Institut für Formale Methoden der Informatik
Abteilung Sichere und Zuverlässige Softwaresysteme
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis Nr. 3120

Program Analysis and Probabilistic SAT-solving

Johannes Frederik Jesper Traub

Course of Study:	Information Technology
Examiner:	PD Dr. habil. Dirk Nowotka
Supervisor:	Dipl. inf. Gordon Haak, Daimler AG
Commenced:	10th June 2010
Completed:	7th December 2010
CR-Classification:	F.3.2, G3

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Johannes Frederik Jesper Traub)

Contents

Chapter 1

Introduction

In this thesis two aspects regarding the correctness of hardware and software of computer systems are handled: Program Analysis [22] and Probabilistic SAT-solving [24, 2]. These two topics have become more and more important over the last few years as computer systems have captured an irreplaceable status in the current industry and everyday life. But with the increasing development of computer systems their complexity increases even more. Therefore the proof for the correctness of their hard- and software becomes more and more difficult.

The first challenge of this thesis – residing in the area of Program Analysis – is the transformation of the Intermediate Representation of LLVM [17] describing a possible runtime error of a program into a SMT [11] formula in order to find a witness or to prove the absence of this error.

The second challenge of this thesis – arranged in the domain of Probabilistic SAT-solving – is the implementation and evaluation of algorithms Moser introduced in his papers: "A constructive proof to the Lovász Local Lemma" [19] and "A constructive proof to the general Lovász Local Lemma" [20]. The special interest of the evaluation is the constraint on the structure of the input problem the algorithms are bounded to and how the algorithms scale when the constraint is violated.

Another interest in this thesis is the question whether it is possible to let the two challenges interact with each other and if this interaction leads to an improvement of the design and analysis process in order to build correct hardware and software.

Chapter 2

Foundations

2.1 Program Analysis

In general, (Static) Program Analysis [22] describes the (automatic) analysis of the behavior of programs. The process of Program Analysis on a given program in C code used in this thesis is described in the following: Abstract Interpretation is performed on the program in order to prove the absence of runtime errors (like for example division by zero, overflow, ...).

Definition 1: Abstract Interpretation
Abstract Interpretation [8] is the approach to analyze the semantics of a program by analyzing an abstraction of the concrete semantics. Therefore the concrete semantics of the program is mapped to an abstract semantics. If the abstraction is sound, an analysis result in the abstract semantics also holds in the concrete representation, but might be easier to compute.

An intuitive example given by Sintzoff in [25] showing the rule of signs is illustrated in the following.

Example:
Let `-1515 * 17` *be an instruction. From this instruction the abstraction to the domain of signs* {`+, -`} *can be derived. The abstract operation* `*` *for the abstract values* `+` *and* `-` *is as follows:*

$$\texttt{-(+) * (+)} \Rightarrow \texttt{(-) * (+)} \Rightarrow \texttt{(-)}.$$

This leads to the conclusion, that `-1515 * 17` *will be a negative number without computing its exact value.*

The analysis performed by Abstract Interpretation is restricted to the Halting Problem.

Definition 2: Halting Problem
The halting problem [27] is a decision problem. The question asked is: "Will the program terminate or run forever?" In 1936 Alan Turing proved the non existence of a general algorithm, which is able to solve the halting problem.

This result can be generalized to the theorem of Rice [14].
It is possible to build analysis for abstract semantics, which is not restricted by the theorem of Rice, but for the price of imprecision. This imprecision in the abstraction can lead to spurious errors, which do not occur in the concrete program.

The process of proving the absence or finding witnesses of runtime errors in C source code, which have been detected by Abstract Interpretation, is based on a **Compiler Framework** layout and its steps are illustrated in the following:

1. The LLVM Frontend **Clang** is responsible for parsing the code files and converting them into the **Intermediate Representation** of LLVM.

2. In order to **reduce the problem size** in regard to the possible error techniques like Abstract Interpretation, **Program Slicing** based on **Data Flow Analysis** and so on are used.

3. The resulting portion of code and the error candidate are then transformed into a **Model Checking Problem** and passed to a suitable **Solver**.

4. If the **Solver** finds an assignment to the problem, a witness to the error is found, otherwise the error found by Abstract Interpretation is proven spurious. (The solving process is also restricted by the halting problem).

In the next section a **Compiler Framework** is described in order to get an idea of the hierarchy of Clang and LLVM.

2.1.1 Compiler Framework

A Compiler [1] is the tool used to transform source code into target code. Basically a Compiler consists of three parts: Frontend, Middleend, Backend. The Frontend is the interface between the text file (source code) and the internal code-representation: the Intermediate Representation. The Middleend, which is operating on the Intermediate Representation, is equipped with tools like an Optimizer - based on Data Flow Analysis - in order analyze and optimize the code. Finally the Backend generates target code out of the optimized Intermediate Representation.

2.1.2 Clang

Clang [12] is a frontend for the C language family of LLVM - a Compiler Infrastructure, which is introduced in the next section. Clang is able to process the source

code in several ways. Its main purpose is still the functionality as a Compiler which produces executable target code. Another feature of Clang is the conversion of the input source code into the LLVM Intermediate Representation. The result can either be a text file or a bitcode file, which can both be read by LLVM. An equivalent tool is for the Gnu C Compiler (gcc), but in this thesis Clang is used in order to generate LLVM code. Therefore Clang is invoked either with the parameter "-emit-llvm" in order to create a file in the LLVM text format or with the parameter "-emit-llvm-bc" in order to generate a LLVM bitcode file. In the next section LLVM including its Intermediate Representation is introduced.

2.1.3 Low Level Virtual Machine (LLVM)

LLVM [17] the "low level virtual machine" is a Compiler Infrastructure. It provides several language-dependent frontends like Clang (introduced in [Chapter 2.1.2]), llvm-gcc - the gnu c compiler front end - and so on. It also features a middleend including a collection of tools like an optimizer operating on the intermediate representation, which is introduced in the following section, as well as a number of language-dependent code generators.

2.1.3.1 LLVM Intermediate Representation (IR)

The LLVM Intermediate Representation (IR) consists of a target-independent virtual instruction set and a language-independent typing system.

Virtual Instruction Set
The target-independent virtual instruction set is based on a RISC-like instruction set, without providing architecture specific information like registers, pipelines and calling conventions. Instead the instruction set features an infinite number of virtual registers in Static Single Assignment (SSA) form, introduced in [Definition 3]. The instruction set provides load and store instructions to transfer values between registers and memory. LLVM contains with only 31 instructions a quite small number of instructions, because each instruction is only defined once and able to operate on any operand with any type.

Definition 3: Static Single Assignment (SSA)
Static Single Assignment [3] ensures a unique variable assignment. For each assignment of a variable a new variable is generated. In order to handle assignments, in which a variable depends on a join node, phi nodes are introduced.

Example:

In table 2.1 an example for the transformation from C code to SSA form is shown.

original statement	transformed to SSA
x = 0;	x_0 = 0;
x = x + 1;	x_1 = x_0 + 0;
if (x < 0)	if (x_1 < 0)
y = 3 * x	y_0 = 3 * x_1
else	else
y = x + 1	y_1 = x_1 + 1
z = $\frac{y}{2}$;	y_2 = ϕ(y_0, y_1)
	z_0 = $\frac{y_2}{2}$;

Table 2.1: Transformation from C Source Code to SSA Form

Each variable from the original statement results a new variable in the SSA form on the right hand side. The assignment **z = $\frac{y}{2}$** *results in a phi node in the SSA form, as the value of* **y** *depends either on the if branch (*$\mathbf{y}_1$*) or on the else branch (*$\mathbf{y}_1$*).*

Type System
The type system of LLVM is language-independent in respect to the source language and features the most common types like: void, bool, signed and unsigned integer with a size of 8 to 64 bits and single- and double-precision floating-point types of size 32 to 128 bits. LLVM provides only four derived types: pointer, array, structure and function. As the LLVM type system is language-independent and features no high-level types, source-level types must be lowered to the LLVM type system. This lowering process is based on the idea of lowering source level code to machine code.

Program Structure
The Intermediate Representation of a program in LLVM is provided by the LLVM module - the in-memory representation. A LLVM module is a doubly-linked list identified by the ModuleID. It contains the following items: a list of functions and a list of global variables. Each function inside LLVM contains a list of Basic Blocks (BBs), where each Basic Block contains a list of LLVM instructions. For each source code file a separate module is created.

The following example helps to get an overview of the Intermediate Representation of LLVM. A detailed description of the LLVM language can be found in the "Language Reference Manual" [16].

Example:
In Figure 2.1 a thermostat is defined in C code. It consists of two functions representing the cool and heat functionality, which are triggered in the main function using the boolean variable **heating**. *Initially the temperature (***current***) is set to*

$18°C$ *and the variable* **`heating`** *is set to true - initiating the thermostat to the functionality of a heater. As long as the temperature resides inside the interval* $18 \leq$ **`current`** < 22 *it gets increased by* $0.2°C$ *per iteration. Once* $22°C$ *is passed the variable* **`heating`** *is set to* **`false`***, which puts the thermostat into cooling mode. In this mode the temperature is decreased by* $0.2°C$ *per timestep as long as the temperature stands in the interval* $19 <$ **`current`** ≤ 23*.*

```
double current = 0;
bool heating = true;

void cool() {
  if (current > 19
      && current <= 23)
    current -= 0.2;
  else
    heating = true;
}

void heat() {
  if (current >= 18
      && current < 22)
    current += 0.2;
  else
    heating = false;
}
```

```
int main() {
  current = 18;
  heating = true;
  while (1) {
    if (heating)
      heat();

    else if (!heating)
      cool();

    else
      break; // error state
  }
  return 0;
}
```

Figure 2.1: Simple Thermostat: C Source Code

In the following the LLVM Intermediate Representation, which has been generated from the C code using Clang, is explained, starting in Figure 2.2 with the identifier, the target architecture and the global variables **`current`** *and* **`heating`** *from the resulting LLVM module are shown.*

```
; ModuleID = 'thermostat.c'
target triple = "i686-pc-win32"
@current = common global double 0.000000e+000, align 8
@heating = common global i32 0, align 4
```

Figure 2.2: Simple Thermostat: LLVM IR - ID, globals

The function `cool` *is illustrated on the left hand side of Figure 2.3. A function inside of LLVM is build out of basic blocks (BB). The first BB represents the first condition of the if statement. Therefore the value of the global variable* `current` *is loaded into the internal variable* `%1` *and the result of the compare operation "signed greater than" into* `%2`*. Each BB has to be completed by a "Terminator Instruction", in this case it is a conditional branch instruction. Based on the truth value of* `%2` *the next BB can either be* `label %3` *(the second condition of the if statement) or* `label %12`*. BB* `label %6` *represents the instruction* `current -= 0.2`*. First the value of current is loaded in* `%7`*, the result of the subtraction is stored in* `%8` *and finally written back to the address of current. The else branch of the function cool is inside BB* `label %9`*, in here the value of heating is set to true (*`1`*). The final BB is* `label %10`*, which returns the value void of the function.*
On the right hand side of Figure 2.3 the function `heat` *is shown. It has the almost same functionality as function* `cool`*, except that instead of decreasing the temperature , it gets increased.*

```
define void @cool() nounwind {
  %1 = load double* @current
  %2 = fcmp ogt double %1, 1.90e+001
  br i1 %2, label %3, label %9

; <label>:3
  %4 = load double* @current
  %5 = fcmp ole double %4, 2.30e+001
  br i1 %5, label %6, label %9

; <label>:6
  %7 = load double* @current
  %8 = fsub double %7, 2.000000e-001
  store double %8, double* @current
  br label %10

; <label>:9
  store i32 1, i32* @heating
  br label %10

; <label>:10
  ret void
}
```

```
define void @heat() nounwind {
  %1 = load double* @current
  %2 = fcmp oge double %1, 1.80e+001
  br i1 %2, label %3, label %9

; <label>:3
  %4 = load double* @current
  %5 = fcmp olt double %4, 2.20e+001
  br i1 %5, label %6, label %9

; <label>:6
  %7 = load double* @current
  %8 = fadd double %7, 2.0e-001
  store double %8, double* @current
  br label %10

; <label>:9
  store i32 0, i32* @heating
  br label %10

; <label>:10
  ret void
}
```

Figure 2.3: Simple Thermostat: LLVM IR - function cool

In Figure 2.4 the LLVM module of the function **`main`** *of the thermostat example is shown. The initial BB initiates the global variables* **`current`** *with* **`1.8`** *and* **`heating`** *with* **`1`** *and closes with an unconditional branch instruction, which next label is* **`%2`**. *BB* **`label %2`** *represents the if statement in function main, which either leads to the BB* **`label %5`** *- the if branch calling function heat() - or BB* **`label %6`**, *the else if statement. From BB* **`label %6`** *depending on the else if condition either BB* **`label %9`** *- the else if branch calling function cool() - or BB* **`label %13`** *- the else branch, returning* **`0`** *- is reached. The next BB to BBs* **`label %5`** *and* **`label %9`** *is BB* **`label %2`**. *This loop represents the while statement.*

```
define i32 @main(i32 %argc, i8** %argv) nounwind {
  %1 = alloca i32
  %argc.addr = alloca i32           ; <label>:6
  %argv.addr = alloca i8**            %7 = load i32* @heating
  store i32 0, i32* %1                %8 = icmp ne i32 %7, 0
  store i32 %argc, i32* %argc.addr    br i1 %8, label %13, label %9
  store i8** %argv, i8*** %argv.addr
  store double 1.800000e+001, double* @current
  store i32 1, i32* @heating
  br label %2                       ; <label>:9
                                      call void @cool()
; <label>:2                           br label %2
  %3 = load i32* @heating
  %4 = icmp ne i32 %3, 0            ; <label>:13
  br i1 %4, label %5, label %6        store i32 0, i32* %1
                                      %14 = load i32* %1
; <label>:5                           ret i32 %14
  call void @heat()                 }
  br label %2
```

Figure 2.4: Simple Thermostat: LLVM IR - function main

2.1.4 Techniques to reduce the problem size

As the model checking problem - refer to [Chapter 2.1.5] - is limited by the halting problem, it is necessary to reduce the size of the problem as much as possible. There exist several approaches to reduce the problem size, like for example Abstract Interpretation - introduced in [Definition 1] -, Program Slicing and so on.

In this thesis the technique of Program Slicing is used to reduce the code size of the Intermediate Representation.

2.1.4.1 Program Slicing

Program Slicing [30] is a technique used to delimit programs only to the relevant set of instructions according to some given slicing criterion. The goal is to eliminate all code artifacts which are not affecting the values of the point of interest, defined by the slicing criteria. In this case Program Slicing is based on Data Flow Analysis.

Definition 4: Data Flow Analysis
Data Flow Analysis [15] is the process of gathering information about the program semantics in a program. This process is based on a Control Flow Graph, the analysis of the abstract semantics [Definition 1] and a Callgraph of the program.

2.1.5 Model Checking

In order to prove the absence or to find a witness of runtime errors in C source code the Intermediate Representation of LLVM - introduced in [Chapter 2.1.3.1] - one solution is the transformation into a model checking problem.

Definition 5: Model Checking
Model Checking [4] is the process to automatically verify if some model, which is described by a finite state system, meets a given specification. Therefore the process traverses over the state space of the specification's underlying system. This process is limited by the state explosion problem [28], as in the worst case the entire state space is examined.

One of the simplest and earliest approaches to handle model checking is the Satisfiability Problem, which is introduced in the following section.

2.1.5.1 Satisfiability Problem

The Satisfiability Problem [7] is a decision problem of propositional logic. Given a formula Φ in propositional logic the goal is to determine, whether Φ is satisfiable or not. The formula Φ itself is usually given in conjunctive normal form.
The Satisfiability Problem belongs to the class of $\mathcal{NP}$ problems. This means the problem is verifiable in non-deterministic polynomial time.

Definition 6: Clause
A clause is the disjunction of literals, where a literal is either the positive or negative occurrence of a variable.

Definition 7: Conjunctive Normal Form (CNF)
Conjunctive Normal Form (CNF) is the conjunction of clauses.

Example:
Let Φ = $(a \vee b) \wedge (\neg a \vee \neg b)$ *be the input to the decision problem. For small input problems a quite simple way to test* Φ *for Satisfiability is to introduce a truthtable for* Φ*, like shown in Table 2.2.*

a	b	(a∨b)	(¬a∨¬b)	(a∨b)∧(¬a∨¬b)
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

Table 2.2: Truthtable of Φ

Obviously a truthtable is not a suitable method for proving when the complexity of the input problem increases. But there exist several tools, for example the SAT Solver, which is based on the theory of SAT Solving introduced in [Chapter 2.2.1].

It is possible to transform LLVM into the Satisfiability Problem. But if this transformation has to cover the entire behavior of LLVM the resulting Satisfiability Problem huge. Therefore some parts of the behavior are only mapped to abstract variables in the Satisfiability Problem. But the resulting abstraction might be imprecise in respect to the original behavior, because if there exists a satisfying solution for the abstraction, the underlying arithmetic of those variables might not be feasible.
But there subsists another theory which is able to handle boolean and arithmetic parts: the **Satisfiability Modulo Theories**. This theory is described in detail in the next section.

2.1.5.2 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) Problem [11] is also a decision problem. The input formula Ψ is not bound on boolean logic only, but is expendable with a various number of theories, such as the theory of real numbers, the theory of integers, etc. Ψ is still treated as a conjunction of clauses, where a clause consists of one literal or is a disjunction of at least two literals. But a literal does not have to be boolean any more, it can also be a mathematical statement, equation or inequation. Most of the Solvers for the SMT Problem depend on the SMTLIB standard input format. But this standard only allows a static input formula and as C Code also includes functions containing loops, jumps, etc., this format is not the suitable one. These types of problems are usually handled by Bounded Model Checking.

Definition 8: Bounded Model Checking (BMC)
Bounded Model Checking [5] is a special case of Model Checking [Definition 5] and

defined as the process to verify that a model meets a given specification for a fixed number of time steps. Therefore the model is unrolled for at most k time steps into a formula and checked in each step by a Solver until either time step k is reached or the Solver proves a violation to the specification in one of those iterations. The process of unrolling can be described as the sequence of instruction steps until timestep k.

Example:
Let `M` be a simple thermostat, which is shown in Figure 2.5. The thermostat has the same behavior as the thermostat from Figure 2.1.

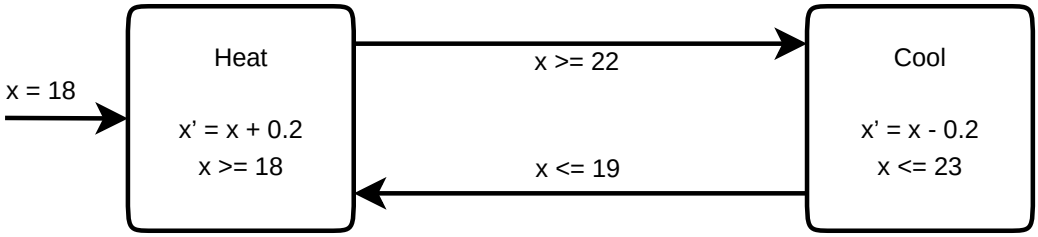


Figure 2.5: Simple Thermostat: Automaton

In this example the goal is to prove that the thermostat never violates the safety properties of state $\texttt{heat}\ (x \geq 18)$ *or state* $\texttt{cool}\ (x \leq 23)$*. This task seems to be quite easy at the first look at Figure 2.5, because the automaton looks like it is well defined, but it is not possible to prove such criteria for a deterministic automaton. The only proof that can be given is for a defined number* $k \in \mathbb{N}$ *of timesteps. Therefore the automaton can be unrolled into a formula* Φ_M*, for example for timestep* `k=1`*:*

$$\Phi_M = \underbrace{(x_0 = 18 \wedge \texttt{heat} \wedge \texttt{not(cool)} \wedge \ldots)}_{k=0} \wedge$$
$$\underbrace{((\texttt{heat} \wedge x_0 \geq 18 \rightarrow \texttt{cool}) \wedge (x_1 = x_0 + 0.2) \wedge \ldots)}_{k=1}$$

There exist a few Solvers, which perform BMC on the input problem in order to find a satisfiable solution, operating on the hys language. This language is introduced in the following section.

2.1.5.3 hys Language

The hys language was first introduced in [13] and in general describes the input problem as some kind of state machine. This language is the form of Satisfiability Problem the LLVM module is going to be transformed into. The detailed transformation process is described in [Chapter 3].

The hys language contains four different sections: DECL, INIT, TRANS and TARGET. In the first section DECL all the variables appearing in the input problem are defined. The state machine or automaton corresponding to the input problem is defined in the three sections INIT, TRANS and TARGET. The INIT section describes the initial state of the input problem and usually assigns the variables with their initial values. In the section TRANS all possible transitions representing the input problem are defined. TARGET finally introduces the target state representing the code segment or condition which is tried to be reached by taking a sequence of transitions from TRANS.

Accepted types are boolean, float or integer, where it is recommended for the numerical types to append the range in which the variable is valid in order to keep the search space as small as possible. According to the types there exist several operators, which are shown in Table 2.3.

boolean operators:	**arithmetic operators:**
and	$+, -, *, \hat{}$
or	abs
nand	min
nor	max
xor	exp
nxor ($\leftrightarrow$)	sin
impl ($\rightarrow$)	cos
not (!)	nrt

Table 2.3: hys Language - Operators

In order to provide a possibility to handle temporal statements, the operator "'" has been introduced. The operator is applicable to every type of variable (boolean, float and integer). The statement: `x' = x + 1` at time interval zero is interpreted as: $\texttt{x}_1 = \texttt{x}_0 + 1$. In Table 2.3 there exists no arithmetic operator for division, but with the temporal operator "'" it is easy to create an equivalent statement:

$$\texttt{x} = \frac{\texttt{y}}{\texttt{z}} \Leftrightarrow \texttt{z} * \texttt{x'} = \texttt{y}$$

Example:
Let `M` again be the thermostat defined in Figure 2.5. In order to have a proper TARGET for the hys language, let the given safety property be:

"Is it possible for `x` to reach a value, which is lower than `18`°C or greater than `23`°C?"

The according definition of the thermostat in the hys language is given in Figure 2.6. In DECL the temperature variable `x` *is defined over the interval* `[17.0, 24.0]`*, as well as the two states of the thermostat:* `cool` *and* `heat`*. In the section INIT* `x` *is initiated to* `18°` *and the initial state is set to* `heat`*. The section TRANS represents the actual transition system of the automaton. The first four logic statements describe the according transition to be taken due to the condition on the temperature* `x`*, where for example* `cool'` *states that in the next time step the automaton is inside state* `cool`*. For example the first one: If the current state is* `heat` *and* $x \geq 22.0°$*, then the next state is* `cool`*. The next two transitions represent the actions to be executed if the thermostat is in mode cooling or heating. The last transition item guarantees that the thermostat is either in state* `cool` *or in state* `heat`*, but not in both at the same time. In the section TARGET the stated safety property - "if* `x` *can reach a temperature outside the bounds* $[18, 23]$*" - is described.*

```
DECL

real [17.0 ,24.0] x;
boole cool, heat;


INIT

x = 18.0;
!cool and heat;


TRANS

heat and x >= 22.0 → cool';
heat and x >= 18.0 → heat';
cool and x <= 19.0 → heat';
cool and x <= 23.0 → cool';

heat and heat' →
  x' = x + 0.2;
cool and cool' →
  x' = x - 0.2;

cool' + heat' = 1;


TARGET

(x < 18) or (x > 23)
```

Figure 2.6: Simple Thermostat: hys Language

2.2 Probabilistic SAT Solving

Probabilistic SAT Solving [24, 2] covers a wide range of approaches to handle SAT Solving. In this thesis Probabilistic SAT Solving refers to the approach Moser introduced in his paper: "A constructive proof of the Lovász Local Lemma" [19], which is based on the Lovász Local Lemma [23]. The implementation, test and research on the algorithms introduced by Moser is the second task of this thesis and described in detail in [Chapter 4]. The fundamentals to this approach are given in section 2.2.2, but first of all SAT Solving is introduced in the following section.

2.2.1 SAT Solving

SAT Solving [21] is the process to solve the Satisfiability Problem [Chapter 2.1.5.1] and is in general the search for a satisfying assignment to a given input formula. There exist several approaches to find such an assignment, but most state of the art SAT Solvers are based on the DPLL algorithm, which is described in [Definition 12]. Another approach is for example the construction of a Binary Decision Diagram (BDD), but the construction of such a BDD is quite expensive, because the size of a BDD increases - depending on the variable ordering - either linear or in the worst case even exponential. Due to this reason, BDDs are not used in most of the cases.

The successive definitions are necessary to understand the DPLL algorithm properly.

Definition 9: Satisfiability of clauses
A clause is satisfiable, if at least one of its containing literals is fulfilled.

Example:
Let $l = (a \vee b \vee \neg c)$ *be a clause, then* l *is satisfied, if for example* $a = 1$ *holds.*

Definition 10: Unitclause
A clause is a unitclause (unit), if only one literal has no assignment, all other literals are already assigned and the clause is not fulfilled under the current assignment. Using the definition of satisfiability of clauses it follows that this last literal has to be assigned with the correct truth value, because otherwise the clause could not be satisfied any more.

Example:
Let $l = (a \vee b \vee \neg c)$ *be a clause, where* $b=0$ *and* $c=1$ *are already assigned, hence* a *must be set to* 1*.*

Definition 11: (Unit)Propagation
From the definition of a unitclause the definition of propagation follows straight

forward. Unit propagation is the process to spread the value of a literal assignment from a unit clause through the entire set of clauses. The consequence of unit propagation could be new unit clauses, which recursively invoke unit propagation themselves.

Example:
Let $l = (a \vee b \vee \neg c)$, $m = (a \vee d)$, $n = (\neg a \vee d)$, ... *be a set of clauses with the current assignment* $b = 0$, $c = 1$. *Thus* m *is unit and* $a = 1$ *has to be the next assignment, which leads to the result that* m *is satisfied and* n *becomes unit. So on the next step* d *has to be assigned to* 1 *and so on.*

Definition 12: DPLL Algorithm
The Davis Putnam Logemann Loveland (DPLL) [9, 10] algorithm is shown in Figure 2.7. The search process of most state of the art SAT Solvers is based on the DPLL algorithm.

```
function DPLL(Φ)
  if all clauses in Φ are satisfied
    return Φ = SAT
  if one clause is unsatisfiable
    return Φ = UNSAT
  for each unitclause c ∈ Φ
    Φ = propagate(c, Φ)
  for each pure literal l ∈ Φ
    Φ = assign(l, Φ)
  l = nextLiteral(Φ)
  return DPLL(Φ∧ l) ∨ DPLL(Φ∧¬ l)
```

Figure 2.7: DPLL Algorithm

The algorithm has two return values: SAT or UNSAT. First of all it is checked, if all clauses are satisfied, because in this case the formula Φ is satisfied. Whereas if only one clause is unsatisfied, the entire formula is as well. In the next step unit propagation is performed and the assignment of literals occurring purely in Φ. Then a new literal is chosen and DPLL is called recursively once with $\Phi \wedge l$ and with $\Phi \wedge \neg l$.

Usually the DPLL algorithm is extended with conflict analysis which includes a backtracking mechanism in combination with a learning mechanism. Backtracking enables undoing a decision, if a conflict during the search process has occurred, and the learning mechanism will ensure that the same conflict will not happen any more during the search process. The search algorithm is displayed in Figure 2.8.

```
loop
  propagate();
  if !conflict
    if all variables assigned
      return SAT;
    else
      decide(); //pick new variable
  else
    analyze();
    if conflict@top-level
      return UNSAT;
    else
      backtrack();
```

Figure 2.8: SAT Search Algorithm

A quite new approach to solve SAT formulas has been derived from a lemma stated in 1975 by L. Lovász in [23], the: Lovász Local Lemma. Basically the lemma is defined on events, but when extending an event to a clause it is also applicable on the satisfiability problem. The definition of the lemma is given in the next section.

2.2.2 Lovász Local Lemma (LLL)

The Lovász Local Lemma (LLL) [23] is a tool which is used in probability theory. In order to describe the lemma in words, the succeeding definitions are required:

Let f be a k-cnf formula and let $\mathcal{A} = \{A_1, \ldots, A_m\}$ be a set of events, where each event $A_i \in \mathcal{A}$ is determined by a finite set of mutually independent random variables $p_i \in \mathcal{P}, 1 \leq i \leq n$ in a probability space, where each variable p_i has its domain D_i. Let $\mathcal{G}$ be the dependency graph of f : $\mathcal{G}_f = (V, E)$ with vertices $V = \mathcal{A}$ and edges $E = \{(A_i, A_j) \in f |\ (A_i \neq A_j) \wedge (vbl(A_i) \cap vbl(A_j) \neq \emptyset\}$, where $vbl(A_i)$ denotes the set of variables occurring in A_i.

Given a set of events, the lemma declares the probability that none of them occurs is not zero if the probability of an event itself is smaller than the number of events depending on it. The proof of the lemma itself is non-constructive and is therefore not illustrated in this thesis.

Moser eased the LLL in [19] to the circumstances of the satisfiability problem. Under the assumptions that: $k \in \mathbb{N}$ and f is an k-cnf formula and given a clause $c \in f$, the set of neighbors of c is defined as $\Gamma_f(c) = \{c_2 \in f | (c \neq c_2) \wedge (vbl(c) \cap vbl(c_2) \neq \emptyset)\}$,

where $vbl(c)$ is the set of all variables occurring in c, the simplified version of the LLL is shown in the following theorem:

Theorem 1 [23]
If f is a k-cnf formula and all clauses $c \in f$ satisfy the characteristic $|\Gamma_f(c)| \leq 2^{k-2}$, then f is satisfiable.

Extending the neighborhood of a clause $c \in f$ to $\Gamma_f^+(c) = \Gamma_f(c) \cup \{c\}$, Moser introduced a second theorem:

Theorem 2 [19]
If f is a k-cnf formula and all clauses $c \in f$ satisfy the property $|\Gamma_f^+(c)| \leq 2^{k-5}$, then f is satisfiable and there exists a randomized algorithm that finds a satisfying assignment to f in expected time polynomial in $|f|$ (independent of k).

Moser proved the lemma in [19] to be correct, which is at least for those problems meeting the condition an aperture in SAT-solving, as the general satisfiability problem belongs to the class of $\mathcal{NP}$ problems. In the paper [20] Moser and Tardos even refined the lemma and gave an expected run time in $O(\log^2(m))$, where m is the number clauses occurring in f.

In the following section the algorithms to the LLL are described.

2.2.2.1 Algorithms

In general Moser introduced three algorithms based on the Lovász Local Lemma, the first one in [19], which is described in pseudo code in Figure 2.9.

```
function solve_lll(f)
   assignment = random assignment over variables v_i;
   run = 0;
   while ∃ clause ∈ f :  unsatisfied do
     if (run++ > log(m) + 2 )
       assignment = random assignment over variables v_i;
       run = 0;
       restart;
     assignment = logically_correct(f, assignment, clause);
   return assignment;
```

Figure 2.9: Moser: First LLL Algorithm - solve_lll

Initially the function `solve_lll` creates a random assignment for all variables in

the formula `f` and starts the search process. This process runs until all clauses are satisfied under the current assignment. So in the worst case, if the function `f` does not meat the LLL constraint and `f` is unsatisfiable, the process will never terminate. While there exists a clause which is not fulfilled under the current assignment, the process updates the assignment by invoking `logically_correct` and passing the formula, the current assignment and the clause to it. Once the number of steps of a run has passed the threshold `log(m) + 2`, the process is restarted with a new initial assignment.

```
function logically_correct(F, A, C)
   A = A : ∀ var ∈ C: A[var] = rand(0,1);
   while ∃ clause ∈ Γ⁺(C): unsatisfied do
     A = logically_correct(F, A, C);
   return A;
```

Figure 2.10: Moser: First LLL Algorithm - logically_correct

The function `logically_correct` illustrated in Figure 2.10 updates the assignment by random bits at all the positions of those variables which the clause C contains. In the following, for each clause which is a neighbor to C and which is not fulfilled under the updated assignment, the assignment will be updated again by invoking `logically_correct` recursively with the corresponding clause.

In the paper [20] Moser released along with G. Tardos two algorithms, which are derived from the first version.

```
function sequential_lll
   ∀ var ∈ 𝒫 do
     assignment[var] = rand(0,1);
   while ∃ A ∈ 𝒜:  A is violated do
     assignment = assignment :  ∀ var ∈ A:  var = rand(0,1);
   return assignment;
```

Figure 2.11: Moser and Tardos: Sequential LLL Algorithm

The function `sequential_lll` shown in Figure 2.11 is almost the simplified version of `solve_lll` and its functional description follows straightforward from the pseudo code. Starting from an initially random assignment, the process updates the bits assignment until there is no event violated any more.

```
function parallel_lll
   ∀ var ∈ 𝒫 do in parallel
     assignment[var] = rand(0,1);
   while ∃ A ∈ 𝒜:  A is violated do
     S = maximal independent set in subgraph 𝒢_𝒜
       induced by all A : violated(A) (constructed in parallel);
     ∀ var ∈ ⋃_{A∈S} do in parallel
       assignment[var] = rand(0,1)
```

Figure 2.12: Moser and Tardos: Parallel LLL Algorithm

As parallel computing has become a large impact over the last few years, it is not quaint that a parallel version in Figure 2.12 of the algorithm `parallel_lll` has been introduced. In this version initially all bits of the assignment are assigned to random values in parallel. In the recursive part of the function the process computes in parallel the set of all clauses which are unfulfilled under the current assignment. In order to keep the necessity of independent events the maximal independent set of this set is calculated. During the following update all bits in the assignment, which correspond to the variables occurring in the independent set of clauses, are set to a random value in parallel.

Moser and Tardos also invented a deterministic variant of those algorithms. The main idea of this variant is based on having witness trees with size in the range $[\log(m), 2\log(m)]$, where each node represents an entry from the entire truthtable for the input problem. Due to the fact that for the computation of these input tables the sequential algorithm is used and since the only purpose of this master thesis is to abuse the algorithm for hopefully finding a solution in at most polynomial time in respect to the size of the input problem, the deterministic variant will not be discussed here any further.

Chapter 3

Program Analysis

In this thesis Program Analysis refers to the process of proving the absence or finding witnesses of runtime errors in C source code as described in [Chapter 2.1]. The actual task is the transformation of LLVM, which represents the source code [Chapter 2.1.3], into the hys language introduced in [Chapter 2.1.5.3].

3.1 Transformation from LLVM to hys Language

The hys language has been chosen as the target for the transformation process, because it features the description of automata or state machines and the representation of the source code in LLVM characterizes some kind of state machine. There exist other approaches, but in most of them it is necessary to statically perform loop unrolling and "flattening" of the model, which is described by the Intermediate Representation. Another advantage of the hys language is that the Solvers operating on that language can use the feature of learning. The learning refers to the reuse of some conflicts of previous iterations, which are still valid in the current iteration and therefore do not need to be computed again. This feature can lead to a speedup of the solving process.

The requirements for the hys language needed to describe the virtual machine of LLVM are described in the next section.

3.1.1 The hys Target Machine

For the purpose of proving the absence or finding witnesses of runtime errors in C source code, the hys language itself is a good choice, because of the featured automaton (or state machine) description. A state machine describing the program semantics from LLVM is the construction stone. But as the virtual machine of LLVM is based on a Risk-like load and store architecture [Chapter 2.1.3], the machine described in the hys language must have the same features. So the machine must

provide a memory management in order to load data from the memory to a register and store data vice versa. The values inside the virtual registers and the memory of a LLVM module are in bitvector representation in order to be able to provide signed and unsigned data types, but as the hys language only features the data types boolean, float and integer - which must be defined over a fixed interval - a solution to provide the same behavior must be found. Also exception handling is a necessity the hys target machine must feature, like for example overflow exceptions for arithmetic operations, the divide-by-zero exception or memory exceptions.
The witness the hys target machine is trying to find is a sequence of steps where the goal is to reach and fulfill the TARGET section. The target defined in the TARGET section is the defect candidate.
In Figure 3.1 the hys target machine meeting the above named constraints is shown.

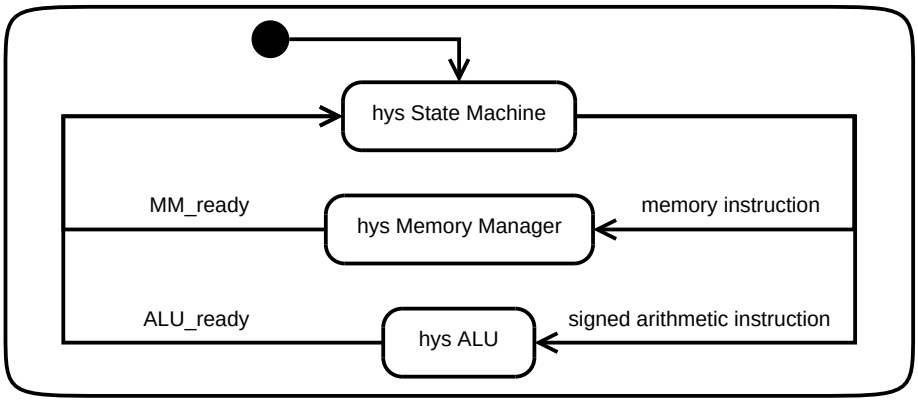


Figure 3.1: hys Target Machine: UML State Diagram

The hys target machine contains three states: the hys State Machine, the hys Memory Manager and the hys ALU. The registers and the memory of the hys target machine are realized as variables of the according type (bool, float or int). In order to be able to assign as well a signed as an unsigned value to a register or memory address, the actual value of a variable in the hys target machine is stored as the decimal number of the corresponding bitvector representation. So a 32 bit wide integer is defined on the interval $[0, 2^{32} - 1]$, which can either represent a signed integer with a range of value from $[-2^{31}, 2^{31} - 1]$ or an unsigned integer with the total range of value.
Most of the instructions are handled by the hys State Machine [Chapter 3.1.1.1], in case of arithmetic logic instructions the value of the operands, which is stored inside a register, must be of type unsigned. If the value is of type signed, the instruction is handled by the hys ALU [Chapter 3.1.1.3]. Each memory instruction like a load or store instruction is passed to the hys Memory Manager [Chapter 3.1.1.2].

3.1.1.1 hys State Machine

The state machine in the hys language is responsible for handling the control flow of the program described by the LLVM IR. In order to achieve a unique ID for the states in this machine and to have a proper identification of the position in the IR, a state in the hys machine is a boolean variable identified by the name, which is the concatenation of the function name and the actual basic block. For the identification of the instructions each basic block is fit with an instruction program counter.
In order to handle jumps to an instruction inside a basic block, like for example when returning from a call, a jump table is required. Each instruction has a unique entry in this table. The return address, which is represented by the index of the table, has to be properly defined. Therefore the hys target machine features the register `JTI`.

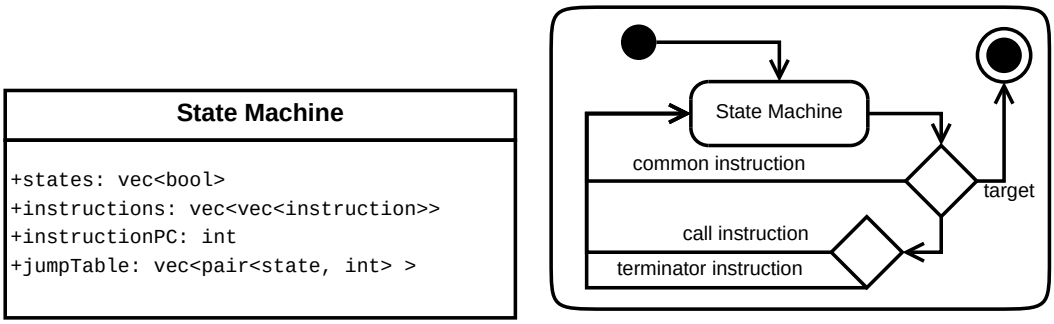


Figure 3.2: hys State Machine: UML Class and State Diagram

In Figure 3.2 the UML class diagram and the UML state diagram of the hys state machine are illustrated. As described in advance, the state machine contains a vector of booleans representing the states. The instructions of the hys target machine are ordered according to the containing function and basic block of the LLVM IR in a vector of vectors. In addition the state machine features the instructionPC and the jump table.
For the state machine diagram shown on the right hand side of Figure 3.2 in general there exist three possible transitions to be fired. The first transition `common instruction` represents the general instruction and has the following hys logic:

$$\texttt{states}_i \wedge \texttt{instructionPC = k} \rightarrow$$
$$\texttt{states}_i\texttt{'} \wedge \texttt{instructions}_k \wedge$$
$$\texttt{instructionPC' = instructionPC + 1;}$$

The machine is in $\texttt{states}_i$ and the `instructionPC` equals `k`, which leads to the execution of $\texttt{instructions}_k$ and the machine resides in $\texttt{states}_i$ but with an incremented `instructionPC`. Once the target condition is reached and fulfilled the

transition `target` is fired, which terminates the hys target machine and states that a witness has been found.
The third possible transition is split into two transitions: the `call instruction` and the `terminator instruction`. A call instruction has the following hys logic:

$$\texttt{states}_i \wedge \texttt{instructionPC = k} \rightarrow$$
$$\texttt{states}_{calledFunction}\texttt{'} \wedge \texttt{instructionPC' = 0} \wedge$$
$$\texttt{JTI' = index(states}_i\texttt{, instructionPC + 1);}$$

In this case the machine is in $\texttt{states}_i$ and the current instruction is a `call instruction`. Then the machine changes into the initial state of the function to be called: $\texttt{states}_{\texttt{calledFunction}}$, resets the `instructionPC` and assigns the value of the next instruction from $\texttt{states}_i$ from the jump table to the register `JTI`.
For the `terminator instruction` two different cases must be compared. First if it is a `common terminator instruction` versus second if it is a `return instruction`. The hys logic for the `common terminator instruction` is:

$$\texttt{states}_i \wedge \texttt{instructionPC = k} \rightarrow$$
$$\texttt{states}_{next}\texttt{'} \wedge \texttt{instructionPC' = 0;}$$

The machine is in $\texttt{states}_i$ and the `instructionPC` equals the last instruction of the current basic block, which leads to the activation of the next state ($\texttt{states}_{\texttt{next}}$) and the reset of the `instructionPC`. A `return instruction` is mapped onto the following hys logic:

$$\texttt{states}_i \wedge \texttt{instructionPC = k} \rightarrow$$
$$\texttt{jump';}$$

In this case the machine is in $\texttt{states}_i$ and the current instruction is a `return instruction`. Then the machine changes into the state `jump`. In this state the machine determines where to jump according to the return address previously set by a `call instruction` to the register `JTI`. An entry in the jump table looks like:

$$\vdots$$
$$\texttt{jump} \wedge \texttt{JTI = 12} \rightarrow \texttt{states}_i\texttt{'} \wedge \texttt{instructionPC' = 0;}$$
$$\texttt{jump} \wedge \texttt{JTI = 13} \rightarrow \texttt{states}_i\texttt{'} \wedge \texttt{instructionPC' = 1;}$$
$$\vdots$$
$$\texttt{jump} \wedge \texttt{JTI = 29} \rightarrow \texttt{states}_j\texttt{'} \wedge \texttt{instructionPC' = 0;}$$
$$\vdots$$

In order to provide a correct behavior of the state transition system, where the machine can reside only in one state at a time - except for the concurrency of the state machine with the memory manager or the arithmetic logic unit -, the following equation has to be added to the machine:

$$\texttt{states}_0\texttt{'} + \texttt{states}_1\texttt{'} + \ldots + \texttt{states}_{\texttt{last}}\texttt{'} = 1$$

The following example illustrates the introduced semantics of the hys state machine.

Example:
The LLVM instruction `%add = add i32 %x %y`*, where* `%add`*,* `%x` *and* `%y` *are of type unsigned int, is an example for the* `common instruction` *transition. The instruction resides inside the function* `sum` *and the basic block* `entry` *and is the third instruction in this block. The according hys logic is:*

```
sumentry ∧ instructionPC = 3 →
    sumentry' ∧ add' = x + y ∧
    instructionPC' = instructionPC + 1;
```

As an example for the LLVM `terminator instruction` *the following branch instruction is taken:* `br i1 %cmp, label %if.then, label %if.else`*. The instruction is located in the function* `check` *and the basic block* `BB0` *and is the fifth instruction of this block. The virtual register* `%cmp` *contains a boolean value which is identified by the* `i1`*. According to that value the branch instruction either jumps to the label* `if.then` *or* `if.else`*. The resulting hys logic of this instruction is the following:*

```
checkBB0 ∧ instructionPC = 5 ∧ cmp →
    if_then' ∧ instructionPC' = 0;
checkBB0 ∧ instructionPC = 5 ∧ !cmp →
    if_else' ∧ instructionPC' = 0;
```

The labels `if.then` *and* `if.else` *are replaced by* `if_then` *and* `if_else` *as the hys language does not support the dot character in a variable name.*

For the LLVM `call instruction` *the instruction* `call void @compute(i32 %tmp, i32 %tmp1, i32* %res)` *is taken as an example. This example is located in function* `main`*, basic block* `entry` *and is the 10th instruction inside this block. The function compute is defined as:* `compute(i32 %x, i32 %y, i32* result)` *and its initial basic block is also named* `entry`*. The corresponding hys logic for this instruction is:*

```
mainentry ∧ instructionPC = 10 →
    computeentry' ∧ instructionPC' = 0 ∧
    JTI' = index(mainentry, 11) ∧ var_x' = var_tmp ∧
    var_y' = var_tmp1 ∧ var_result' = var_res;
```

The `terminator instruction ret void` *of the function compute is a* `return instruction` *which is located in basic block* `if.end` *and is the first and only instruction in this block. It is mapped to the following hys logic:*

```
computeif_end ∧ instructionPC = 0 →
    jump';
```

In the next section the hys memory manager is introduced.

3.1.1.2 hys Memory Manager

The memory manager of the hys target machine handles the transport of data values between the registers inside the hys target machine and the memory addresses.

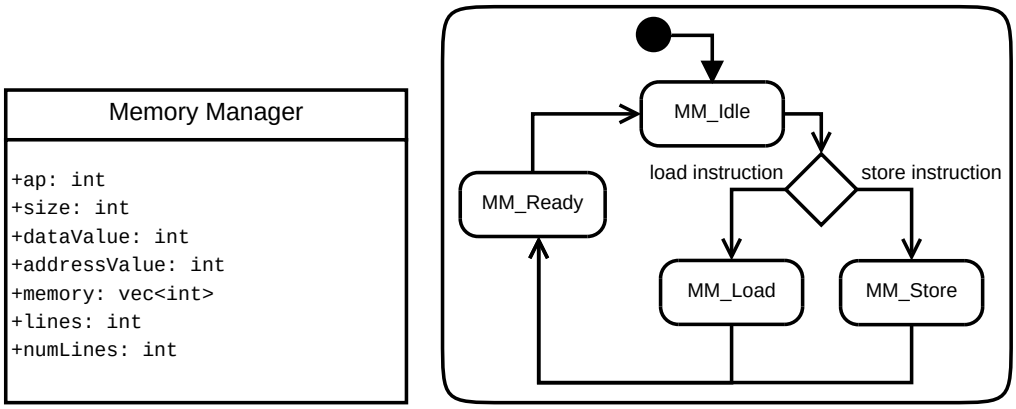


Figure 3.3: hys Memory Manager: UML Class and State Diagram

On the left hand side of Figure 3.3 the UML class diagram of the hys memory manager is shown. The address pointer (`ap`) represents the register which points to the current memory address. The item `size` defines the data size to be loaded or stored from or to the memory. The register `dataValue` is the input-output register of the memory manager. As the address size is fixed, the internal register `addressValue` is used to transfer values between the register `dataValue`, which can contain values of various type, and the actual memory cells, which are implemented as a vector of integers.
On the right hand side of the Figure the UML state diagram of the memory manager is illustrated. Initially the manager enters the state `MM_Idle`. Depending on the type of memory instruction, the manager changes either to state `MM_Load` taking transition `load instruction` or state `MM_Store` taking transition `store instruction`. Once the memory operation is finished, the manager enters state `MM_Ready` in order to notify the waiting instruction.
The transition `load instruction` which enables the load operation of the hys memory manager has the following hys logic:

```
states_i ∧ instructionPC = k ∧ MM_Idle →
   MM_Load' ∧ ap' = address ∧ size' = datasize;
states_i ∧ instructionPC = k ∧ MM_Ready →
   states_i' ∧ instructionPC' = instructionPC + 1 ∧
   data' = dataValue;
states_i ∧ instructionPC = k ∧ !MM_Ready →
   states_i' ∧ instructionPC' = instructionPC;
```

The machine is in $\texttt{states}_i$, the instruction `k` is a load instruction and the memory manager is idle, then the memory manager is enabled and the `MM_Load` state entered (first implication). Also the base address is set to the address pointer (`ap`) and the data size is assigned to the register `size`. Once the signal `MM_Ready` is triggered the machine increments the `instructionPC` and reads the loaded value from the register `dataValue` (second implication). Until `MM_ready` is triggered the machine resides in the current state at the current instruction (third implication).
Transition `store instruction` enables the store operation of the hys memory manager and has the following hys logic:

```
states_i ∧ instructionPC = k ∧ MM_Idle →
   MM_Store' ∧ ap' = address ∧ dataValue' = value ∧
   size' = datasize;
states_i ∧ instructionPC = k ∧ MM_Ready →
   states_i' ∧ instructionPC' = instructionPC + 1;
states_i ∧ instructionPC = k ∧ !MM_Ready →
   states_i' ∧ instructionPC' = instructionPC;
```

In this case the instruction `k` in the current state $\texttt{states}_i$ is a store instruction and the memory manager is idle, thus the memory manager enables the store process via `MM_Store`. Like for the load instruction the address pointer (`ap`) and the data size (`size`) are assigned, but in this case also the value, which has to be stored, is assigned to the `dataValue` register. The machine also waits in the current state at the current instruction until the signal `MM_ready` is triggered and then continues with the next instruction.

In Figure 3.4 the state `MM_Load` - representing the load operation of the hys memory manager - is shown in form of a UML state diagram. The underlying state transition system of this diagram contains the states `MM_Load_Init` and `MM_Load_computeWord`. The state `MM_Load_init` exists in order to ensure that the register `addressValue` contains the correct value in respect to the current address pointer (`ap`). Therefore the hys target machine has been extended with a table of entries, which uses the state `MM_Load_init` as a trigger. The entry for memory address number 137 looks like:

```
MM_Load_init ∧ ap = 137 → addressValue' = memory_0x137;
```

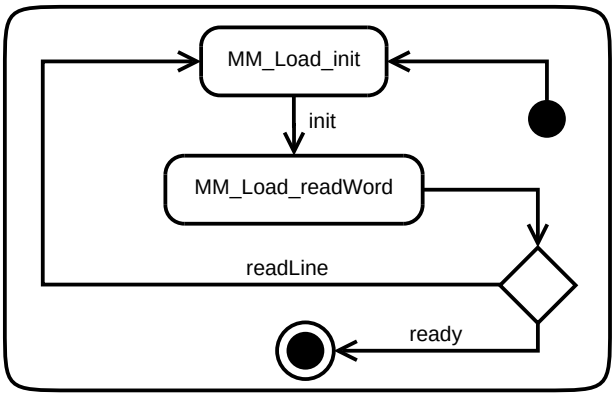


Figure 3.4: hys Memory Manager - state load: UML State Diagram

In order to start or continue with the computation of the value, which consists of several values assigned to the register `addressValue` according to the size to be loaded, the transition `init` is fired. The hys logic of this transition is as follows:

$$\texttt{MM\_Load\_init} \rightarrow \texttt{ap'} = \texttt{ap} \wedge \texttt{lines'} = \texttt{lines} \wedge \texttt{startBit'} = \texttt{startBit};$$

The state `MM_Load_readWord` is used for the computation of the value. The computation is triggered by the firing of transition `readLine`, which has the following hys logic:

$$\texttt{MM\_Load\_readWord} \wedge \texttt{lines} \leq \texttt{numLines} \rightarrow \texttt{MM\_Load\_init'} \wedge \texttt{ap'} = \texttt{ap} + 1 \wedge \texttt{lines'} = \texttt{lines} + 1 \wedge \texttt{dataValue'} = \texttt{dataValue} + \texttt{addressValue} * 2^{\texttt{startBit}} \wedge \texttt{startBit'} = \texttt{startBit} + 8;$$

As long as the condition $\texttt{lines} \leq \texttt{numLines}$ is fulfilled, the `dataValue` is updated and the address pointer `ap`, the line counter `lines` and the factor for the current address line `startBit` are incremented. If the condition is not fulfilled any more the transition `ready` is taken, which indicates the termination of the load process and puts the memory manager into state `MM_Ready` to notify the waiting instruction that the word can be read from `dataValue`. It has the following hys logic:

$$\texttt{MM\_Load\_readWord} \wedge \texttt{lines} > \texttt{numLines} \rightarrow \texttt{MM\_Ready'};$$

Figure 3.5 shows the state `MM_Store`, which represents the store operation of the hys memory manager.
Initially the state `MM_Store_computeWord` is entered. This state has two general transitions to take. If the storage process is finished the transition `ready` is fired. Until the word is totally stored in the memory some of the transitions: `updateData`,

updateLineValue or writeLine are taken. Those three transitions do all have the same precondition: lines $\leq$ numLines.

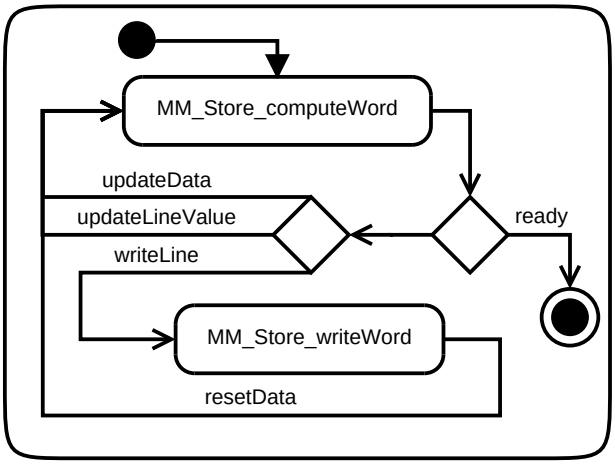


Figure 3.5: hys Memory Manager - state store: UML State Diagram

The transition updateData has the following hys logic:

$$\texttt{MM\_Store\_computeWord} \wedge \texttt{lines} \leq \texttt{numLines} \wedge \texttt{startBit} \geq 0 \rightarrow \texttt{MM\_Store\_computeWord'} \wedge \texttt{power'} = 2^{\texttt{dataBit}} \wedge \texttt{dataBit'} = \texttt{dataBit} - 1 \wedge \texttt{startBit'} = \texttt{startBit} - 1;$$

This transition updates the three internal registers: power, dataBit and startBit, as long as the value in register startBit is greater or equal than zero.
The transition updateLineValue is used to update the value for the next line, which is going to be written to the memory, and has the hys logic:

$$\texttt{MM\_Store\_computeWord} \wedge \texttt{lines} \leq \texttt{numLines} \wedge \texttt{dataValue} > \texttt{power} \rightarrow \texttt{MM\_Store\_computeWord'} \wedge \texttt{addressValue'} = \texttt{addressValue} + \texttt{power} \wedge \texttt{dataValue'} = \texttt{dataValue} - \texttt{power};$$

The two transitions updateData and updataLineValue can be fired in parallel, but as long as the condition startBit $\geq$ 0 is fulfilled, transition updateData must be fired.
Otherwise if the condition is violated the transition writeLine is fired and traverses into state MM_Store_writeWord. The hys logic of this transition is:

$$\texttt{MM\_Store\_computeWord} \wedge \texttt{lines} \leq \texttt{numLines} \wedge \texttt{startBit} < 0 \rightarrow \texttt{MM\_Store\_writeWord'}$$

From state MM_Store_writeWord only transition resetData can be taken, which has the following hys logic:

```
MM_Store_writeWord →
   MM_Store_computeWord' ∧ addressValue' = 0 ∧
   startBit' = 8 ∧ ap' = ap - 1 ∧ lines' = lines + 1;
```

The state `MM_Store_writeWord` (corresponding to the state `MM_Load_init`) is also used as a trigger for the memory manager to write a value from the register `addressValue` to a designated memory address. Therefore a table of entries is added to the hys target machine, where for example the entry for the memory address number 137 looks like:

```
MM_Store_writeWord ∧ ap = 137 → memory_0x137' = addressValue;
```

The signal `MM_Ready`, which notifies the waiting instruction that the memory manager has finished its current process, is only enabled for one cycle. Therefore the following hys logic is added to the memory manager:

```
MM_Ready → MM_Idle';
```

As the memory manager can also reside in one of states at a time, the following equation must be added:

```
MM_Idle' + MM_Load_readWord' + MM_Store_computeWord' +
MM_Store_writeWord' + MM_Ready' = 1;
```

Example:
The LLVM instruction ***`store i32 %add, i32* %z`*** *is an example for the* ***`store instruction`****. This instruction is located in function* ***`sum`****, in basic block* ***`entry`*** *and is the fourth instruction inside this block. The corresponding hys logic for this instruction is:*

```
sumentry ∧ instructionPC = 4 ∧ MM_Idle →
   MM_Store' ∧ ap' = z_address ∧ dataValue' = add ∧ size' = 32;
sumentry ∧ instructionPC = 4 ∧ MM_Ready →
   sumentry' ∧ instructionPC' = 5;
sumentry ∧ instructionPC = 4 ∧ !MM_Ready →
   sumentry' ∧ instructionPC' = 4;
```

The first implication enters the memory manager and enables the store process. Also the base address of the pointer ***`%z`*** *is set to the address pointer (*`ap`*), the value* ***`add`*** *is assigned to the register* ***`dataValue`*** *and the datasize (*`size`*) is set to 32 bit. The second implication is taken once the memory manager has finished the store process, which is notified by the signal* ***`MM_Ready`****. The machine then continues with the next instruction. Until this signal is enabled the third implication has to be taken, which lets the machine wait in state* ***`sumentry`*** *at instruction 4.*

Also in the function ***`sum`*** *and in basic block* ***`entry`*** *resides the LLVM* ***`load instruction: %a = load i32* %b`****. It is the ninth instruction in this basic block and has the following hys logic:*

```
sumentry ∧ instructionPC = 9 ∧ MM_Idle →
   MM_Load' ∧ ap' = b_address ∧ size' = 32;
sumentry ∧ instructionPC = 9 ∧ MM_Ready →
   sumentry' ∧ instructionPC' = 10 ∧ var_a' = dataValue;
sumentry ∧ instructionPC = 9 ∧ !MM_Ready →
   sumentry' ∧ instructionPC' = 9;
```

Again the first transition enters the memory manager, but in this case the load process is enabled. The base address of the pointer **%z** *is set to the address pointer (***ap***) and the datasize (***size***) is set to 32 bit. Once the memory manager has enabled the signal* **MM_Ready** *the instruction number nine in function* **sum** *and basic block* **entry** *wakes up and assigns the value to load from dataValue to* **var_a**.

Exception Handling

The hys memory manager also has to feature exception handling like for example a stack overflow. Therefore the hys memory manager has been extended with the boolean variable `stackOverflow`, which can be treated as a signal. Once the address pointer (`ap`) or the stack pointer (`sp`) are set to a value which is greater than the index of the last memory address, `stackOverflow` is set to True. In the default case the model checking process should find a witness, which does not contain a stack overflow exception. Therefore the variable `stackOverflow` has been added to the target section in negated form. If the user wants the hys target machine to terminate in case of a stack overflow, he only needs to add the `stackOverflow` variable to the target section. Usually the memory manager also has to feature exception handling for addressing errors and so on, but as these are architecture specific errors, they are discussed later on.

In the next section the hys arithmetic logic unit is introduced.

3.1.1.3 hys Arithmetic Logic Unit

The hys language provides a lot of operations [Chapter 2.1.5.3, Table 2.3]. But as the values stored inside the memory address registers (variables) [Chapter 3.1.1.2] are stored in bitvector representation, the operations can only be used straight forward if the value inside a register is of type unsigned. In the other case - where the value is of type signed - the value must first be converted from the bitvector representation into the corresponding signed number which can then be assigned to the operation. After the operation the result must be converted back into the bitvector representation.
In order to handle this, the hys arithmetic logic unit has been created, which is shown in Figure 3.6.

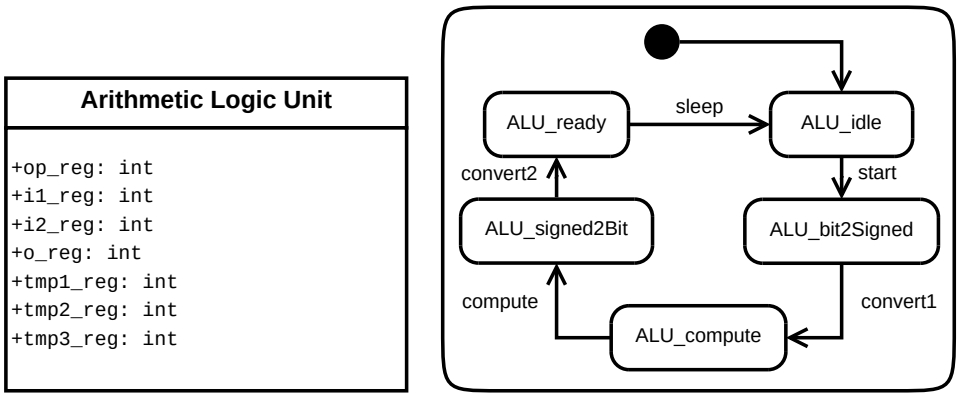


Figure 3.6: hys Arithmetic Logic Unit: UML State Diagram

On the left hand side of Figure 3.6 the UML class diagram of the hys arithmetic logic unit is illustrated. The unit contains three input registers: `op_reg, i1_reg` and `i2_reg` and one output register: `o_reg`. The three temporal registers: `tmp1_reg, tmp2_reg` and `tmp3_reg` are used to store the signed values of the two input registers (`i1_reg` and `i2_reg`) and the signed value of the result of the operation. These are the only registers in the machine which are defined on the interval $[-2^{31}, 2^{31} - 1]$. The UML state diagram for the hys arithmetic unit is shown on the right hand side. Initially the unit is set to idle mode. An arithmetic logic instruction is translated into the hys logic as:

```
states_i ∧ instructionPC = k ∧ ALU_idle →
   ALU_bit2Signed' ∧ op_reg' = m ∧ i1_reg' = op1 ∧
   i2_reg' = op2;
states_i ∧ instructionPC = k ∧ ALU_ready →
   states_i' ∧ instructionPC' = instructionPC + 1 ∧
   result' = o_reg;
states_i ∧ instructionPC = k ∧ !ALU_ready →
   states_i' ∧ instructionPC' = instructionPC;
```

The first implication initiates the arithmetic logic unit and assigns the instruction code to the `op_reg`, as well as the two operands of the current instruction to the input registers `i1_reg` and `i2_reg`. The second implication is triggered once the arithmetic logic unit enables the signal `ALU_ready`, reads the result from the output register `o_reg` and increments the instruction program counter. The third implications ensures that the machine resides in the current and at the current instruction until the signal `ALU_ready` is triggered.
From `ALU_bit2Signed` the bitvector values from the two input registers `i1_reg` and `i2_reg` are converted to their according signed value and assigned to the temporal registers `tmp1_reg` and `tmp2_reg` by firing transition `convert1`, which has the following hys logic:

```
ALU_bit2Signed → ALU_compute';
ALU_bit2Signed ∧ i1_reg ≥ 2^31 − 1 → tmp1_reg' = i1_reg - 2^32;
ALU_bit2Signed ∧ i1_reg < 2^31 − 1 → tmp1_reg' = i1_reg;
ALU_bit2Signed ∧ i2_reg ≥ 2^31 − 1 → tmp2_reg' = i2_reg - 2^32;
ALU_bit2Signed ∧ i2_reg < 2^31 − 1 → tmp2_reg' = i2_reg;
```

The state `ALU_compute` is left firing the transition `compute`, which computes the result of the operation according to the type of operation which is identified by the value in `op_reg`. The result is assigned to the temporal register `tmp3_reg`. For the operation `add` for example the operation code is 8, so the hys logic for the add operation on transition `compute` is:

```
ALU_compute ∧ op_reg = 8 →
   ALU_signed2Bit' ∧ tmp3_reg' = tmp1_reg + tmp2_reg;
```

Once the result is computed, which is stored as a signed value in `tmp3_reg`, it must be converted back into the bitvector representation, before it can be assigned to the output register `o_reg`. This is achieved by taking transition `convert2`, which has the following hys logic:

```
ALU_signed2Bit → ALU_ready';
ALU_signed2Bit ∧ tmp3_reg ≥ 0 → o_reg' = tmp3_reg;
ALU_signed2Bit ∧ tmp3_reg < 0 → o_reg' = tmp3_reg + 2^32;
```

After being in the state `ALU_ready`, which indicates the waiting instruction that the result is stable at the output the arithmetic logic unit is set to sleep until the next request is made. Therefore the transition `sleep` is fired. This transition has the hys logic:

```
ALU_ready → ALU_idle';
```

Also the arithmetic logic unit can only reside in one state at a time and therefore the following hys logic must be added:

```
ALU_idle' + ALU_convert_bit2Signed' + ALU_compute' +
ALU_convert_signed2Bit' + ALU_ready' = 1;
```

Example:
In this example the LLVM instruction ***%add = add nsw i32 %x, %y****, where* ***%add****,* ***%x*** *and* ***%y*** *are of type int. The short* ***nsw*** *stands for "no signed wrap", which states that the instruction might have the side affect of an overflow. As the registers* ***%x*** *and* ***%y*** *are of signed type, the hys arithmetic logic unit must be used for the computation of the result. The instruction itself is located in the function* ***sum****, in basic block* ***entry*** *and is the third instruction of this block. It is translated into the following hys logic:*

```
sumentry ∧ instructionPC = 3 ∧ ALU_idle →
   ALU_bit2Signed' ∧ op_reg' = 8 ∧ i1_reg' = x ∧ i2_reg' = y;
sumentry ∧ instructionPC = 3 ∧ ALU_ready →
   sumentry' ∧ instructionPC' = 4 ∧ z' = o_reg;
sumentry ∧ instructionPC = 3 ∧ !ALU_ready →
   sumentry' ∧ instructionPC' = 3;
```

*The first implication initiates the ALU and sets the operation type (*8 *for addition) and assigns the input registers with the correct values. The second implication is only taken when the ALU finished the computation and the result is stable at the output register* `o_reg`*. Then the value can be assigned to* `z` *and the* `instructionPC` *is incremented in order to process the following instruction. Until the ALU has finished the computation the machine must wait in the current state and the current instruction. This is realized in the third implication.*

Exception Handling

The hys arithmetic logic unit also provides exception handling for the exceptions: overflow and div-by-zero. Therefore the two boolean variables `ALU_overflow` and `ALU_div-by-zero` have been introduced, which are enabled if the corresponding exception occurs. The overflow exception is checked via a 64 bit register, which the result of the current instruction is assigned to. With the help of this register the variable `ALU_overflow` is set to true, once the value assigned to the register is greater than $2^{32} - 1$. The default value of the result register contains the wrap around result, if an overflow has occurred. The signal `ALU_div-by-zero` is enabled if the dividend of the divide instruction equals zero and in the default case the hys target machine terminates in case of a div-by-zero exception. If the user wants the hys target machine to terminate on an overflow exception, he has to assign the corresponding variable to the target section. In the other case the exception must not occur in the witness, so the corresponding variable has to be added to the target section in negated form. This triggers the solvers internal backtracking mechanism if an exception occurred on the current trail.

In order to provide this exception handling also for unsigned instructions, the hys arithmetic logic unit has been extended with the variable `ALU_typeUnsigned`. This variable is added to the convert implications `ALU_bit2Signed` and `ALU_signed2Bit`. If the variable is set to true the value inside the input registers is not converted, but just passed to the internal register, which allows the arithmetic logic unit to operate on the total value range of unsigned values.

3.1.2 LLVM IR to hys Target Machine

In order to transform a program which is in LLVM IR onto the hys target machine, every feature from the LLVM IR must be mapped into a corresponding hys feature or if hys does not provide this feature, a workaround must be found.

3.1.2.1 Transformation of the Type System

The first task is the transformation of the type system of LLVM into the hys type system, which differ a lot as shown in Table 3.1.
The three types "bool", "float" and "integer" can be mapped straight forward from the LLVM IR into the hys language. The absence of the type "void" is not a huge problem, as it just declares for a function that nothing is returned.

	LLVM IR	hys language
simple types:	bool float integer void	bool float integer -
derived types:	array function pointer structure	- - - -

Table 3.1: Differences in the Type Systems of the LLVM IR and the hys Language

The first challenge occurs with the "array" type, which is not featured in the hys language. An array inside of the LLVM IR is an ordered sequence of pointers and the hys language does also not support the type "pointer". The dereferencing of the pointer type can be mapped onto the hys memory manager with the help of some additional logic for the interpretation of the intended value type, which has to be defined. But in order to provide a instantiation method for pointers in the hys machine the hys memory manager must also be extended with some kind of allocation operation - refer to [Chapter 3.1.2.2].
Furthermore the "function" type is not supported in the hys language, but it is possible to transform a function into a state transition system, which describes the same behavior as the function itself. The type "structure" is also not featured in the hys language. Whenever a variable of type structure is used in the LLVM IR, it must be completely mapped into the hys language. So every item contained in the structure must be mapped to a unique variable.

3.1.2.2 Transformation of the Memory Management and Addressing

The memory management of the LLVM IR provides the instructions: `alloca`, `load`, `store` and `getelementptr`. The instructions `load` and `store` can be mapped directly onto the operations provided the hys memory manager. But the instruction `alloca`, which allocates memory and returns a pointer, is not supported in the hys memory manager right now. The tasks occurring on the quest to realise the allocation operation are:

1. to define a proper memory alignment,
2. to choose the correct endianess,
3. to create exception handling for memory operations and
4. to realise the pointer type.

In order to provide a proper stack management for the transformed LLVM module inside the hys target machine, which reserves and frees memory for each function and the variables of the module, the hys memory manager must be extended with a equivalent allocation and free operation.
Another challenge for the transformation from the LLVM IR into the hys language, which has not been mentioned till now, is the transformation of intrinsic functions and phi functions.

3.1.2.3 An Approach to the Transformation Process

In the previous sections a lot of tasks have been stated, which would occur when the LLVM IR is mapped onto the hys target machine. In order to solve these tasks the idea was to take a look into the LLVM virtual machine, if some of their features might be reused. During this process also the LLVM Backend got reviewed. This Backend generates machine code for some specific computer architectures. Therefore the LLVM IR is transformed into the corresponding machine code, which includes providing a proper memory and stack generation, mapping the virtual registers to the machine registers or to memory, dereferencing of pointers and the elimination of phi nodes. So in the general with the help of the LLVM Backend most of the challenges described in the previous sections can be avoided.

In Table 3.2 the pros and cons of the transformation process using the LLVM IR directly compared to the LLVM Backend are illustrated.

	LLVM IR	LLVM Backend
pros:	○ target independent	○ compatible type system ○ compatible memory management (including correct stack management) ○ no need for memory exception handling, except for stack overflow
cons:	○ incompatible type system ○ incompatible memory management and solution for stack management required ○ solution for phi functions and intrinsic functions required	○ target dependent ○ solution for mapping between LLVM IR and assembly language required

Table 3.2: LLVM IR versus LLVM Backend

The results of the Table show, that the advantages of the LLVM Backend outperform the direct usage of the LLVM IR for the transformation process onto the hys target machine. The negative aspect of being target dependent can almost be disregarded, because in order to proper verify that a program contains a runtime error or that it does not this verification has to be built based on a specific machine in respect to the target architecture. The second aspect still needs to be realized in future, but this task is quite simple compared the requirements needed to transform the LLVM IR onto the hys target machine.

3.1.3 LLVM Backend (Assembly Writer) to hys Target Machine

The LLVM Backend is used to generate machine code from the LLVM IR for a specific target machine. The Backend provides a list of targets like arm, alpha, mips, ppc, x86, etc.. Out of these target machines the Mips [26] architecture - especially the Mips R2000 architecture - was chosen, because it has quite a small instruction set, provides a memory management with Big-endian and Little-endian mode, as well as 8byte alignment. The Mips Assembly Writer - used by the Backend - transforms the LLVM IR into the Mips Assembly Language [6]. The resulting assembly code already has a proper memory management with only load and store operations and contains no array or pointer types, etc. In this case only the “simple” assembly code must be transformed onto the hys target machine.

Instead of operating on the generated assembly code, the Mips Assembly Writer, which is based on the function pass manager of LLVM, has been rewritten. The underlying function pass manager traverses over a LLVM module function by function, for each function over all basic blocks and for each basic block over all instructions. So the i-th instruction of function `test` in basic block `BB2` in the hys target machine is still encoded as:

```
testBB2 ∧ instructionPC = i →
   testBB2' ∧ instructions_i ∧ instructionPC' = instructionPC + 1;
```

For a detailed description refer to [Chapter 3.1.1.1] (hys State Machine). In the following sections the transformation of specific instructions from the mips assembly language into the hys language is described.

3.1.3.1 Mapping Arithmetic and Logic Instructions

Arithmetic Instructions
The Mips instruction set consists of the basic arithmetic instructions like `add, addi, addiu, addu`, `sub, subu`, `mult, multu` and `div, divu`. The instructions for `add` and `sub` are mapped onto the hys logic instructions `add, sub`. The "i" inside the Mips instruction `add` stands for immediate, which means that the second operand is of type constant, all the other instructions accept only registers as operands. All the instructions without a "u" at the end execute a trap, if an overflow appears during the execution, which also has to be mapped into the hys logic. For all other instructions no overflow check is performed. The instructions for `mult` and `div` compute a 64bit value, whose high part (bits 63-32) is stored in the `hi` register of the Mips machine and the low part (bits 31-0) in the `low` register. This is also mapped one-to-one into hys logic.

Logic Instructions
The logic instructions in the Mips instruction set `and, andi, or, ori, xor, nor, slt, slti` are also mapped one-to-one into hys logic, where `slt` represents set on less than.

3.1.3.2 Mapping Branch and Jump Instructions

The branch instructions `beq, bne` (branch: on equal, not equal) inside the Mips architecture can also be mapped directly into hys logic, because the destination is always the id of a basic block. The same behavior is applied to the jump instructions `j, jr` (jump: , to address), but the semantic of the jump instruction `jal, jalr` (jump: and link, and link to address) differs. These types of jump instructions are used when call instructions are invoked. The hys target machine still jumps to a designated state, but the jump table index of the next instruction is stored in the `ra` (return address) register.

3.1.3.3 Mapping Data Transfer Instructions

The memory instructions `lb, lh, lw, ld` and `sb, sh, sw, sd` from the mips assembly language can be mapped onto the load and store operations provided by the hys memory manager. The prefix of the mips memory instructions stand for:

- b: byte
- w: word
- h: half word
- d: double word

The following example shows the transformation of a function defined in C source code onto the hys target machine.

Example:
In this example the C function `mult`*, which is shown in Figure 3.7, is transformed into the hys language step by step.*

```
int mult(int a, int b) {
   return a * b;
}
```

Figure 3.7: Function mult - C Source Code

The function returns the result of the multiplication of the two input integers a and b. In Figure 3.8 the intermediate representation of LLVM of the function `mult` *is shown.*
For the return value `%retval` *and each of the two input values* `%a` *and* `%b`*, a pointer is created using the function* `alloca`*. In the next step the input values are stored at the addresses of their corresponding pointers* `%a.addr` *and* `%b.addr`*. Then the values are loaded from the memory into the two new registers* `%tmp` *and* `%tmp2` *in order to satisfy the SSA form. In the following step they are multiplied with the result assigned to the register* `%mult`*, which is then stored at the address of the return value pointer* `%retval`*. Finally the return value is loaded from the memory to the new register* `%0` *and returned.*

```
define i32 @mult(i32 %a, i32 %b) nounwind {
entry:
   %retval = alloca i32, align 4
   %a.addr = alloca i32, align 4
   %b.addr = alloca i32, align 4
   store i32 %a, i32* %a.addr
   store i32 %b, i32* %b.addr
   %tmp = load i32* %a.addr
   %tmp1 = load i32* %b.addr
   %mul = mul i32 %tmp, %tmp1
   store i32 %mul, i32* %retval
   %0 = load i32* %retval
   ret i32 %0
}
```

Figure 3.8: Function mult - LLVM IR

In Table 3.3 the mips assembly code and the hys target machine are illustrated, which are both generated from the LLVM IR using the Mips Assembly Writer. The basic block ***entry*** *from the original LLVM IR has been translated to* ***BB0*** *in the mips assembly language. Thus the state in the according hys target machine is:* ***multBB0****. For each mips assembly instruction - shown on the left hand side - the corresponding hys expression or expressions are displayed on the right hand side. In order to reduce the size of the table hys expressions* ***instructionPC = i*** *and* ***instructionPC' = j*** *have been mapped to:* IPC_i *and* IPC_j*'. All registers from the mips assembly language are mapped to a variable inside the hys target machine, get the prefix* ***var*** *and the* ***$*** *is erased as this character is not supported by the hys language. Only the stack pointer (****$sp****) and the return address (****$ra****) do not have the prefix* ***var****.*

*The first mips instruction decrements the stack pointer (****$sp****) with a value of -32, which means that for the function mult, 32 bytes are reserved on the stack. The addition is of type unsigned and therefore the common add operation of the hys language can be used. In the next two instructions the values of register* ***$4*** *and* ***$5*** *are stored in the stack at the positions* ***$sp + 20*** *and* ***$sp + 24****. The hys target machine uses the hys memory manager to store the values on the stack.*

	mips assembly code	hys target machine
1	addiu $sp, $sp, -32	multBB0 ∧ IPC$_0$ → multBB0' ∧ IPC$_1$' ∧ sp' = sp + -32 ;
2	sw $4, 20($sp)	multBB0 ∧ IPC$_1$ ∧ MM_Idle → MM_Store' ∧ ap' = sp + 20 ∧ dataValue' = var_4 ∧ dataSize' = 32; multBB0 ∧ IPC$_1$ ∧ MM_Ready → multBB0' ∧ IPC$_2$'; multBB0 ∧ IPC$_1$ ∧ !MM_Ready → multBB0' ∧ IPC$_1$';
3	sw $5, 24($sp)	multBB0 ∧ IPC$_2$ ∧ MM_Idle → MM_Store' ∧ ap' = sp + 24 ∧ dataValue' = var_5 ∧ dataSize' = 32; multBB0 ∧ IPC$_2$ ∧ MM_Ready → multBB0' ∧ IPC$_3$'; multBB0 ∧ IPC$_2$ ∧ !MM_Ready → multBB0' ∧ IPC$_2$';
4	lw $2, 20($sp)	multBB0 ∧ IPC$_3$ ∧ MM_Idle → MM_Load' ∧ ap' = sp + 20 ∧ dataSize' = 32; multBB0 ∧ IPC$_3$ ∧ MM_Ready → multBB0' ∧ var_2' = dataValue ∧ IPC$_4$'; multBB0 ∧ IPC$_3$ ∧ !MM_Ready → multBB0' ∧ IPC$_3$';
5	nop	
6	mult $2, $5	multBB0 ∧ IPC$_4$ ∧ ALU_Idle → ALU_bit2Signed' ∧ ALU_op_reg' = 10 ∧ ALU_i1_reg' = var_2 ∧ ALU_i2_reg' = var_5; multBB0 ∧ IPC$_4$ ∧ ALU_Ready → multBB0' ∧ lo' = ALU_o_reg ∧ IPC$_5$'; multBB0 ∧ IPC$_4$ ∧ !ALU_Ready → multBB0' ∧ IPC$_4$';
7	mflo $2	multBB0 ∧ IPC$_5$ → multBB0' ∧ var_2' = lo ∧ IPC$_6$';
8	sw $2, 16($sp)	multBB0 ∧ IPC$_6$ ∧ MM_Idle → MM_Store' ∧ ap' = sp + 16 ∧ dataValue' = var_2 ∧ dataSize' = 32; multBB0 ∧ IPC$_6$ ∧ MM_Ready → multBB0' ∧ IPC$_7$'; multBB0 ∧ IPC$_6$ ∧ !MM_Ready → multBB0' ∧ IPC$_6$';
9	addiu $sp, $sp, 32	multBB0 ∧ IPC$_7$ → multBB0' ∧ IPC$_8$' ∧ sp' = sp + 32 ;
10	jr $ra	multBB0 ∧ IPC$_8$ → JUMP' ∧ JTI' = ra;
11	nop	

Table 3.3: Mips Assembly Language vs hys Language

Instruction number four assigns the value of stack address: `$sp + 20` *to register* `$2`*, which also uses the hys memory manager. The fifth instruction is a* `nop`*, which follows each jump or load instruction and is not translated to any hys operation. The actual multiplication of the function* `mult` *takes place at instruction number six. In this case, as the* `mult` *instruction is of type signed, the hys target machine must use the hys arithmetic logic unit to compute the result. The next instruction number seven assigns the result of the multiplication, which is stored in the* `lo` *register of the mips machine, to register* `$2`*. Instruction eight stores the value of register* `$2` *at the stack position:* `$sp + 16`*. The next instruction - number nine - increments the stack pointer* `$sp` *with a value of 32, which frees the stack space reserved for the current function* `mult`*. Finally instruction number ten jumps the address stored in the return address register* `$ra`*. In the hys target machine the jump is handled by the jump index table, which is part of the hys state machine. The last instruction number eleven is another* `nop` *instruction, which as mentioned before has to follow each jump instruction.*

3.2 Conclusions

In this chapter it is shown that it is possible to convert the intermediate representation of LLVM into the hys language for the purpose of finding a witness for possible runtime errors or proving their absence. In order to avoid the tasks when mapping the LLVM IR directly into the hys language, which are illustrated in [Chapter 3.1.2], the Mips Assembly Writer of the LLVM Backend has been used for the transformation. The hys target machine, which is described in [Chapter 3.1.1], runs on the instructions of the mips assembly language, which are generated from the LLVM Backend. So it actually models a real mips architecture and if a solver finds a solution for the model, this solution is a witness for the runtime error, which has occurred on the mips machine.

In order to provide a dynamic target architecture, which is a goal for the future, it is possible to create a new target machine for the LLVM Backend. In this case the user can adopt the target machine to the architecture he needs the program to be verified against.
Another interesting idea for future work is to vary the number of registers and the stack size, as the complexity of the model described in the hys target machine might change when the machine is equipped with more static registers, or the number of registers is decreased and therefore the stack size is increased.

In the following section a short description on how to create a new target machine for the LLVM Backend is given.

3.2.1 Creating a new Target Machine

For the creation of new Target Machine for the LLVM Backend the following steps have to be performed:

1. Create a subclass of TargetMachine,
2. describe the register set with tablegen,
3. describe the instruction set with tablegen,
4. describe the selection and conversion from instructions in LLVM IR to target instructions (DAG $\rightarrow$ DAG) and
5. create an assembly writer which produces hys code.

For a detailed documentation on how to create a Backend for LLVM, please refer to [29].

Chapter 4

Probabilistic SAT Solving

In this chapter the implementation of the algorithms [Chapter 2.2.2.1] introduced by Moser to the Lovász Local Lemma [23], testcases to it and their according results - which is the second task of this thesis - are described.

4.1 Implementation of the LLL algorithms

The implementation of the algorithms introduced by Moser to the Lovász Local Lemma is based on the process illustrated in Figure 4.1.

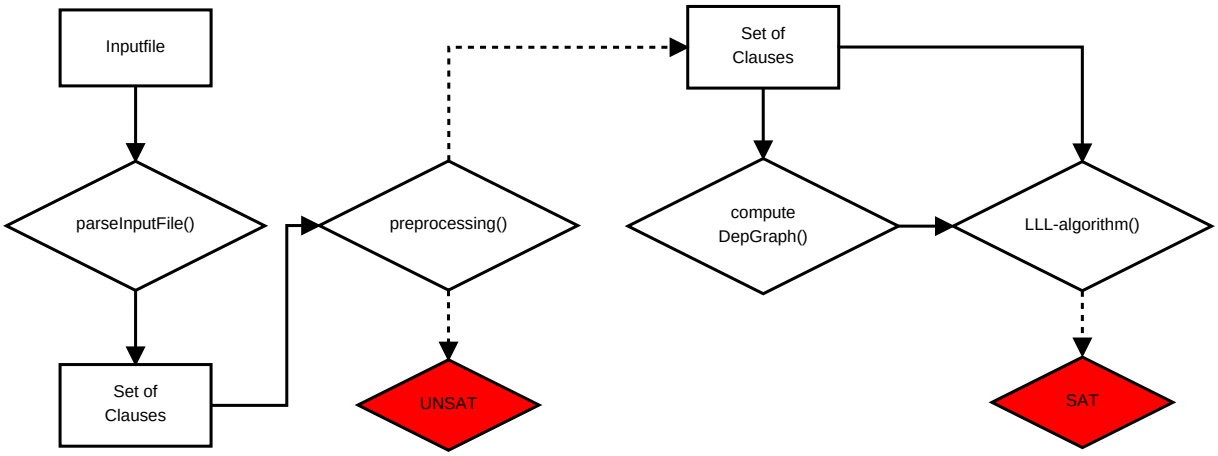


Figure 4.1: Roadmap: Implementation of the LLL

The process can be separated in three parts. First the input formula is parsed into an internal representation, which is a set of clauses. In the second step preprocessing based on unit propagation [Definition 11] is performed on the set of clauses, which either constructs a possibly reduced set of clauses or may lead to an early termination

returning UNSAT. This step is not part of Moser's algorithms, but as this feature might result in a possible performance gain, it is inserted before the actual LLL algorithm takes place. If the process is still running, the third part is entered, in which the dependency graph over the set of clauses is computed and the LLL condition $d \leq 2^{k-5}$ is checked. If the check fails, the user is notified by a message and has the choice to terminate the program. In the following step one of the LLL algorithms is invoked on the set of clauses, which either finds a satisfying assignment to the input formula - in at most polynomial time, if the condition is fulfilled - and returns SAT or might keep running "forever".

4.1.1 Classes

For the implementation it has been necessary to design suitable structures. As the entire implementation is in C++, the following classes have been introduced: Assignment, Clause, DependencyGraph, Model, Search. The relationship between these classes is shown in Figure 4.2.

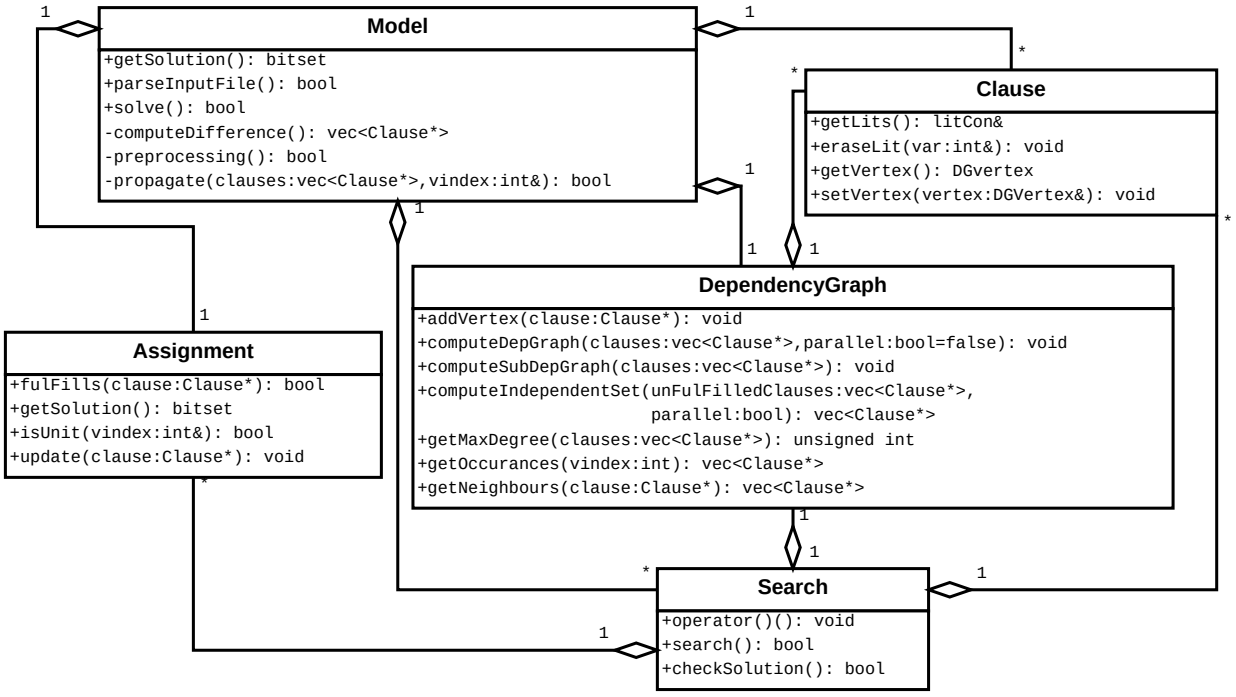


Figure 4.2: Dependency View of the Classes: UML Class Diagram

In the succeeding subsections the classes and their functionality are described in detail.

Class Assignment

The class Assignment represents the assignment through which it is tried to satisfy the input formula. In Figure 4.3 the UML diagram of the class is shown.

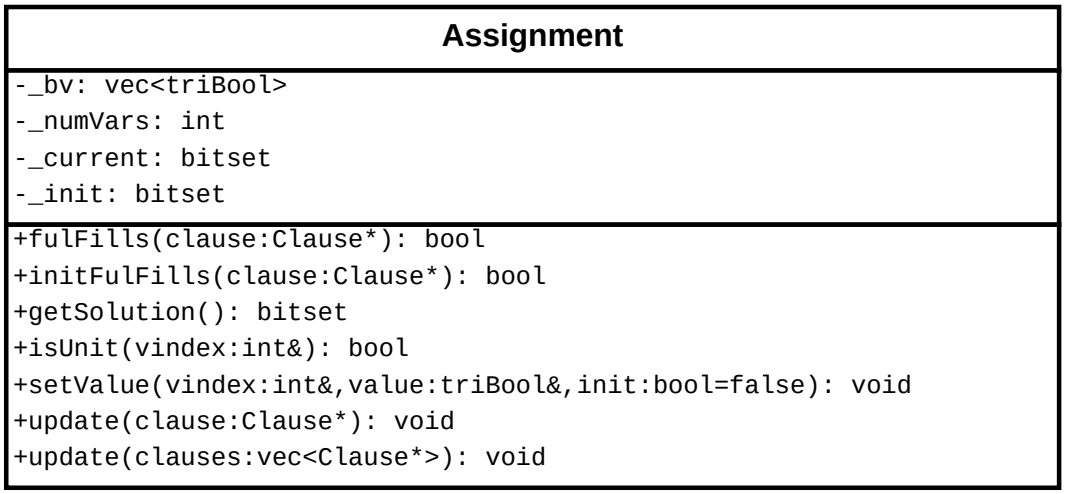


Figure 4.3: Class Assignment: UML Class Diagram

The member `_numVars` represents the number of variables of the input formula. The elements: `_bv, _current` and `_init` are containers of the size `_numVars`. The vector `_bv` is introduced for preprocessing, because therefore the initial assignment has to be undefined. During preprocessing each variable occurring or becoming unit is assigned with the according truth value, which is set using the function `setValue()` in `_bv` and `_init`. In the bitvector `_current` the actual assignment is stored, which can be updated using the functions `update()` at the variable positions of either one clause or a set of clauses. The function `fulFills()` (analog: `initFulFills()`) checks whether the `_current` (`_bv`) assignment satisfies the clause or not. To check if a variable has been set due to unit propagation the function `isUnit()` is used. Finally the `getSolution()` returns the bitvector `_current`.

Class Clause

The class Clause incorporates a clause from the input formula. The UML diagram of the class is illustrated in Figure 4.4.
The class consists of a vector of literals stored in `_lits`, which must contain at least one. Each clause is also equipped with a `_vertex`, which represents its node inside the dependency graph. Via the function `getLits()` a vector containing all literals of the clause is returned. In order to erase a literal from a clause the function `eraseLit()` is used. The functions `getVertex()` and `setVertex()` are introduced in order to retrieve or set the node of a clause.

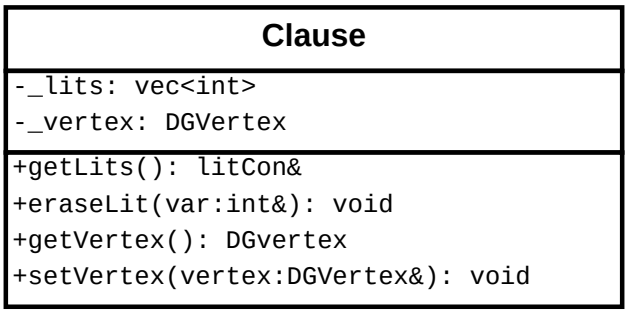


Figure 4.4: Class Clause: UML Class Diagram

Class DependencyGraph

The class DependencyGraph is shown in Figure 4.5. It contains a matrix in which the occurrence of a variable in a clause is stored. This is realized as a vector of vectors `_varDeps`. From this matrix the dependency graph is built using the function `computeDepGraph()` over the set of clauses in the model, which is located in `_dependencyGraph`. The member `_subDepGraph` - representing a sub-dependency graph to an according set of clauses - is built using the function `computeSubDepGraph()`, but is only invoked when the parallel version of the LLL algorithm is executed.

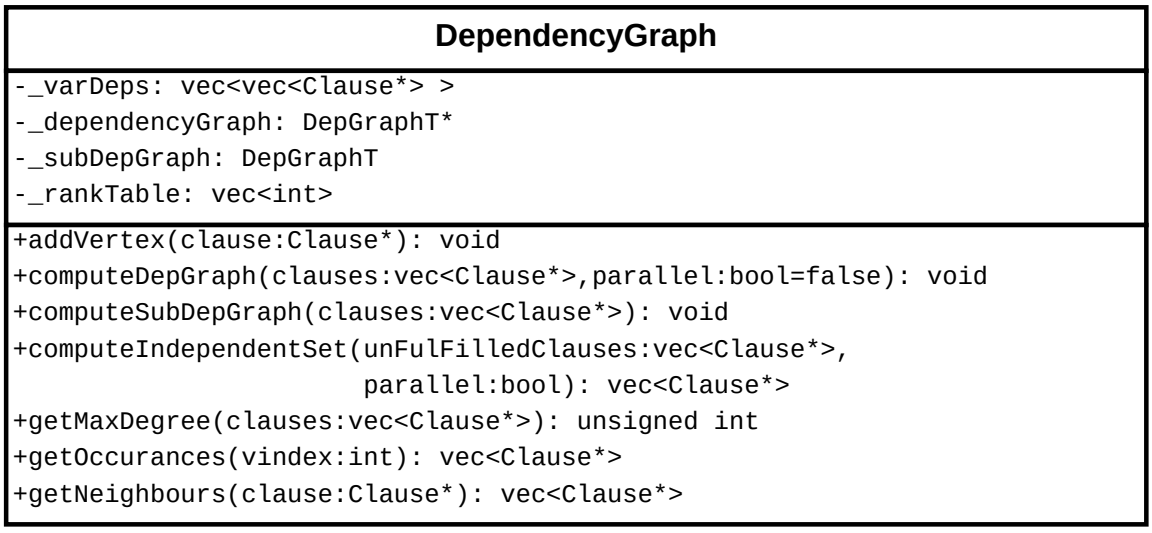


Figure 4.5: Class DependencyGraph: UML Class Diagram

The function `computeSubDepGraph()` is constructed based on the algorithm for the computation of the maximal or minimal set of Luby [18]. In the original algorithm from Luby each node is associated with a processor, but as a computer in the present

has a maximum of four processors with 8 or 12 cores each, it is only possible to run at most 48 threads in parallel. So in most of the cases, where the input formula has more than 10000 or even 100000 clauses, it is not possible to reach the runtime theoretically proven for Luby's algorithm.
For testing purposes the variable `_rankTable` has been introduced, which shows the number of clauses of degree `i` for each `i` in $0 \leq i \leq d$.

Class Model

The class Model, which is illustrated in Figure 4.6, represents the basis for the process described in Figure 4.1.

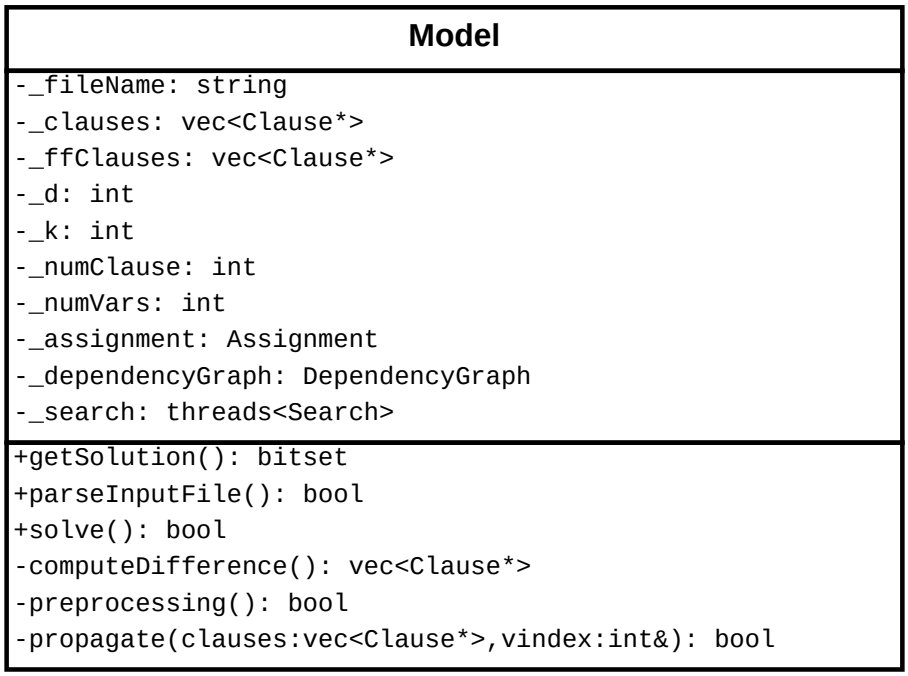


Figure 4.6: Class Model: UML Class Diagram

With the function `parseInputFile()` the formula contained in `_fileName` is read and the clauses are stored in `_clauses` as well as the maximal number of literals in a clause is written to `_k`. In the next step the function `solve()` is invoked, which starts with a call to the function `preprocessing()`. `preprocessing()` works on the set of clauses, which computes the unit clauses and propagates their assignments through the set of clauses, storing each fulfilled clause in `_ffClauses` and returning either false (UNSAT) if unit propagation leads to a conflict or true. If the process is still running, the new set of clauses is computed via the function `computeDifference()` and is passed on to the actual search process, which is represented by the class

Search. The search process can be invoked in multiple threads, where each thread has its own random initial assignment.

Class Search

The class Search - shown in Figure 4.7 - represents the actual implementation of the LLL algorithms.

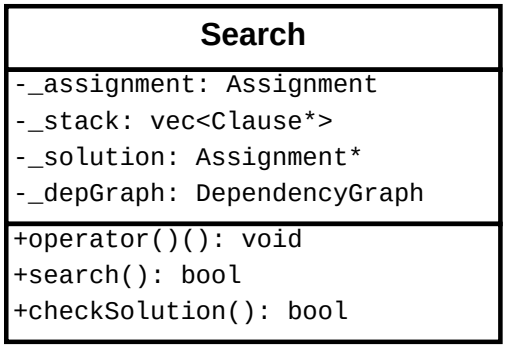


Figure 4.7: Class Search: UML Class Diagram

As input it gets the set of clauses, an initial assignment and the dependency graph. The `operator()()` is necessary, because the search is invoked as a thread from the function `solve()` in class Model. If the function `search()` finds a satisfying assignment, it writes it to `_solution` and notifies the `solve()` process, in order to terminate.

4.2 Testcases

In order to test the performance and to verify the behavior of the implemented algorithms testcases are needed. In general - when testing SAT Solvers - the benchmarks from the SAT competition web pages are used as testcases. But those benchmarks are not used in this thesis as almost every benchmark from these sites does not fulfill the condition required from the Lovász Local Lemma ($d \leq 2^{k-5}$).

4.2.1 Generation

A cnf generator has been created with which it is possible to create randomized testcases fulfilling the condition or distracting it, thus to provide testcases and to discuss the behavior of the LLL implementation. The generator is based on the algorithm shown in Figure 4.8.

The generator requires the parameters: k for the number of literals of a clause, `numClauses` for the maximum number of clauses in the formula, `numVars` for the number of variables in the formula and `offset` for being able to generate testcases violating the LLL condition. First of all the bound for LLL is computed and the set of clauses is initiated. In the following the actual computation of the clauses is started. Therefore a variable is randomly picked and it is checked that the sum of the maximum number of neighbors of any clause containing the variable and the current neighbors is not greater than the bound. If the check is satisfied and the variable is not already in the clause either the positive or negative value of the variable is added to the clause. Once a clause has reached the maximum size of k or there exists no new variable to choose, the clause is added to the set of clauses and the generation of the next clause is started.

```
generate(k, numClauses, numVars, offset):
   bound = 2^(k-2+offset)
   clauses = []
   while (len(clauses) < numClauses):
     clauselLen = 0
     neighbors = 0
     lits = []
     while (clauseLen < k):
       var = random.randint(1,numVars)
       maxNeighbors = getMaximumNumberofNeighbors(var)
       if (neighbors + maxNeighbors < bound):
         if ((var not in lits) and (-var not in lits)):
           if (random.randint(0,1) == 0):
             lits.append(var)
           else:
             lits.append(-var)
           neighbors += maxNeighbors
         clauseLen++
     clauses.append(lits)
```

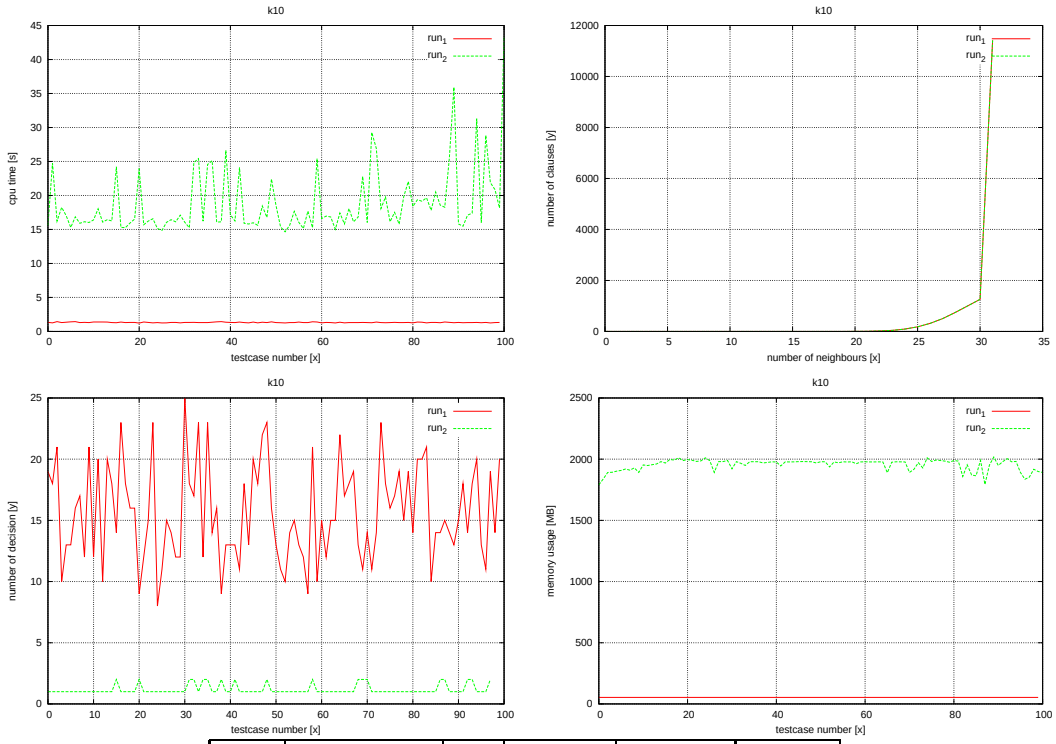
Figure 4.8: Random Testcase Generator: Algorithm

For the verification of the correct behavior of the implemented algorithm any testcase satisfying the bound for the LLL condition $d \leq 2^{k-5}$ is suitable. But in order to test the performance the upper corner case of the bound is of interest, where most of the clauses are at the bound. It is also interesting to see how the algorithms perform if the bound is violated by almost every clause.
In order to create usual testcases it is recommendable to choose the number of variables to be larger than the number of clauses per testcase. If a violation of the

bound of the LLL condition is of interest, the number of clauses has to outnumber the number of variables per testcase a lot.

4.2.2 Analysis of the Results

All the testcases have been run on a Windows XP System running on an Intel(R) Core(TM)2 Duo CPU with 2.53GHz clock frequency and 2.99GB RAM. This hardware specification is good enough for the sequential algorithm, but not quite excellent for the parallel algorithm - as it presumes one processor core per clause. The parallel variant has still been tested, but only with some small testcases.



run	algorithm	d	bound	clauses	vars
1	sequential	31	$2^{k-5.0}$	15597	43632
2	parallel	31	$2^{k-5.0}$	15597	43632

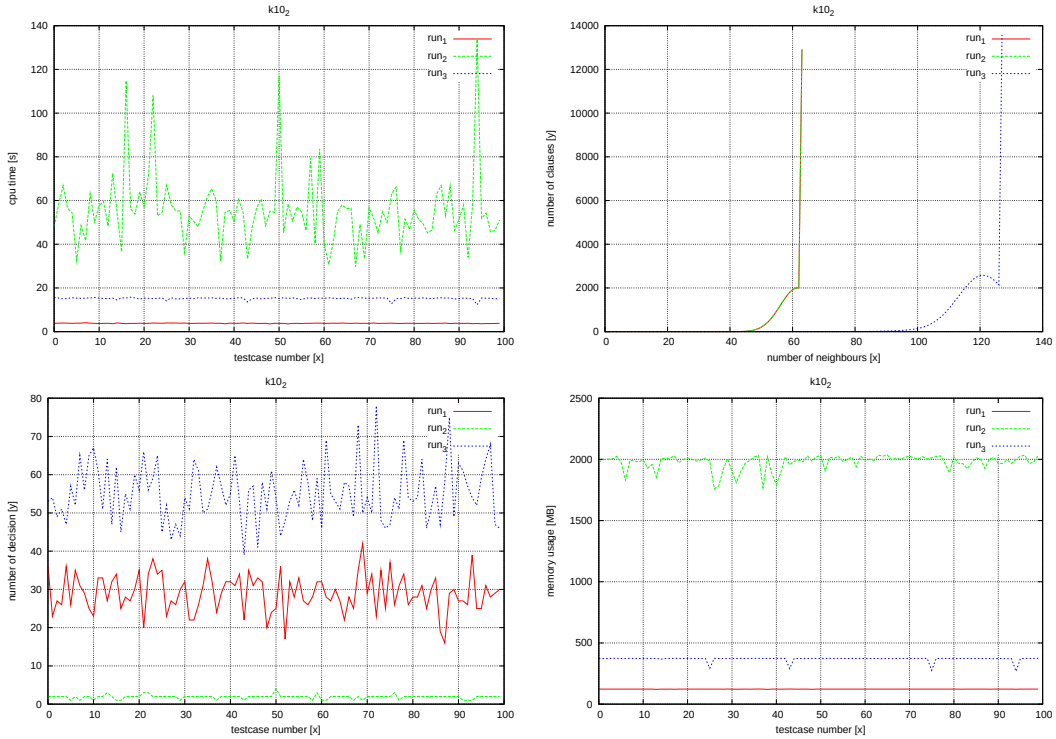
Figure 4.9: Random generated testcases with $k = 10$

In Figure 4.9 the testcases with $k = 10$ are shown. The testcases have been generated using the random testcase generator with the request to create a formula, which meets the LLL condition $d \leq 2^{k-5}$, having a maximum of 2134023 clauses and 43632 variables. As the number of clauses outnumbers the number of variables and the bound met, the generator creates testcases where most of the clauses have the maximum number of allowed neighbors, which is due to the definition of the

neighborhood relationship $\Gamma = 2^{k-5} - 1 = 31$. This is shown on the upper right chart in Figure 4.9. The two charts on the left show the CPU time (upper chart) and the according number of decisions (lower chart) made by the algorithms. In case of CPU time the sequential algorithm wins against the parallel one, but this is due to the lack of available hardware for the test (as mentioned before). But in respect to the number of decisions made, the parallel algorithm outperforms the sequential one, with the number of decisions: $log(m)$ for the parallel algorithm, where the sequential one lasts m decisions. This result confirms the statement about the total number of resampling steps from the paper [20]:

$$\sum_{A \in \mathbb{A}} \frac{x(A)}{1-x(A)} \text{ for the sequential and } \frac{1}{\epsilon} log\left(\sum_{A \in \mathbb{A}} \frac{x(A)}{1-x(A)}\right) \text{ for the parallel algorithm}$$

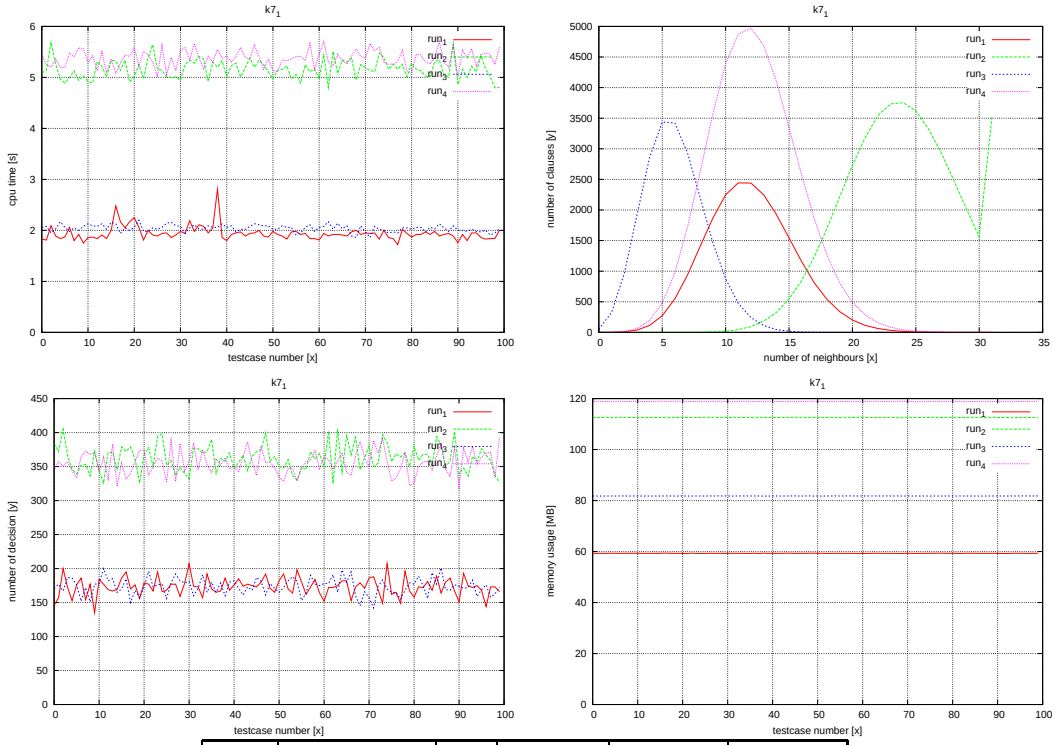
Where $x(A)$ is the probability that event (clause) A is violated (not fulfilled) under the current assignment and in respect $1 - x(A)$ is the probability that event A is not violated.



run	algorithm	d	bound	clauses	vars
1	sequential	63	$2^{k-4.0}$	28400	43632
2	parallel	63	$2^{k-4.0}$	28400	43632
3	sequential	127	$2^{k-3.0}$	28400	43632

Figure 4.10: Random generated testcases with $k = 10$

The testcases shown in Figure 4.10 are also of length $k = 10$, but do not meet the bound of LLL condition any more with $d = 63$ and $d = 127$. From the neighbor chart in the upper right it is observable that all clauses violate the condition $d \leq 2^{k-5}$ in both of the testcases, in the second one even by far. But still - as shown on the decision chart on the lower left - the sequential algorithm performs in maximum number of $\sum_{A \in \mathbb{A}} \frac{x(A)}{1-x(A)}$ steps. And even the parallel algorithm works in predicted time $log(\sum_{A \in \mathbb{A}} \frac{x(A)}{1-x(A)})$. The CPU time on the upper left chart varies a lot for the parallel algorithm, whereas it is almost equal for the sequential one.

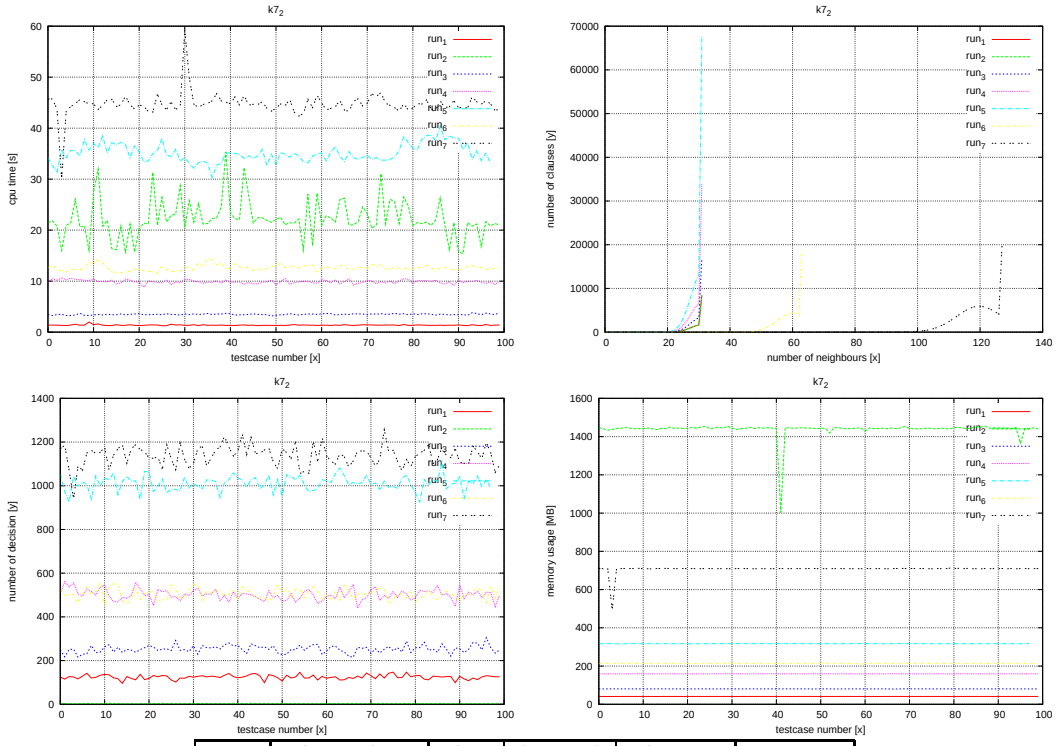


run	algorithm	d	bound	clauses	vars
1	sequential	31	$2^{k-2.0}$	21340	87264
2	sequential	31	$2^{k-2.0}$	43632	87264
3	sequential	31	$2^{k-2.0}$	21340	174528
4	sequential	31	$2^{k-2.0}$	43632	174528

Figure 4.11: Random generated testcases with $k = 7$

In Figure 4.11 the testcases, which have also been generated using the random test generator, with $k = 7$ are shown. This time the bound for the LLL condition is violated quite far with $d = 31$, where $2^{k-5} = 4$. But for the generation the number of variables has been chosen greater in respect to the number of clauses per testcase. The according neighbor chart is shown on the upper right and illustrates for all

the testcases a normal distribution over the clauses according to their number of neighbors. With an increasing number of clauses on a fixed amount of variables the distribution drifts towards the bound. The peak of the curve is raised in respect to the increased number of clauses. Responsible for the occupied amount of memory used during the Search is also the number of clauses, which is shown on the lower right chart. But from the CPU time chart - on the upper left - it is observable that most of the testcases with the same amount of clauses have almost the same computation time even if the number of variables differs a lot. This result is also observable from the decision chart - shown on the lower left. So the number of decision steps is bound to the number of clauses per testcase and not to the amount of variables. The sequential algorithm still operates in expected time.



run	algorithm	d	bound	clauses	vars
1	sequential	31	$2^{k-2.0}$	15034	21816
2	parallel	31	$2^{k-2.0}$	15034	21816
3	sequential	31	$2^{k-2.0}$	30062	43632
4	sequential	31	$2^{k-2.0}$	60123	87264
5	sequential	31	$2^{k-2.0}$	120236	174528
6	sequential	63	$2^{k-1.0}$	56333	43632
7	sequential	127	$2^{k-0.0}$	109427	43632

Figure 4.12: Random generated testcases with $k = 7$

The results from Figure 4.11 have shown that for the average testcases of length $k = 7$ the algorithm computes in expected time. In order to check the corner cases again, testcases with $k = 7$ have been generated where the amount of clauses (2134023) outnumbers the amount of variables ($\{21816, 43632, 87264, 174528\}$)by far. The results of those testcases are shown in Figure 4.12.
On the decision chart it is still observable that the algorithms behave in the expected way. So the sequential algorithm is still operating in $\sum_{A \in \mathbb{A}} \frac{x(A)}{1-x(A)}$ and the parallel one in $log(\sum_{A \in \mathbb{A}} \frac{x(A)}{1-x(A)})$ number of decision steps. The already observed behavior on the number of decisions steps is also noticeable on the lower left chart of Figure 4.12. The testcases of run_3, run_6 and run_7 have the same amount of variables, but differ a lot on the number of clauses, as does the number of decision steps with an increasing number of clauses. But the testcases of run_4 with 60123 clauses and of run_6 with 56333 clauses have almost the same amount of clauses and the sequential algorithm needed almost the same number of decision steps for them. Also the testcases of run_5 and run_7 are close together in respect to the number of decisions, but with a difference of about 10000 clauses. The rest of the already observed behavior is also noticeable on these results here.

4.3 Conclusions

From the results shown in [Chapter 4.2.2] the following conclusions can be made:

1. The algorithms provided by Moser and Tardos behave in the way they have been predicted to and meet the according maximal number of decision steps:
   - $\sum_{A \in \mathbb{A}} \frac{x(A)}{1-x(A)}$ for the sequential algorithm and
   - $log(\sum_{A \in \mathbb{A}} \frac{x(A)}{1-x(A)})$ for the parallel algorithm.
2. But furthermore, if the constraint is violated by far like in the testcases from Figure 4.12, the algorithms still perform in at most polynomial time.

In the future it would be interesting to cluster the parallel algorithm in order to increase its computation time and really make use of the advantages of using as many processor cores as possible for the computation of the maximal independent set of violated clauses.

Another interesting quest would be to create a solver, where the common features of SAT-solving are combined with the parallel algorithm of Moser. In the following there are some ideas how this interaction could be performed:

1. The trivial one, in which one or several threads of the common search process operate on the same assignment stack and in another thread the parallel algorithm of Moser runs on a separate assignment stack. So if the formula satisfies the constraint $d \leq 2^{k-5}$, the algorithm will find in at most polynomial time a solution for the formula unless the other threads have not found one in advance. And if the formula fulfills the constraint $d \leq 2^{k-2}$ at least one of the SAT threads has to find a solution, as the constraint states that the formula is satisfiable.

2. Again one or several threads perform the common search process operating on the same assignment stack and in another thread the parallel algorithm of Moser runs on a separate assignment stack. But in this case the assignment stack of Moser's algorithm gets updated with the current assignment from the other threads. So the algorithm only updates those variables which are not assigned under the current assignment of the common search process.

3. Another idea is to abuse the Dependency Graph to split the input formula into its independent subsets of clauses. Each of these subsets can then be assigned to a separate SAT or LLL thread.

4. If the input formula does not meet the LLL constraint, the following approach can be applied:

   (a) Perform the common SAT search based on the DPLL algorithm.

   (b) After each decision step the variable which has just been assigned is deleted from all clauses it occurs in, if the current assignment does not lead to a conflict. Still the occurrences must be saved somewhere in order to provide a proper backtracking mechanism.

   (c) Until the "new" formula does not meet the LLL constraint, go to (a).

   (d) The formula does now fulfill the LLL constraint and therefore a solution must exist.

   (e) Use for example the approach from 3. to split the "new" formula into its independent subsets and compute the solution in parallel.

5. Usually the propagation process of the SAT can result in one conflict clause, on which conflict analysis is performed in the next step. But due to the current assignment there might be an entire set of conflict clauses. So if this entire set is computed, Luby's algorithm can be used to compute the maximal independent set of clauses, which are currently unfulfilled. In the next step the conflict analysis can then be performed in parallel threads on each of these clauses from the set. On the other hand it might also be interesting to compute the maximal dependent sets of violated clauses in order to find that clause from these sets, which has been assigned the earliest in respect to the decision level.

Chapter 5

Conclusions

In this thesis two challenges have been dealt with: one in the domain of Program Analysis and the other in the area of Probabilistic SAT-solving.

In the Program Analysis part [Chapter 3] it has been shown that the intermediate representation of LLVM describing a program can be transformed into a SMT formula in order to find a witness or to prove the absence of runtime errors in the program. The chosen SMT format is the hys language with which it has been possible to model the program as a state machine. Therefore it was not necessary to apply loop unrolling on the program, because the solvers operating on the hys language perform bounded model checking. The solvers can also take advantage of the learning process in which earlier conflicts of the search process can be reused forward and sometimes even backward.

The Probabilistic SAT-solving part [Chapter 4] has been the implementation and evaluation of the algorithms Moser introduced in his works on the Lovász Local Lemma. The results have shown that the algorithms reside in the runtime Moser has predicted and that they even scale pretty good if the constraint the input formula is bound to is violated. The conclusions have also lead to some interesting ideas, which might improve the SAT- and SMT-solving processes.

The results of the Probabilistic SAT-solving part are also applicable on the Program Analysis part. This conclusion can be made due to the fact that a SMT formula is an extended SAT formula, which can also be treated as a set of events, where each event (clause) is determined by a finite set of mutually independent random variables. So if the solver operating on the hys language would be extended with a method to compute the dependency graph over the clauses of the input formula for each iteration and if this graph fulfills the structural constraint that every event has at most 2^{k-2} neighbors, the Lovász Local Lemma predicts that it must be possible to find a witness. This might be a huge advantage for the user, because an input

problem of large size usually results in a long decision time for the solving process. But if the user already knows that a solution for the formula exists, it is worth to wait for the solver to find the witness.
In order to provide such information also the fourth idea from the conclusions in [Chapter 4.3] might be used. In this idea the size of the input problem, which does not fulfill the LLL constraint by default, gets reduced step by step of all those variables, which have been assigned without raising a conflict in the search process. After each reduction step the LLL constraint gets checked again and once it is satisfied there must exist a solution for the remaining problem.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Kim Allan Andersen and Daniele Pretolani. Easy cases of probabilistic satisfiability. *Ann. Math. Artif. Intell.*, 33(1):69–91, 2001.

[3] John Aycock and Nigel Horspool. Simple generation of static single assignment form. In *Proceedings of the 9th International Conference in Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2000.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003.

[6] Robert Britton. *MIPS Assembly Language Programming*. Pearson Education, 2003.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[9] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[10] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, July 1960.

[11] Leonardo de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: An Appetizer. In Marcel V. Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications*, volume 5902, chapter 3, pages 23–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[12] Dominic Fandrey. Clang/LLVM Maturity Report. June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[13] Martin Fränzle and Christian Herde. Efficient proof engines for bounded model checking of hybrid systems. *Electron. Notes Theor. Comput. Sci.*, 133:119–137, May 2005.

[14] John Hopcroft and Jeffrey Ullman. *Introduction to automata theory, languages, and computation.*, pages 185–192. Addison-Wesley, 1979.

[15] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. Taylor and Francis, 2009.

[16] Chris Lattner. *LLVM Language Reference Manual*. http://llvm.org/docs/LangRef.html.

[17] Chris Lattner. *Low Level Virtual Machine LLVM*. http://llvm.org.

[18] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15:1036–1055, November 1986.

[19] Robin A. Moser. A constructive proof of the lovász local lemma. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 343–350, New York, NY, USA, 2009. ACM.

[20] Robin A. Moser and Gábor Tardos. A constructive proof of the general lovász local lemma. *J. ACM*, 57(2):1–15, 2010.

[21] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[23] Paul Erdös and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In *COLLOQUIA MATHEMATICA SOCIETATIS JAN05 BOLYAI 10. INFINITE AND FINITE SETS, KESZTHELY (HUNGARY)*, 1973.

[24] Emad Saad. Probabilistic reasoning by sat solvers. In *Proceedings of the 10th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, ECSQARU '09, pages 663–675, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] Michel Sintzoff. Calculating properties of programs by valuations on specific models. In *Proceedings of ACM conference on Proving assertions about programs*, pages 203–207, New York, NY, USA, 1972. ACM.

[26] Dominic Sweetman. *See MIPS Run, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[27] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[28] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, 1998. Springer-Verlag.

[29] Mason Woo and Misha Brukman. *Writing an LLVM Compiler Backend*. http://llvm.org/docs/WritingAnLLVMBackend.html.

[30] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30:1–36, March 2005.

List of Definitions

List of Figures

List of Tables