.8. Lotsize is 100 in 10 days.
- Material Y is needed: 100 pc on 1.8., 200 pc on 1.9. Lotsize is 200 in 10 days.
- Material W is needed: 100 pc on 1.7., 100 pc on 1.9. Lotsize is 100 in 10 days.

Cleaning time W after B is 10 days, W after Y is 10 days, Y after B is 10 days. Cleaning time B after W or Y is 0 days.
What is your production plan?
(To simplify calculations, each month is considered to be split in 3 time periods of 10 days each.)

B. Constraints

First lets introduce variables for the production of each material or cleaning in each time period n:

b_n, y_n, w_n, c_n.

As there is only one machine that can be used for one task at a time, $b_n + y_n + w_n + c_n = 1$ must hold true for each n.

The main formulas are those for the wanted production:

- $100b_1 + 100b_2 + 100b_3 \geq 200$
- $100b_1 + 100b_2 + 100b_3 - 200 + 100b_4 + 100b_5 + 100b_6 \geq 100$
- $200y_1 + 200y_2 + 200y_3 + 200y_4 + 200y_5 + 200y_6 \geq 100$
- $200y_1 + 200y_2 + 200y_3 + 200y_4 + 200y_5 + 200y_6 - 100 + 200y_7 + 200y_8 + 200y_9 \geq 200$
- $100w_1 + 100w_2 + 100w_3 \geq 100$
- $100w_1 + 100w_2 + 100w_3 - 100 + 100w_4 + 100w_5 + 100w_6 + 100w_7 + 100w_8 + 100w_9 \geq 100$

As production of white or yellow directly after black or white after yellow is not permitted, another set of constraints has to enforce the cleaning process. So for example at most one thing can be done, either producing black in this time period or producing white in the next time period.

- $atmost(1, \{b_n, w_{n+1}\}) \equiv b_n + w_{n+1} \leq 1$
- $atmost(1, \{b_n, y_{n+1}\}) \equiv b_n + y_{n+1} \leq 1$
- $atmost(1, \{y_n, w_{n+1}\}) \equiv y_n + w_{n+1} \leq 1$

The following rules are necessary for not finding inefficient solutions. Inefficient in this case means unnecessary cleaning of the machine.

- $atmost(1, \{c_n, c_{n+1}\}) \equiv c_n + c_{n+1} \leq 1$
  (do not clean two times with no production between)
- $atmost(1, \{w_n, c_{n+1}\}) \equiv w_n + c_{n+1} \leq 1$
  (do not clean after white production, never necessary)
- $atmost(1, \{c_n, b_{n+1}\}) \equiv c_n + b_{n+1} \leq 1$
  (do not clean before black production, never necessary)

Inefficiency could also be seen in producing more than needed, but in our example this can not happen. Instead of introducing these additional constraints, one might also use a cost function, solving the example as an optimization problem rather than a decision problem.

C. Result

The constraints are satisfied, when:

$$
\begin{array}{llll}
b_1 = 0, & y_1 = 0, & w_1 = 1, & c_1 = 0, \\
b_2 = 1, & y_2 = 0, & w_2 = 0, & c_2 = 0, \\
b_3 = 1, & y_3 = 0, & w_3 = 0, & c_3 = 0, \\
b_4 = 0, & y_4 = 0, & w_4 = 0, & c_4 = 1, \\
b_5 = 0, & y_5 = 1, & w_5 = 0, & c_5 = 0, \\
b_6 = 1, & y_6 = 0, & w_6 = 0, & c_6 = 0, \\
b_7 = 0, & y_7 = 0, & w_7 = 0, & c_7 = 1, \\
b_8 = 0, & y_8 = 0, & w_8 = 1, & c_8 = 0, \\
b_9 = 0, & y_9 = 1, & w_9 = 0, & c_9 = 0
\end{array}
$$

It is therefore possible to produce the wanted amounts of product Black, Yellow and White in the given time.

The following table shows the production plan derived from this values:

Start	Type	End	Amount
1.6.	Product W	10.6.	100
10.6.	Product B	20.6.	100
20.6.	Product B	1.7.	100
1.7.	Cleaning	10.7.	
10.7.	Product Y	20.7.	200
20.7.	Product B	1.8.	100
1.8.	Cleaning	10.8.	
10.8.	Product W	20.8.	100
20.8.	Product Y	1.9.	200

TABLE II
PRODUCTION PLAN

D. Translation to SAT

The Constraints were translated to SAT, using truth tables and the CNF. Seven distinctive truth tables with up to 512 rows were needed for this. A set of 204 Clauses was derived, expressing the same production planing problem as given above. As no additional variables were introduced, the number of variables is the same as in the set of Pseudo-Boolean Constraints: 36.

VI. CONCLUSION

Pseudo-Boolean Constraints are a powerful tool. Many real life problems can easily be encoded in Pseudo-Boolean Constraints. Encoding the same problems in SAT Clauses can be much more difficult. In Pseudo-Boolean Constraints they can be expressed more naturally and in a human readable form.

Pseudo-Boolean Constraints remain close to SAT and can therefore benefit from the advances in SAT Solving. In context of SAT solving it is also good to know, that there are efficient ways to translate Pseudo-Boolean Constraints to SAT Clauses and vice-versa. Clauses can just be written in Pseudo-Boolean Constraints without further work. From Pseudo-Boolean Constraints to Clauses it needs more work, but several methods are available as a choice for the implementation of automatic translation.

When dealing with Boolean Satisfiability, Pseudo-Boolean Constraints should be kept in mind as an alternative. Working with Clauses does not exclude working with Pseudo-Boolean Constraints. They can also be very useful when they are just applied at specific points in a model consisting mainly of Clauses.

REFERENCES

[1] N. Eén and N. Sörensson, *Translating Pseudo-Boolean Constraints into SAT* Journal on Satisfiability, Boolean Modeling and Computation, 2:226, 2006.

[2] R. Fortet, *Application de l'algèbre de Boole en recherche opérationelle* Revue Française de Recherche Opérationelle, 4:1726, 1960.

[3] V. Manquinho and O. Roussel, *Pseudo-Boolean Evaluation 2007* http://www.cril.univ-artois.fr/PB07, 2007.

[4] M. Mekschrat 2011, *Logistical Applications and Optimisations-Functionality of Production Processes* https://www.fbi.h-da.de/filead min/personal/m.mekschrat/LAO_lecture/SS2012/overview2012/FH_Lect ure_Part_4_Overview_e.ppt

[5] O. Roussel and V. Manquinho, *Pseudo-Boolean and Cardinality Constraints* in: A. Bierem M. Heule, H. van Maaren and T. Walsh (Eds.) *Handbook of Satisfiability*, chapter 22, IOS Press, 2009.

[6] U. Schöning, *Logik für Informatiker*, 5th ed. Spektrum Heidelberg-Berlin, 2000.

[7] J. P. Warners, *A linear-time transformation of linear inequalities into conjunctive normal form* Information Processing Letters, 68(2):63 69, 1998.

[8] Wikimedia Commons, *4-Bit Ripple Carry Adder* http://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg

[9] Wikipedia.de, *Volladdierer* http://de.wikipedia.org/wiki/Volladdierer

E. Tables for Approach on Addition

The tables shown here are the complete version of the truncated table 3.13, respectively table 3.12.

E.1. Half Adder

Inputs		Outputs		Valid?	Clauses
a	b	s	c		
0	0	0	0	1	
0	0	0	1	0	$a \vee b \vee s \vee \neg c$
0	0	1	0	0	$a \vee b \vee \neg s \vee c$
0	0	1	1	0	$a \vee b \vee \neg s \vee \neg c$
0	1	0	0	0	$a \vee \neg b \vee s \vee c$
0	1	0	1	0	$a \vee \neg b \vee s \vee \neg c$
0	1	1	0	1	
0	1	1	1	0	$a \vee \neg b \vee \neg s \vee \neg c$
1	0	0	0	0	$\neg a \vee b \vee s \vee c$
1	0	0	1	0	$\neg a \vee b \vee s \vee \neg c$
1	0	1	0	1	
1	0	1	1	0	$\neg a \vee b \vee \neg s \vee \neg c$
1	1	0	0	0	$\neg a \vee \neg b \vee s \vee c$
1	1	0	1	1	
1	1	1	0	0	$\neg a \vee \neg b \vee \neg s \vee c$
1	1	1	1	0	$\neg a \vee \neg b \vee \neg s \vee \neg c$

$\left.\begin{array}{l} a \vee b \vee \neg s \vee c \\ a \vee b \vee \neg s \vee \neg c \end{array}\right\} a \vee b \vee \neg s$

$\left.\begin{array}{l} a \vee \neg b \vee s \vee c \\ a \vee \neg b \vee s \vee \neg c \end{array}\right\} a \vee \neg b \vee s$

$\left.\begin{array}{l} \neg a \vee b \vee s \vee c \\ \neg a \vee b \vee s \vee \neg c \end{array}\right\} \neg a \vee b \vee s$

$\left.\begin{array}{l} \neg a \vee \neg b \vee \neg s \vee c \\ \neg a \vee \neg b \vee \neg s \vee \neg c \end{array}\right\} \neg a \vee \neg b \vee \neg s$

Table E.1.: Half-Adder approach to addition modeling – Extended truth table, derived clauses, and reduced set of clauses.

E.2. Full Adder

Inputs			Outputs		Valid?	Clauses
a	b	c_{in}	c_{out}	s		
0	0	0	0	0	1	
0	0	0	0	1	0	$a \vee b \vee c_{in} \vee c_{out} \vee \neg s$
0	0	0	1	0	0	$a \vee b \vee c_{in} \vee \neg c_{out} \vee s$
0	0	0	1	1	0	$a \vee b \vee c_{in} \vee \neg c_{out} \vee \neg s$
0	0	1	0	0	0	$a \vee b \vee \neg c_{in} \vee c_{out} \vee s$
0	0	1	0	1	1	
0	0	1	1	0	0	$a \vee b \vee \neg c_{in} \vee \neg c_{out} \vee s$
0	0	1	1	1	0	$a \vee b \vee \neg c_{in} \vee \neg c_{out} \vee \neg s$
0	1	0	0	0	0	$a \vee \neg b \vee c_{in} \vee c_{out} \vee s$
0	1	0	0	1	1	
0	1	0	1	0	0	$a \vee \neg b \vee c_{in} \vee \neg c_{out} \vee s$
0	1	0	1	1	0	$a \vee \neg b \vee c_{in} \vee \neg c_{out} \vee \neg s$
0	1	1	0	0	0	$a \vee \neg b \vee \neg c_{in} \vee \vee c_{out} \vee s$
0	1	1	0	1	0	$a \vee \neg b \vee \neg c_{in} \vee c_{out} \vee \neg s$
0	1	1	1	0	1	
0	1	1	1	1	0	$a \vee \neg b \vee \neg c_{in} \vee \neg c_{out} \vee \neg s$
1	0	0	0	0	0	$\neg a \vee b \vee c_{in} \vee c_{out} \vee s$
1	0	0	0	1	1	
1	0	0	1	0	0	$\neg a \vee b \vee c_{in} \vee \neg c_{out} \vee s$
1	0	0	1	1	0	$\neg a \vee b \vee c_{in} \vee \neg c_{out} \vee \neg s$
1	0	1	0	0	0	$\neg a \vee b \vee \neg c_{in} \vee c_{out} \vee s$
1	0	1	0	1	0	$\neg a \vee b \vee \neg c_{in} \vee c_{out} \vee \neg s$
1	0	1	1	0	1	
1	0	1	1	1	0	$\neg a \vee b \vee \neg c_{in} \vee \neg c_{out} \vee \neg s$
1	1	0	0	0	0	$\neg a \vee \neg b \vee c_{in} \vee c_{out} \vee s$
1	1	0	0	1	0	$\neg a \vee \neg b \vee c_{in} \vee c_{out} \vee \neg s$
1	1	0	1	0	1	
1	1	0	1	1	0	$\neg a \vee \neg b \vee c_{in} \vee \neg c_{out} \vee \neg s$
1	1	1	0	0	0	$\neg a \vee \neg b \vee \neg c_{in} \vee c_{out} \vee s$
1	1	1	0	1	0	$\neg a \vee \neg b \vee \neg c_{in} \vee c_{out} \vee \neg s$
1	1	1	1	0	0	$\neg a \vee \neg b \vee \neg c_{in} \vee \neg c_{out} \vee s$
1	1	1	1	1	1	

$\}\, a \vee b \vee c_{in} \vee \neg c_{out}$

$\}\, a \vee b \vee \neg c_{in} \vee \neg c_{out}$

$\}\, a \vee \neg b \vee c_{in} \vee \neg c_{out}$

$\}\, a \vee \neg b \vee \neg c_{in} \vee \vee c_{out}$

$\}\, \neg a \vee b \vee c_{in} \vee \neg c_{out}$

$\}\, \neg a \vee b \vee \neg c_{in} \vee c_{out}$

$\}\, \neg a \vee \neg b \vee c_{in} \vee c_{out}$

$\}\, \neg a \vee \neg b \vee \neg c_{in} \vee c_{out}$

Table E.2.: Full-Adder approach to addition modeling – Extended truth table, derived clauses, and reduced set of clauses.

F. Example for MC/DC constraints

$$
\begin{aligned}
&\{ \\
&\quad [\\
&\qquad \neg((c_{true1,1,1} \leftrightarrow c_{true1,1,2}) \wedge (c_{false1,1,1} \leftrightarrow c_{false1,1,2})) \\
&\qquad \wedge((c_{true1,2,1} \leftrightarrow c_{true1,2,2}) \wedge (c_{false1,2,1} \leftrightarrow c_{false1,2,2})) \\
&\qquad \wedge((c_{true1,3,1} \leftrightarrow c_{true1,3,2}) \wedge (c_{false1,3,1} \leftrightarrow c_{false1,3,2})) \\
&\qquad \wedge((c_{true1,4,1} \leftrightarrow c_{true1,4,2}) \wedge (c_{false1,4,1} \leftrightarrow c_{false1,4,2})) \\
&\qquad \wedge\neg((d_{if1,1} \leftrightarrow d_{if1,2}) \wedge (d_{else1,1} \leftrightarrow d_{else1,2})) \\
&\quad] \vee [\\
&\qquad \neg((c_{true1,1,1} \leftrightarrow c_{true1,1,3}) \wedge (c_{false1,1,1} \leftrightarrow c_{false1,1,3})) \\
&\qquad \wedge((c_{true1,2,1} \leftrightarrow c_{true1,2,3}) \wedge (c_{false1,2,1} \leftrightarrow c_{false1,2,3})) \\
&\qquad \wedge((c_{true1,3,1} \leftrightarrow c_{true1,3,3}) \wedge (c_{false1,3,1} \leftrightarrow c_{false1,3,3})) \\
&\qquad \wedge((c_{true1,4,1} \leftrightarrow c_{true1,4,3}) \wedge (c_{false1,4,1} \leftrightarrow c_{false1,4,3})) \\
&\qquad \wedge\neg((d_{if1,1} \leftrightarrow d_{if1,3}) \wedge (d_{else1,1} \leftrightarrow d_{else1,3})) \\
&\quad] \vee [\\
&\qquad \neg((c_{true1,1,2} \leftrightarrow c_{true1,1,3}) \wedge (c_{false1,1,2} \leftrightarrow c_{false1,1,3})) \\
&\qquad \wedge((c_{true1,2,2} \leftrightarrow c_{true1,2,3}) \wedge (c_{false1,2,2} \leftrightarrow c_{false1,2,3})) \\
&\qquad \wedge((c_{true1,3,2} \leftrightarrow c_{true1,3,3}) \wedge (c_{false1,3,2} \leftrightarrow c_{false1,3,3})) \\
&\qquad \wedge((c_{true1,4,2} \leftrightarrow c_{true1,4,3}) \wedge (c_{false1,4,2} \leftrightarrow c_{false1,4,3})) \\
&\qquad \wedge\neg((d_{if1,2} \leftrightarrow d_{if1,3}) \wedge (d_{else1,2} \leftrightarrow d_{else1,3})) \\
&\quad] \\
&\} \wedge \{ \\
&\quad [
\end{aligned}
$$

$$
\begin{aligned}
&((c_{true1,1,1} \leftrightarrow c_{true1,1,2}) \wedge (c_{false1,1,1} \leftrightarrow c_{false1,1,2}))\\
&\wedge\neg((c_{true1,2,1} \leftrightarrow c_{true1,2,2}) \wedge (c_{false1,2,1} \leftrightarrow c_{false1,2,2}))\\
&\wedge((c_{true1,3,1} \leftrightarrow c_{true1,3,2}) \wedge (c_{false1,3,1} \leftrightarrow c_{false1,3,2}))\\
&\wedge((c_{true1,4,1} \leftrightarrow c_{true1,4,2}) \wedge (c_{false1,4,1} \leftrightarrow c_{false1,4,2}))\\
&\wedge\neg((d_{if1,1} \leftrightarrow d_{if1,2}) \wedge (d_{else1,1} \leftrightarrow d_{else1,2}))\\
]\vee[\\
&((c_{true1,1,1} \leftrightarrow c_{true1,1,3}) \wedge (c_{false1,1,1} \leftrightarrow c_{false1,1,3}))\\
&\wedge\neg((c_{true1,2,1} \leftrightarrow c_{true1,2,3}) \wedge (c_{false1,2,1} \leftrightarrow c_{false1,2,3}))\\
&\wedge((c_{true1,3,1} \leftrightarrow c_{true1,3,3}) \wedge (c_{false1,3,1} \leftrightarrow c_{false1,3,3}))\\
&\wedge((c_{true1,4,1} \leftrightarrow c_{true1,4,3}) \wedge (c_{false1,4,1} \leftrightarrow c_{false1,4,3}))\\
&\wedge\neg((d_{if1,1} \leftrightarrow d_{if1,3}) \wedge (d_{else1,1} \leftrightarrow d_{else1,3}))\\
]\vee[\\
&((c_{true1,1,2} \leftrightarrow c_{true1,1,3}) \wedge (c_{false1,1,2} \leftrightarrow c_{false1,1,3}))\\
&\wedge\neg((c_{true1,2,2} \leftrightarrow c_{true1,2,3}) \wedge (c_{false1,2,2} \leftrightarrow c_{false1,2,3}))\\
&\wedge((c_{true1,3,2} \leftrightarrow c_{true1,3,3}) \wedge (c_{false1,3,2} \leftrightarrow c_{false1,3,3}))\\
&\wedge((c_{true1,4,2} \leftrightarrow c_{true1,4,3}) \wedge (c_{false1,4,2} \leftrightarrow c_{false1,4,3}))\\
&\wedge\neg((d_{if1,2} \leftrightarrow d_{if1,3}) \wedge (d_{else1,2} \leftrightarrow d_{else1,3}))\\
]\\
\}\wedge\{\\
[\\
&((c_{true1,1,1} \leftrightarrow c_{true1,1,2}) \wedge (c_{false1,1,1} \leftrightarrow c_{false1,1,2}))\\
&\wedge((c_{true1,2,1} \leftrightarrow c_{true1,2,2}) \wedge (c_{false1,2,1} \leftrightarrow c_{false1,2,2}))\\
&\wedge\neg((c_{true1,3,1} \leftrightarrow c_{true1,3,2}) \wedge (c_{false1,3,1} \leftrightarrow c_{false1,3,2}))\\
&\wedge((c_{true1,4,1} \leftrightarrow c_{true1,4,2}) \wedge (c_{false1,4,1} \leftrightarrow c_{false1,4,2}))\\
&\wedge\neg((d_{if1,1} \leftrightarrow d_{if1,2}) \wedge (d_{else1,1} \leftrightarrow d_{else1,2}))\\
]\vee[\\
&((c_{true1,1,1} \leftrightarrow c_{true1,1,3}) \wedge (c_{false1,1,1} \leftrightarrow c_{false1,1,3}))\\
&\wedge((c_{true1,2,1} \leftrightarrow c_{true1,2,3}) \wedge (c_{false1,2,1} \leftrightarrow c_{false1,2,3}))\\
&\wedge\neg((c_{true1,3,1} \leftrightarrow c_{true1,3,3}) \wedge (c_{false1,3,1} \leftrightarrow c_{false1,3,3}))\\
&\wedge((c_{true1,4,1} \leftrightarrow c_{true1,4,3}) \wedge (c_{false1,4,1} \leftrightarrow c_{false1,4,3}))
\end{aligned}
$$

$$
\begin{aligned}
&\quad \wedge\neg((d_{if1,1} \leftrightarrow d_{if1,3}) \wedge (d_{else1,1} \leftrightarrow d_{else1,3}))\\
&]\vee[\\
&\quad ((c_{true1,1,2} \leftrightarrow c_{true1,1,3}) \wedge (c_{false1,1,2} \leftrightarrow c_{false1,1,3}))\\
&\quad \wedge((c_{true1,2,2} \leftrightarrow c_{true1,2,3}) \wedge (c_{false1,2,2} \leftrightarrow c_{false1,2,3}))\\
&\quad \wedge\neg((c_{true1,3,2} \leftrightarrow c_{true1,3,3}) \wedge (c_{false1,3,2} \leftrightarrow c_{false1,3,3}))\\
&\quad \wedge((c_{true1,4,2} \leftrightarrow c_{true1,4,3}) \wedge (c_{false1,4,2} \leftrightarrow c_{false1,4,3}))\\
&\quad \wedge\neg((d_{if1,2} \leftrightarrow d_{if1,3}) \wedge (d_{else1,2} \leftrightarrow d_{else1,3}))\\
&]\\
\}\wedge\{&\\
&[\\
&\quad ((c_{true1,1,1} \leftrightarrow c_{true1,1,2}) \wedge (c_{false1,1,1} \leftrightarrow c_{false1,1,2}))\\
&\quad \wedge((c_{true1,2,1} \leftrightarrow c_{true1,2,2}) \wedge (c_{false1,2,1} \leftrightarrow c_{false1,2,2}))\\
&\quad \wedge((c_{true1,3,1} \leftrightarrow c_{true1,3,2}) \wedge (c_{false1,3,1} \leftrightarrow c_{false1,3,2}))\\
&\quad \wedge\neg((c_{true1,4,1} \leftrightarrow c_{true1,4,2}) \wedge (c_{false1,4,1} \leftrightarrow c_{false1,4,2}))\\
&\quad \wedge\neg((d_{if1,1} \leftrightarrow d_{if1,2}) \wedge (d_{else1,1} \leftrightarrow d_{else1,2}))\\
&]\vee[\\
&\quad ((c_{true1,1,1} \leftrightarrow c_{true1,1,3}) \wedge (c_{false1,1,1} \leftrightarrow c_{false1,1,3}))\\
&\quad \wedge((c_{true1,2,1} \leftrightarrow c_{true1,2,3}) \wedge (c_{false1,2,1} \leftrightarrow c_{false1,2,3}))\\
&\quad \wedge((c_{true1,3,1} \leftrightarrow c_{true1,3,3}) \wedge (c_{false1,3,1} \leftrightarrow c_{false1,3,3}))\\
&\quad \wedge\neg((c_{true1,4,1} \leftrightarrow c_{true1,4,3}) \wedge (c_{false1,4,1} \leftrightarrow c_{false1,4,3}))\\
&\quad \wedge\neg((d_{if1,1} \leftrightarrow d_{if1,3}) \wedge (d_{else1,1} \leftrightarrow d_{else1,3}))\\
&]\vee[\\
&\quad ((c_{true1,1,2} \leftrightarrow c_{true1,1,3}) \wedge (c_{false1,1,2} \leftrightarrow c_{false1,1,3}))\\
&\quad \wedge((c_{true1,2,2} \leftrightarrow c_{true1,2,3}) \wedge (c_{false1,2,2} \leftrightarrow c_{false1,2,3}))\\
&\quad \wedge((c_{true1,3,2} \leftrightarrow c_{true1,3,3}) \wedge (c_{false1,3,2} \leftrightarrow c_{false1,3,3}))\\
&\quad \wedge\neg((c_{true1,4,2} \leftrightarrow c_{true1,4,3}) \wedge (c_{false1,4,2} \leftrightarrow c_{false1,4,3}))\\
&\quad \wedge\neg((d_{if1,2} \leftrightarrow d_{if1,3}) \wedge (d_{else1,2} \leftrightarrow d_{else1,3}))\\
&]\\
\}&
\end{aligned}
$$

G. Grammar for the Parser

```
<code>                --> <function> { <function> }

<function>            --> <function_header> <function_body>

<function_body>       --> ‘{‘ <instruction_list> ‘}‘

<function_header>     --> <type> <function_name> ‘(‘ <parameter_list>
                          ‘)‘

<parameter_list>      --> <parameter> { ‘,‘ <parameter> }

<parameter>           --> <type> <variable_name>

<instruction_list>    --> <instruction> { <instruction> }

<instruction>         --> <if>
                      --> <single_instruction> ‘;‘

<if>                  --> ‘if‘ ‘(‘ <test> ‘)‘ ‘{‘ <instruction_list>
                          ‘}‘ [ <else> ‘{‘ <instruction_list> ‘}‘ ]

<test>                --> <disjunct> { ‘||‘ <disjunct> }

<disjucnt>            --> <conjunct> { ‘&&‘ <conjunct> }

<conjunct>            --> ‘(‘ <test> ‘)‘
                      --> ‘!‘ <conjunct>
                      --> <boolean_expression>
```

```
<boolean_expression>  --> <addsub>
                      --> <addsub> ‘<’ <addsub>
                      --> <addsub> ‘<=‘ <addsub>
                      --> <addsub> ‘==‘ <addsub>
                      --> <addsub> ‘>=‘ <addsub>
                      --> <addsub> ‘>‘ <addsub>
                      --> <addsub> ‘!=‘ <addsub>
<value>               --> <variable_name>
                      --> <constant>
                      --> ‘(‘ <addsub> ‘)‘
<single_instruction>  --> <variable_name> ‘=‘ <addsub>
                      --> <variable_name> ‘+=‘ <addsub>
                      --> <variable_name> ‘-=‘ <addsub>
                      --> <variable_name> ‘*=‘ <addsub>
                      --> <variable_name> ‘/=‘ <addsub>
                      --> <variable_name> ‘%=‘ <addsub>
                      --> <type> <variable_name> [ ‘=‘ <addsub> ]
<addsub>              --> <multdiv> { <plusminus> <multdiv> }
<multdiv>             --> <value> { <timesby> <value> }
<type>                --> ‘char‘
                      --> ‘signed‘ ‘char‘
                      --> ‘unsigned‘ ‘char‘
                      --> [‘signed‘] ‘short‘ [‘int‘]
```

```
--> ‘unsigned‘ ‘short‘ [‘int‘]

--> [‘signed‘] ‘int‘

--> ‘unsigned‘ ‘int‘

--> [‘signed‘] ‘long‘ [‘int‘]

--> ‘unsigned‘ ‘long‘ [‘int‘]

--> [‘signed‘] ‘long‘ ‘long‘ [‘int‘]

--> ‘unsigned‘ ‘long‘ ‘long‘ [‘int‘]
```

<variable_name> is a name of a variable (including parameters) uniquely identifying a variable. It has to follow the rules of naming variables in C.

<constant> is a hard coded value. This might be an integer, or a character, and is to be of a structure in compliance to one of these types.

H. Test Cases for Large Example

H.1. Condition Coverage

```
Satisfiable, parameters for test cases as follows:
Test case 1: year = 24704, month = 5, day_of_month = 0
Test case 2: year = 1568, month = 10, day_of_month = 0
Test case 3: year = -30664, month = 8, day_of_month = 0
Test case 4: year = 1568, month = 15, day_of_month = 0
Test case 5: year = 27390, month = 11, day_of_month = 0
Test case 6: year = 1582, month = 1, day_of_month = 0
Test case 7: year = 13872, month = 7, day_of_month = 0
Test case 8: year = 3616, month = -9, day_of_month = 0
Test case 9: year = 16188, month = 3, day_of_month = 0
Test case 10: year = -24716, month = 12, day_of_month = 0
Test case 11: year = 928, month = 4, day_of_month = 0
Test case 12: year = 13856, month = 6, day_of_month = 31
Test case 13: year = 1568, month = 9, day_of_month = 15
```

H.2. Condition/Decision Coverage

```
Satisfiable, parameters for test cases as follows:
Test case 1: year = 10149, month = 12, day_of_month = 5
Test case 2: year = 17776, month = -3, day_of_month = 0
Test case 3: year = -30244, month = 5, day_of_month = 0
Test case 4: year = 1586, month = 3, day_of_month = 5
Test case 5: year = -17037, month = 9, day_of_month = 5
Test case 6: year = 1582, month = 10, day_of_month = 5
Test case 7: year = 4, month = 8, day_of_month = 0
Test case 8: year = -31184, month = 4, day_of_month = 31
Test case 9: year = 26190, month = 11, day_of_month = 5
Test case 10: year = 1572, month = 7, day_of_month = 5
Test case 11: year = 1579, month = 1, day_of_month = 0
```

```
Test case 12: year = 1568, month = 13, day_of_month = 15
Test case 13: year = 18438, month = 6, day_of_month = -32753
```

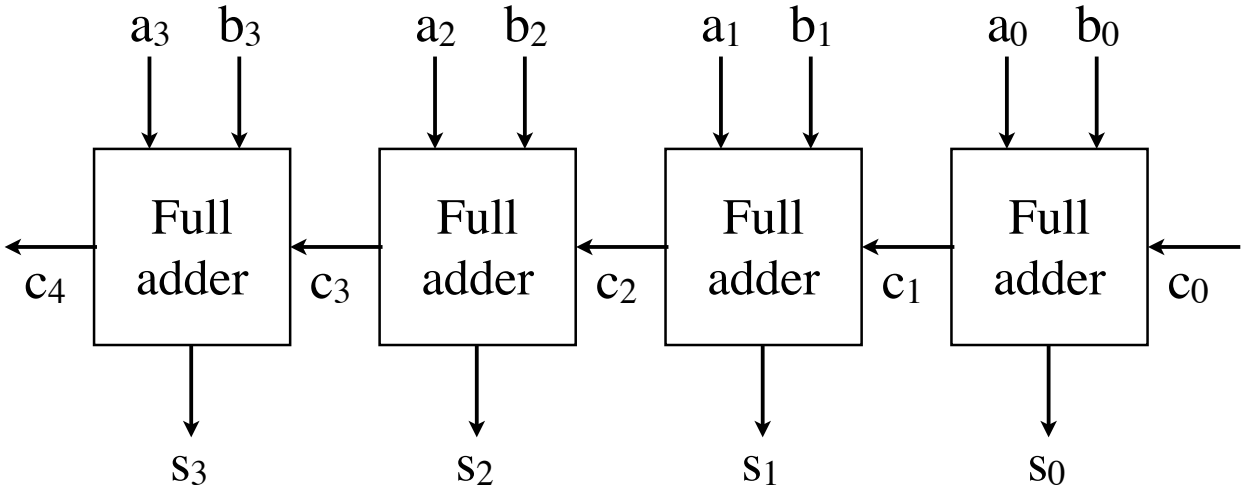
H.3. MC/DC[1]

```
Satisfiable, parameters for test cases as follows:
Test case 1: year = 2004, month = -32767, day_of_month = 4
Test case 2: year = 1582, month = 10, day_of_month = 4
Test case 3: year = 1096, month = 33, day_of_month = 4
Test case 4: year = 1570, month = 6, day_of_month = 0
Test case 5: year = -28000, month = 11, day_of_month = 6
Test case 6: year = 16384, month = 2, day_of_month = -32763
Test case 7: year = 2606, month = 5, day_of_month = 0
Test case 8: year = 1971, month = 4104, day_of_month = 2
Test case 9: year = 1582, month = 10, day_of_month = 2048
Test case 10: year = 7632, month = 524, day_of_month = 0
Test case 11: year = 16901, month = 10, day_of_month = 7
Test case 12: year = 1582, month = 10, day_of_month = 6
Test case 13: year = 3150, month = 3, day_of_month = 1063
Test case 14: year = -31440, month = 1, day_of_month = 3
Test case 15: year = 1582, month = 9, day_of_month = 16386
Test case 16: year = 0, month = 12, day_of_month = 13
Test case 17: year = 1582, month = 8, day_of_month = 12
Test case 18: year = 4032, month = 4, day_of_month = -32690
Test case 19: year = -10686, month = 7, day_of_month = 4
```

[1] *lingeling* (see section 6.2) had to be used instead of PicoSAT as this example reached the limits of PicoSAT (see section 6.3

I. Licenses

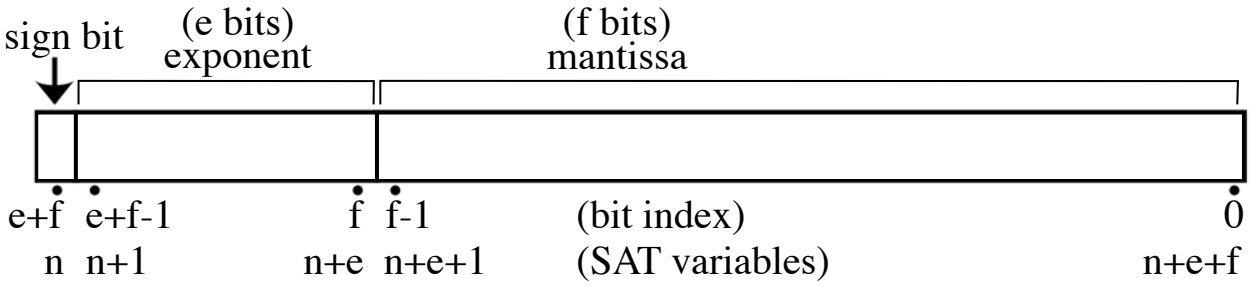
This section serves to comply with the Share-Alike (SA) conditions contained in the licenses of materials used for this thesis. Each licensing refers to an individual picture, and does not apply to any other part of this thesis. For the detailed license conditions, please refer to the links provided.



Picture of a 4-bit ripple carry adder. Derivative work based on [Bur06].

In accordance with the licensing of the material used, I, the author of this thesis, hereby distribute this picture under the same license:

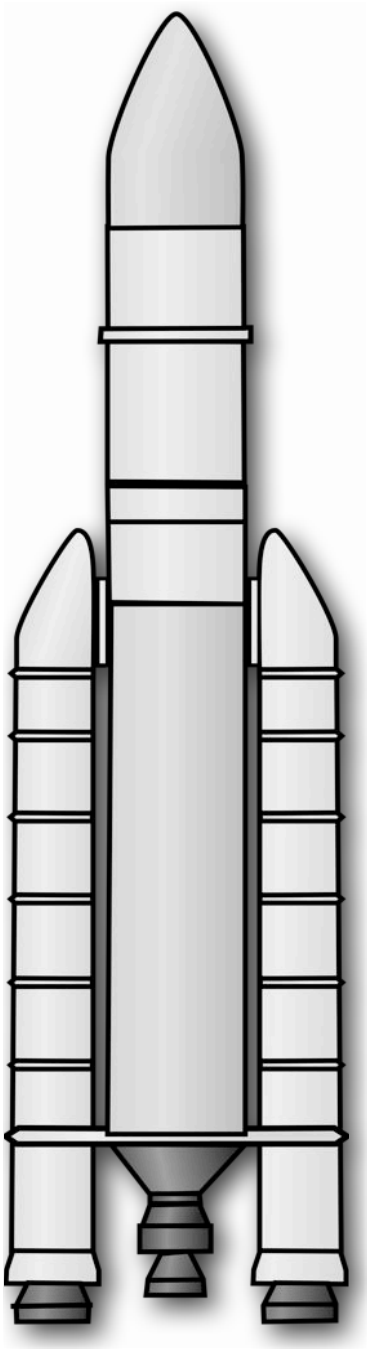
This picture is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license (`http://creativecommons.org/licenses/by-sa/3.0/deed.en`).



Picture of the general binary floating point layout and mapping to SAT-variables. Derivative work based on [Sta07].

In accordance with the licensing of the material used, I, the author of this thesis, hereby distribute this picture under the same license:

This picture is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license (`http://creativecommons.org/licenses/by-sa/3.0/deed.en`).



Picture of Ariane 5. Derivative work based on [Com12].

In accordance with the licensing of the material used, I, the author of this thesis, hereby distribute this picture under the same license:

This picture is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license (`http://creativecommons.org/licenses/by-sa/3.0/deed.en`).

Picture of Proton-M. Derivative work based on [Gra08, Kol11].

In accordance with the licensing of the materials used, I, the author of this thesis, hereby distribute this picture under the same, respectively a compatible license:

This picture is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license (`http://creativecommons.org/licenses/by-sa/3.0/deed.en`).