Combining SAT Solving and Integer Programming for Inductive Verification of Lustre Programs

Anders Franzén

Department of Computer Science and Engineering
Chalmers University of Technology
2004

Abstract

This thesis examines whether a combination of SAT solving and integer programming for inductive verification of safety properties for Lustre programs is a viable alternative to other verification methods. The two methods used for comparison are inductive verification based on SAT solving as implemented in Luke, and abstract interpretation using the NBAC tool. Luke is a tool developed at Chalmers Univerity of Technology for use in a course on software engineering using formal methods, and NBAC has been developed by Bertrand Jeannet at IRISA in France during his PhD studies.

Two methods of combining the decision procedure, offline and online integration are presented. Together with methods of analysing infeasible integer programming problems and a preprocessing stage of constraints, these methods have been implemented in a tool based on Luke. This tool is used to evaluate the different options for the decision procedure, and also to compare these methods to Luke and NBAC.

The method of combining decision procedures is shown to be complementary to existing methods of verification of Lustre programs.

Preface

This Masters' thesis work has been done as the concluding part of a Masters program in Computer Science and Engineering. The project gives 20 Swedish credit points and is equivalent of 20 weeks full time work.

The thesis supervisor has been Koen Claessen at the department of Computing Science at Chalmers University of Technology.

Göteborg, December 2004

Contents

Chapter 1

Introduction

The purpose of this thesis is to evaluate a new decision procedure for formal verification of Lustre programs.

1.1 Formal verification

A formal method is a method which uses a formal notation for mathematical modelling and analysis. With a mathematical model, it is possible to analyze a software program formally, and even construct mathematical proofs of correctness. Verifying software in this way is called formal verification.

The verification method used in this thesis is temporal induction, which is induction over time. The base case is used to prove that the property is true during the k first time points. In the step case, it is proven that if the property is true in k consecutive time points, then it is also true in the following time point.

1.1.1 Translation to the logic

The formal language used in verification is propositional logic extended with integer linear constraints with free variables. Lustre programs can be translated to this language in one of three ways; By creating one constraint for each operator used, by creating one constraint for each expression containing only addition, subtraction or multiplication. The last variant creates guarded constraints.

1.1.2 The decision procedure

The decision procedure is a combination of SAT solving and integer linear programming. A propositional approximation of the original formula is created such that the approximation is unsatisfiable only if the original formula is unsatisfiable. A SAT solver is used to decide satisfiability of the approximation. If it is satisfiable, the SAT model is checked against

the integer constraints in the original formula. If the model conflicts with the constraints, the propositional formula is refined by adding information about the constraints. This is reapeated until either a model for the original formula can be constructed, or the propositional formula becomes unsatisfiable.

1.2 Outline of the thesis

In chapter 2 the programming language Lustre is introduced, together with proposition logic and integer programming. A short explanation of the verification method of induction is also described.

Chapter 3 describes the formal language which will be used in the verification procedure. The chapter contains both an informal description of the language, and a more formal definition of the language and its semantics for the interested reader. The less interested reader may safely skip the formalities.

In chapter 4 it is described how this language is used to describe Lustre programs and their properties.

The decision procedures used for deciding satisfiabílity are described in chapter 5. In chapter 6, the problem of infeasible integer programming problems is discussed, and several algorithms for handling these are described.

The different methods of verifying Lustre programs are evaluated in chapter 7. Related work is described in chapter 8, and the conclusions of the experiments are summarized in 9 together with a desciptions of the future work.

1.3 Aim of the thesis

The thesis aims at evaluating the usefulness of a combined SAT/integer programming decision procedure in inductive verification of Lustre programs. These types of decision procedures have been shown to be useful in other areas, such as planning or verification of discrete-continuous systems. The method developed in the thesis will be compared to two freely available tools for Lustre verification; Luke (see section 8.1), which uses induction on top of a SAT solver, and NBAC (see section 8.1) which uses abstract interpretation.

1.4 Acknowledgements

I would like to thank my supervisor Koen Claessen for his support, his good advice, and the idea of the thesis subject, David Merchat at VERIMAG in Grenoble, France for contributing many of the Lustre programs used for evaluating the tool. Also Bertrand Jeannet at the “Institut de recherche

en informatique et systèmes aléatoires" (IRISA), in Rennes, France for the use of his verification tool NBAC, and his invaluable assistance on how to use it properly. For the statistical analysis, the help of Stefan Franzén was instrumental in developing a sound test plan and analysis method for the evaluation.

Any errors or incorrect statements in the thesis are mine alone.

Chapter 2

Background

As a background to the thesis, this chapter describes the Lustre programming language. Also included is an introduction to propositional logic and Davis-Putnam style SAT solvers, as well as integer linear programming.

2.1 Lustre

This is an informal introduction to the programming language Lustre. For more information, see [31, 33].

A Lustre program consists of *nodes*. A node operates on *streams*, which are an infinite sequence of values of a certain type. A node has zero or more input streams, and one or more output streams. The node defines its output streams in a declarative style. The interface of a node specifies its name, inputs and outputs:

```
node AddOne( X : int ) returns ( Y : int );
```

The outputs are defined using equations

```
Y = X + 1;
```

A node definition would then become

```
node AddOne( X : int ) returns ( Y : int );
let
    Y = X + 1;
tel
```

Here we have a node where the values of the output stream Y is the value of the input stream X, plus one. When given the input stream $[0, 1, 2, 3, \ldots]$, the node will produce the output stream $[1, 2, 3, 4, \ldots]$. There can be more

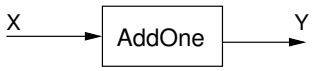

Figure 2.1: The AddOne node

than one input stream, as in this example, where two integers are added together:

```
node Add( X, Y : int ) returns ( Z : int );
let
    Z = X + Y;
tel
```

Sometimes it is necessary to refer to old values of streams, this is done with the **pre** operator. The operand of **pre** is evaluated in the previous time point. Since there is no values before the initial time point, expressions involving **pre** may need an alternate definition for the initial time point. This is accomplished with the "followed by" operator →, which takes two operands. In the inital time point, it evaluates to the left operand, and in all others the right operand.

```
node Accumulate( X : int ) returns ( Y : int );
let
    Y = 0 → X + pre Y;
tel
```

In this example, given the input stream $[1, 2, 3, 4, \ldots]$, we get 0 in the first time point because of the use of the → operator, and so the stream becomes $[0, 2, 5, 9, \ldots]$. This version of Accumulate accumulates all but the first value on the input stream. To accumulate all values on the input stream, a small change is necessary:

```
node Accumulate( X : int ) returns ( Y : int );
let
    Y = X + ( 0 → pre Y );
tel
```

The right operand of the addition is now 0 in the first time point, and the previous value of Y in all others.

2.1.1 Local streams

A node can have local, internal streams as well. These are declared before the definitions. Local streams must be defined, in the same way as output streams.

```
node Counter( X : int ) returns ( Y : int );
    var C : int;
let
    C = 0 → pre C + 1;
    Y = X + C;
tel
```

The local stream C becomes $[0, 1, 2, \ldots]$, regardless of the values on the input stream. Given the input $[1, 2, 3, \ldots]$, the output stream becomes $[1, 4, 6, \ldots]$.

2.1.2 Booleans and if-then-else

Lustre also support boolean streams as we can see below.

```
node Counter( X : bool ) returns ( Y : int );
    var PY : int;
let
    PY = 0 → pre Y;
    Y = if X then PY+1 else PY;
tel
```

Here the **if**−**then**−**else** expression is used to increment the value of the output stream at all time points where the input stream is true. With the input stream alternating between true and false, the output stream would become

Variable	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
X	true	false	true	false	true
PY	0	1	1	2	2
Y	1	1	2	2	3

The stream PY is the previous value of the stream Y.

2.1.3 Node instantiation

Nodes can use other nodes by *instantiation*.

```
node Accumulate( X : int ) returns ( Y : int );
let
    Y = 0 → X + pre Y;
```

```
tel

node Equal( X : int ) returns ( Y : int );
   var A1, A2 : int;
let
   A1 = Accumulate( X );
   A2 = -Accumulate( X );

   Y = A1 - A2;
tel
```

2.2 Propositional logic

Propositional logic is composed of propositional variables $p, q, r, \ldots$ and then connectives $\wedge$ (conjuction or "and"), $\vee$ (disjunction or "or") and $\neg$ (negation or "not"). A propostional value can be either true or false, and variables can be connected with the connectives to form propositional formulas. Formulas in propositional logic are often written in *Conjunctive Normal Form* (CNF). Formulas in CNF are built from *literals*, which are either variables or negated variables ($\neg p$). A set of literals is called a *clause*. Clauses are true if at least one of the literals in the clause is true.

$$\{p, \neg q, r\}$$

For clause above is true if p or r is true, or if q is false. A formula on conjunctive normal form is a set of clauses. The formula is true if and only if all clauses in the formula is true. Take for example the formula

$$\begin{array}{l}\{p, \neg q\} \\ \{q, \neg r\} \\ \{p, q, \neg r\}\end{array}$$

This formula is true if both p and q are true. If one of p and q are false, the formula becomes false. A formula which can be made true with some combination of values for the variables is said to be *satisfiable*. Formulas which can not be made true regardless of the value of the variables are *unsatisfiable*.

The SAT problem is the problem of determining satisfiablility for propositional logic. There are several SAT algorithms in existence today. The Davis-Logemann-Loveland procedure, often called DPLL, extended with conflict clauses [24] is the algorithm which is used here.

The algorithm is a branching search procedure. Initially, none of the variables is assigned a value. The procedure starts by selecting one of

the variables which does not have a value, and assigns a previsously untried value (either true or false) to that variable. This is done in the "decide_next_branch" procedure. This new assignment is used to simplify the formula. The procedure "decide" determines what new variable assignments are a direct consequence of the set of currently assigned variables that are needed for satisfiability of the formula. In a clause

$$\{p, \neg q, r\}$$

if both p and r has been assigned to false, then q must be false for the clause to be true. If all variables can be assigned in this way, the formula is satisfiable. In some cases, however, the deduce procedure will run into trouble. If we have assigned both p and r to false in this formula

$$\{p, \neg q, r\}$$
$$\{p, q, r\}$$

Then the varaible q would have to be both true and false to satisfy the formula. This is called a *conflict*, and the DPLL procedure then backtracks, unmaking the assignments that lead to the conflict. Modern implementations of the DPLL procedure also add a *conflict clause* which describes the reason for the conflict. In this case the reason is that both p and q were false at the same time. The conflict clause is a new clause which explicitly forbids these variable assignments

$$\{p, q\}$$

For the clause to be true, at least one of p and q must be true. If the procedure assigns one of these variables to false, the consequence would be that the other variable is assigned to true.

For more information on SAT solving using a DPLL-style algorithm, see for example [50].

2.3 Integer linear programming

A *constraint* is a relation on the form

$$\sum_i a_i x_i \leq b \quad a_i, b \in \mathbb{R}$$

Constraints are often written $\mathbf{a}^t\mathbf{x} \leq b$, where $\mathbf{a}^t$ is a transposed vector of reals and $\mathbf{x}$ is a vector of variables. An *linear programming* (LP) problem is a system of constraints

$$\begin{cases} \mathbf{a_1}^t\mathbf{x} \leq b_1 \\ \mathbf{a_2}^t\mathbf{x} \leq b_2 \\ \quad\quad\vdots \\ \mathbf{a_n}^t\mathbf{x} \leq b_n \end{cases}$$

Algorithm 2.1 The DPLL procedure

```
loop
  decide next branch
  loop
    status ← deduce
    if status = CONFLICT then
      blevel ← analyse conflict
      if blevel = 0 then
        return UNSATISFIABLE
      else
        backtrack
      end if
    else if status = SATISFIABLE then
      return SATISFIABLE
    else
      break
    end if
  end loop
end loop
```

Where the variables are reals. A *solution* to a LP problem is a variable assignment which satisfies all inequalities in the problem. If a problem lacks a solution it is *infeasible*. A problem that is solvable is called *feasible*. For an infeasible system, there are no combinations of values for the variables which satisfy all constraints. Often it is not enough to find an arbitrary solution to a LP problem. Usually, there is a *goal function* which describe a preference for certains solutions. A goal function is a linear expression over the variables on the form

$$\text{goal}(\mathbf{x}) = \mathbf{c}^t\mathbf{x} + c_0$$

The problem is then solved with the added constraint that the goal function must be maximal/minimal.

Integer (linear) programming ILP is just like LP, with the added constraint that the solutions must be integer. This makes the problem of finding optimal solutions much more difficult, though. While linear programming has polynomial time complexity, integer programming is NP-complete.

There are several solutions to the integer programming problem. The algorithms that are most commonly used are branch-and-bound and branch-and-cut [49]. These algorithms are based on linear programming, and repeatedy solve the *LP-relaxation* of the problem. The LP-relaxation is the problem one gets when one removes the requirement that solutions must be integer. These types of algorithms are used heavily in practice, and have

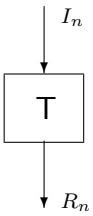

Figure 2.2: A transition function without state

shown themselves to be effective on feasible problems. For infeasible problems using unbounded integers, however, they do not always terminate.

There are other complete methods based on variable elimination, a method discovered by Fourier in 1826, This method was later rediscovered several times in the 20th century, among others by Motzkin in 1936. The method is often called Fourier-Motzkin elimination, or sometimes just Fourier elimination. Their method was created to solve the LP problem, but it can be extended to the ILP problem. One such extension is the Omega test [41].

For more information on integer programming, [49] describes branch-and-bound and branch-and-cut. Information on other methods can be found in [10, 11, 41].

2.4 Temporal induction

The verification method used in this thesis is called *temporal induction* [44, 14], which is simply induction over time, with one small addition to make it complete for finite state systems.

In a Lustre program that does not use the temporal operator **pre** or followed by ($\rightarrow$), the program is a function from its inputs to its outputs. This function is called a *transition function*. A program with a property R can be visualized as in figure 2.2. Proving the program correct is done by proving that there is no combinations of inputs which makes the property false. For programs with state (which uses the temporal operators (**pre** and $\rightarrow$) this is not possible, since there may be a combination of streams of input which makes the property false after a certain number of time points. When the temporal operators are used, we can still view the program as a function by adding extra state streams. S_{n-1} is the value of the stream in the previous time point (called the "previous" state), and S_n is the value of the streams in the current time point (the "current" state). The program is then a function from its inputs and the previous state, to the outputs and the current state as in figure 2.3.

For programs with state, it may be necessary use induction over time. In induction one tries to prove two formulas; The base case and the step case.

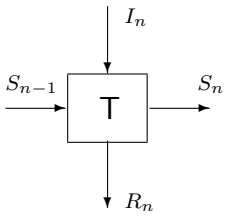

Figure 2.3: A transition function with state

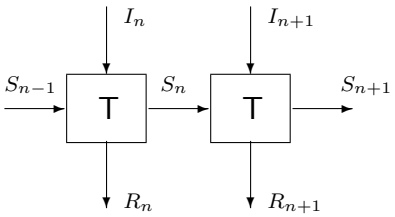

Figure 2.4: Induction step

The base case is the behaviour in the initial time point. To verify the base case, the transition function 2.3 for the program is examined to see if the property is always true regardless of the inputs. The state signals S is fixed to the initial state, given by the "followed by" operators. The property is then a function of the inputs, just as above.

In the step case the proof obligation becomes "If the property is true at time n, it will also be true at time $n + 1$". The induction step is depicted in figure 2.4. For the base case, we need to prove that the property hold in the initial time point, see figure 2.5. If both the step case and the base case can be proven, then we know that the property holds in all time points, regardless of the values on the input streams.

In some cases, it is not possible to prove the step case. This is the same problem as we have in other mathematical disciplines. Consider for instance

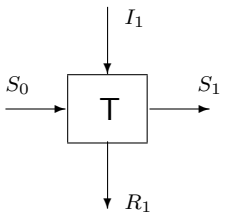

Figure 2.5: Base case

the definition of the Fibonacci sequence:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n \in [0,1] \\ \text{fib}(n-2) + \text{fib}(n-1) & \text{otherwise} \end{cases}$$

If we want to prove $\forall n.fib(n) \geq 0$ with induction, the induction step would become

$$\forall n.\text{fib}(n) \geq 0 \rightarrow \text{f}(n+1) \geq 0$$

This is impossible to prove since no assumption is made for $\text{fib}(n-1)$. This problem is normally solved with what is called *induction with depth*. For the induction step to be provable, we need to assume that both $\text{fib}(n) \geq 0$ and $\text{fib}(n+1) \geq 0$ and then try to prove $\text{fib}(n+2) \geq 0$.

$$\forall n.\text{fib}(n) \geq 0 \wedge \text{fib}(n+1) \geq 0 \rightarrow \text{f}(n+2) \geq 0$$

This can easily be proven by case splitting. The base case also has to be modified, we need to prove both $\text{fib}(0) \geq 0$ and $\text{fib}(1) \geq 0$. The *depth* of the induction proof is the number of base cases/assumptions that are used. The original, "normal" induction attempt has depth 1, and the new variant has depth 2. This can of course be generalized to induction with depth k.

In verification of Lustre programs, we have the same problem. The property that the Fibnacci sequnce is non-negative can be defined in Lustre as

```
node Fibonacci() returns ( OK : bool );
  var Fib, PFib, PPFib : int;
let
  PFib = 0 -> pre Fib;
  PPFib = 0 -> pre PFib;
  Fib = PFib + PPFib;
  OK = Fib >= 0;
tel
```

For examples such as these, we can apply the same technique. Each induction depth can be attempted in sequence until a provable depth is reached. This is not enough however. In the example below, the induction step will always fail, regardless of depth.

```
node Changer( Start, Change : bool ) returns ( Y : bool );
  var Started : bool;
let
  Started = Start -> pre Started or Start;
  Y = false -> if Change and Started then pre not Y else pre Y;
tel
```

```
node Prop( Start, Change : bool ) returns ( OK : bool);
   var HasStarted, Y : bool;
let
   HasStarted = Start → pre HasStarted or Start;
   Y = Changer( Start, Change );

   OK = not HasStarted ⇒ not Y;
tel
```

It can not be verified because there is an arbitrary long sequence of states which lead to a state where the property is false.

Variable	$t = n$	$t = n - 1$	$t = n - 2$	...	$t = n - k$
Start	false	false	false	...	false
Change	false	false	false	...	true
Started	true	true	true	...	true
Y	false	false	false	...	true
HasStarted	false	false	false	...	false
OK	true	true	true	...	false

The solution is the *unique path restriction*. We require that the sequence of states $S_n, S_{n+1}, \ldots, S_{n+k-1}$ is unique. For this example the extra assumptions would be that each state must have a unique set of values on the state variables Started, Y and HasStarted.

2.4.1 Completeness

For bounded integers, induction is complete. For unbounded integers it is not. It is easy to see why: Take a counter which enumerates the natural numbers, together with the property that -1 is not a natural number (the counter will never reach -1).

```
node Counter() returns ( R1 : bool );
   var C : int;
let
   C = 0 → pre C + 1;
   R1 = C ≠ −1;
tel
```

This property can not be verified with induction, since there is an unbounded path of valid unreachable states $\{\ldots, -4, -3, -2\}$ leading to the invalid state where the counter is -1. In this example it is possible to strengthen the property with a property saying that the counter is non-negative.

```
node Prop() returns ( R1, R2 : bool );
    C : int;
let
    C = Counter();
    R1 = C ≥ 0;
    R2 = C ≠ −1;
tel
```

Having two properties like this is the same as the conjunction of the properties. The new property is provable, and therefore the conjunction of properties is provable.. Since the conjunction is valid, then the original property must be valid also, since $\forall x.x \geq 0 \rightarrow x \neq -1$.

Incompleteness of induction is not such bad news as it may seem. In fact, Lustre with unbounded integers is Turing-complete, and thus no method exists of automatically strengthening properties to make induction complete. No other verification technique is complete either, of course.

2.5 Turing-completeness of Lustre

The normal way of proving that a language is Turing-complete is to show that it is possible to implement a Turing machine in the language. A Turing machine has a tape machine with the operations in table 2.1. The operations that can be performed are; Writing a bit (0 or 1) to the position of the tape under the tape head and moving the tape to the left or the right. The tape machine is controlled by a finite state machine (FSM), which given the bit under the tape head and its current state, sends an intruction to the tape machine and changes to a new state.

It is possible to implement a Turing machine in the small fragment of Lustre that only has linear integer expressions. That is, multiplication, division and modulo operators are only allowed with a constant right operand, while addition and subtraction are allowed with arbitrary operands. This is the fragment that is supported by the method discussed in the thesis. The

Operation	Op code
Do nothing	0
Write 1	1
Write 0	2
Move to right	3
Move to left	4

Table 2.1: Opcodes for tape

operations are encoded using an integer, as indicated in table 2.1. The tape itself is encoded in two integers Left and Right. These are seen as infinite

streams of bits, Left being the bits in the left half of the tape and Right the bits in the right half. The bit under the tape head is the least significant bit in Right. Moving the tape one step to the right is equivalent to dividing Left by two and multiplying Right by two. The least significant bit in Left before the operation is added to Right. Moving to the right then becomes

```
PLeft = Init_Left → pre Left;
PRight = Init_Right → pre Right;
Left = PLeft / 2;
Right = PRight * 2 + PLeft mod 2;
```

where PLeft and PRight was the previous value of the tape. Moving to the left is analogous. Reading and writing from/to the tape is simply a matter of manipulating the least significant bit in Right.

The tape machine then becomes

```
node Tape_Machine( Init_Left, Init_Right : int; Operation : int )
returns ( Value : bool; Left, Right : int );
    var PLeft, PRight : int;
let

   PLeft = Init_Left → pre Left;
   PRight = Init_Right → pre Right;

   Left = if Operation = 0 then PLeft
          else if Operation = 1 then PLeft
          else if Operation = 2 then PLeft
          else if Operation = 3 then PLeft / 2;
          else PLeft * 2 + PRight mod 2;

   Right = if Operation = 0 then PRight
           else if Operation = 1 then (PRight / 2) * 2 + 1
           else if Operation = 2 then (PRight / 2) * 2
           else if Operation = 3 then PRight * 2 + PLeft mod 2
           else PRight / 2;

   Value = Right mod 2 = 1
tel
```

The finite state machine has the states $s_0, s_1, \ldots, s_k$, which can be modeled with an integer variable. The initial state s_0 is 0, and the other k states are the k first positive numbers. The machine is a function taking the state and the value under the tape head and returns a new state and an opera-

tion for the tape machine. Since the state machine is finite, it is possible to implement this with a finite number of nestled if-then-else expressions.

```
node FSM( Value : bool ) return ( Operation : int );
   var PState : int;
       State : int;
let
   PState = 0 → pre State;
   State = if PState = 0 and Value then <new state>
           else if PState = 0 and not Value then <new state>
           else if PState = 1 and Value then <new state>
           else if PState = 1 and not Value then <new state>
           ...;

   Operation = if State = 0 then <operation>
               else if State = 1 then <operation>
               else if State = 2 then <operation>
               ...;
tel
```

The Turing machine is the tape machine and the finite state machine connected together.

```
node Turing_Machine( Init_Left, Init_Right : int )
returns ( Left, Right : int );
   var Operation : int;
       Value : bool;
let
    ( Value, Left, Right ) = Tape_Machine( Init_Left, Init_Right,
                                           0 → pre Operation );
    Operation = FSM( Value );
tel
```

Chapter 3

Formal language

For formal verification we need a formal language, i.e. a language with a well-defined meaning. The fragment of Lustre that we are interested in has both boolean and integer variables. Therefore, propositional logic extended with integer arithmetic would be a natural choice. This language is not one of the "standard" languages in logic, so this chapter will give a definition of the language and its semantics. Those readers not interested in the details of the language may safely skip this chapter.

The language definition and semantics is inspired by [42].

3.1 Language definition

Propositional logic uses a set of propositional variables $\mathcal{V}_P$ and the connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence). The language of propositional logic can be defined in the usual way:

1. A variable $p \in \mathcal{V}_P$ is a $\mathcal{L}$-formula
2. If α is a $\mathcal{L}$-formula, then $\neg\alpha$ is a $\mathcal{L}$-formula
3. if α and β is are $\mathcal{L}$-formulas, then $\alpha \wedge \beta$ is a $\mathcal{L}$-formula
4. if α and β is are $\mathcal{L}$-formulas, then $\alpha \vee \beta$ is a $\mathcal{L}$-formula
5. if α and β is are $\mathcal{L}$-formulas, then $\alpha \rightarrow \beta$ is a $\mathcal{L}$-formula
6. if α and β is are $\mathcal{L}$-formulas, then $\alpha \leftrightarrow \beta$ is a $\mathcal{L}$-formula

Examples of $\mathcal{L}$-formulas are for instance

$$(p \rightarrow q) \wedge (q \rightarrow p)$$
$$(\neg p \rightarrow q) \vee \neg(q \leftrightarrow z)$$

We can now extend this language with by adding relations between integer expressions. First, we start by defining linear expressions. $\mathbb{Z}$ is the set of integers, and $\mathcal{V}_\mathbb{Z}$ is the set of integer variables.

1. A constant $c \in \mathbb{Z}$ is a $\mathcal{L}$-term
2. A variable $v \in \mathcal{V}_\mathbb{Z}$ is a $\mathcal{L}$-term
3. If $c \in \mathbb{Z}$ and $v \in \mathcal{V}_\mathbb{Z}$, then cv is a $\mathcal{L}$-term
4. If l_1 and l_2 are $\mathcal{L}$-terms, then $l_1 + l_2$ is a $\mathcal{L}$-term
5. If l_1 and l_2 are $\mathcal{L}$-terms, then $l_1 - l_2$ is a $\mathcal{L}$-term

This means that if $x, y \in \mathcal{V}_\mathbb{Z}$, then 42, $2x + 3$ and $7x - 2y$ are all $\mathcal{L}$-terms, but xy, $x/2$ and $2/x$ are not. We can now define linear relations, where all such relations are $\mathcal{L}$-formulas.

1. If l_1 and l_2 are $\mathcal{L}$-terms and $\bowtie$ is one of $=, \neq, <, \leq, >, \geq$, then $l_1 \bowtie l_2$ is a $\mathcal{L}$-formula.

This means that relations beteen two linear expression is a $\mathcal{L}$-formula. For example,

$$\begin{aligned} 7 &= 3 \\ 1 &> 0 \\ x + 2y &\leq 2x + 2 \\ (x > y) \wedge (y > z) &\rightarrow (x > z) \\ p \wedge q &\rightarrow \neg(x > 2y - 1) \\ (x + 1 = 2) &\leftrightarrow (1 < 0) \end{aligned}$$

are all $\mathcal{L}$-formulas.

3.2 Semantics

It is not enough to have a language, we also need to define the meaning of formulas. In our case we are interested in knowing if a variable assignment of the propositional and integer variables makes the formula true or not. This is done formally by defininig an *interpretation*.

An interpretation $\mathbb{I}$ is a map from propositional and integer variables to truth values and integer values, respectively. Formally $\mathbb{I} = \langle \mathbb{I}_p, \mathbb{I}_\mathbb{Z} \rangle$, where $\mathbb{I}_p : \mathcal{V}_P \mapsto \{\bot, \top\}$, $\mathbb{I}_\mathbb{Z} : \mathcal{V}_\mathbb{Z} \mapsto \mathbb{Z}$. This interpretation can be extended over integer expressions in this way:

- $\hat{\mathbb{I}}_\mathbb{Z}(c) = c$ iff $c \in \mathbb{Z}$
- $\hat{\mathbb{I}}_\mathbb{Z}(v) = \mathbb{I}_\mathbb{Z}(v)$ for every $v \in \mathcal{V}_\mathbb{Z}$

- $\hat{\mathbb{I}}_{\mathbb{Z}}(a+b) = \hat{\mathbb{I}}_{\mathbb{Z}}(a) + \hat{\mathbb{I}}_{\mathbb{Z}}(b)$
- $\hat{\mathbb{I}}_{\mathbb{Z}}(a-b) = \hat{\mathbb{I}}_{\mathbb{Z}}(a) - \hat{\mathbb{I}}_{\mathbb{Z}}(b)$
- $\hat{\mathbb{I}}_{\mathbb{Z}}(ca) = c \cdot \hat{\mathbb{I}}_{\mathbb{Z}}(b)$ for every $c \in \mathbb{Z}$

We can now define satisfiability for $\mathcal{L}$-formulas. That a formula φ is satisfiable with an interpretation $\mathbb{I} = \langle \mathbb{I}_p, \mathbb{I}_{\mathbb{Z}} \rangle$ is written $\mathbb{I} \models \varphi$, and is defined in this way:

- $\mathbb{I} \models v$ iff $v \in \mathcal{V}_P$ and $\mathbb{I}_p(v) = \top$
- $\mathbb{I} \models \neg\varphi$ iff $\mathbb{I} \not\models \varphi$
- $\mathbb{I} \models \varphi \wedge \psi$ iff $\mathbb{I} \models \varphi$ and $\mathbb{I} \models \psi$
- $\mathbb{I} \models \varphi \vee \psi$ iff $\mathbb{I} \models \varphi$ or $\mathbb{I} \models \psi$
- $\mathbb{I} \models \varphi \rightarrow \psi$ iff $\mathbb{I} \not\models \varphi$ or $\mathbb{I} \models \psi$
- $\mathbb{I} \models \varphi \leftrightarrow \psi$ iff either $\mathbb{I} \models \varphi$ and $\mathbb{I} \models \psi$, or $\mathbb{I} \not\models \varphi$ and $\mathbb{I} \not\models \psi$
- $\mathbb{I} \models \alpha \bowtie \beta$, where $\bowtie$ is one of $=, \neq, <, \leq, >, \geq$ iff $\hat{\mathbb{I}}_{\mathbb{Z}}(\alpha) \bowtie \hat{\mathbb{I}}_{\mathbb{Z}}(\beta)$ where $\bowtie$ is the semantic function of $\bowtie$.

Lets look at an example formula:

$$(p \rightarrow (x + 1 < y)) \wedge (q \rightarrow (y = 0))$$

This formula can be satisfied with the interpretation

$$\mathbb{I} = \langle \{p \mapsto \bot, q \mapsto \top, \}, \{x \mapsto 1, y \mapsto 0\} \rangle$$

If the interpretation is changed to make p true, then the formula would not be satisfied.

We now have a language where we can describe relations over integer expressions, and propositional logic. The language has a precise meaning of what it means for a formula to be satisfiable. What we need is a decision procedure which can given an arbitrary formula in our language decide whether of not the formula is satisfiable. If it is we also need the procedure to generate an interpretation (a variable assignment) which satisfies the formula. Deciding satisfiability for these $\mathcal{L}$-formulas is an NP-complete problem [42], so it is possible to create such procedures. In chapters 5 we will give two such algorithms.

3.3 Restrictions

We do not need general relations over integers, so we limit ourselves to two restricted variants of the language. A *constraint* is a relation on the form

$$\sum_i a_i x_i \bowtie b$$

where a_i, b are constants and x_i are variables. Constraints arre often written on the form

$$\mathbf{a}^t\mathbf{x} \bowtie b$$

taken from linear algebra, where $\mathbf{a}^t$ is a transposed vector of constants, $\mathbf{x}$ a vector of variables and b a constant.

3.3.1 Standard constraints

A *standard* constraint is a constraint of the form $\mathbf{a}^t\mathbf{x} \leq b$. A formula with general constraints can easily be rewritten into a equivalent formula with only standard constraints. First all variables are moved to the left hand side and all constants to the right hand side. After simplification, they are rewritten in this way:

Original	Replacement
$\mathbf{a}^t\mathbf{x} < b$	$\mathbf{a}^t\mathbf{x} \leq b - 1$
$\mathbf{a}^t\mathbf{x} \geq b$	$-\mathbf{a}^t\mathbf{x} \leq -b$
$\mathbf{a}^t\mathbf{x} > b$	$-\mathbf{a}^t\mathbf{x} \leq -b - 1$
$\mathbf{a}^t\mathbf{x} = b$	$\mathbf{a}^t\mathbf{x} \leq b \wedge -\mathbf{a}^t\mathbf{x} \leq -b$
$\mathbf{a}^t\mathbf{x} \neq b$	$\mathbf{a}^t\mathbf{x} \leq b - 1 \vee -\mathbf{a}^t\mathbf{x} \leq -b - 1$

3.3.2 Guarded constraints

Constraints that are guarded by a propositional variable, $p \rightarrow$ constraint are called *guarded constraints*, and the literal is called a *guard*. The constraints that are allowed are on the form $\mathbf{a}^t\mathbf{x} \leq b$ or $\mathbf{a}^t\mathbf{x} = b$. The advantage of this is that in interpretation where the guard is false, the truth value of the contraint does not matter for the truth value of the formula. This means that it is possible to have equality constraints, since we never have to negate them.

Chapter 4

Translation from Lustre to $\mathcal{L}$-formulas

The verification idea is to take programs and properties written in Lustre, and use induction to verify those. Each base or step case should be translated to a $\mathcal{L}$-formula which can be analyzed with the procedures described in chapter 5. In this chapter three differrent methods of translating Lustre nodes to transition functions in our logical language is presented. For the casual reader, it is recommended to only read the first paragraph of each section below, which gives an informal description of the methods.

For each Lustre variable v, we create one integer variable per time point $\{v_i | i \in \mathbb{N}\}$, where v_0 is the variable in the initial time point, v_1 the next time point, and so on. A transition function is created, or *instantiated*, at time point n with the function inst_n which takes a Lustre expression and returns the value of the expression. It also creates a number of constraints, which define the returned expression.

4.1 Per operator instantiation

In the "per operator" translation scheme, each operator is tranlated separately into standard constraints. For each operator, two cosntraints will be created which defines the value of the expression. A stream definition for an integer stream is defined as a set of two constraints $\mathbf{a}^t\mathbf{x} \leq b$, $-\mathbf{a}^t\mathbf{x} \leq -b$ for each operator used in the expression. For the Lustre definition

$$\mathsf{X} = \mathsf{Y} + 1;$$

two constraints are needed to describe this using standard constraints:

$$\begin{aligned} X_n + Y_n &\leq 1 \\ -X_n - Y_n &\leq -1 \end{aligned}$$

X_n is the value of the stream X at time point n, and Y_n is the value of the stream Y at time point n. For larger expressions, it is necessary to introduce extra variables for the intermediary expressions.

4.1.1 Formalities

Instantiation using the function inst of a arithmetic integer expression is performed in this way: The function returns a value for a Lustre expression, and also creates a number of constraints.

$$
\begin{array}{lcll}
\mathsf{inst}_n(c) & = & c & \text{if } c \in \mathbb{Z} \\
\mathsf{inst}_n(v) & = & v_n & \text{if } v \in \mathcal{V}_{\mathbb{Z}} \\
\mathsf{inst}_n(\alpha + \beta) & = & z & \textbf{where} \quad -x - y + z \leq 0 \\
 & & & \qquad\quad x + y - z \leq 0 \\
 & & & \qquad\quad x = \mathsf{inst}_n(\alpha) \\
 & & & \qquad\quad y = \mathsf{inst}_n(\beta) \\
 & & & \qquad\quad z \text{ is a fresh variable} \\
\mathsf{inst}_n(\alpha - \beta) & = & z & \textbf{where} \quad -x + y + z \leq 0 \\
 & & & \qquad\quad x - y - z \leq 0 \\
 & & & \qquad\quad x = \mathsf{inst}_n(\alpha) \\
 & & & \qquad\quad y = \mathsf{inst}_n(\beta) \\
 & & & \qquad\quad z \text{ is a fresh variable} \\
\mathsf{inst}_n(\alpha * \beta) & = & z & \text{if } c \in \mathbb{Z} \\
 & & & \textbf{where} \quad -cx + z \leq 0 \\
 & & & \qquad\quad cx - z \leq 0 \\
 & & & \qquad\quad x = \mathsf{inst}_n(\alpha) \\
 & & & \qquad\quad c = \mathsf{inst}_n(\beta) \\
 & & & \qquad\quad z \text{ is a fresh variable} \\
\mathsf{inst}_n(\alpha / \beta) & = & z & \text{if } c \in \mathbb{Z} \\
 & & & \textbf{where} \quad x + cz \leq 0 \\
 & & & \qquad\quad x - cz \leq c - 1 \\
 & & & \qquad\quad x = \mathsf{inst}_n(\alpha) \\
 & & & \qquad\quad c = \mathsf{inst}_n(\beta) \\
 & & & \qquad\quad z \text{ is a fresh variable} \\
\mathsf{inst}_n(\alpha \mod \beta) & = & z & \text{if } c \in \mathbb{Z} \\
 & & & \textbf{where} \quad x - c\mathsf{inst}_n(\alpha/\beta) - z \leq 0 \\
 & & & \qquad\quad -x + c\mathsf{inst}_n(\alpha/\beta) + z \leq 0 \\
 & & & \qquad\quad x = \mathsf{inst}_n(\alpha) \\
 & & & \qquad\quad c = \mathsf{inst}_n(\beta) \\
 & & & \qquad\quad z \text{ is a fresh variable}
\end{array}
$$

For **if**−**then**−**else**, the instantiation is defined as

$$\begin{array}{lcll} \mathsf{inst}_n(\texttt{if } \alpha \texttt{ then } \beta \texttt{ else } \gamma) & = & z & \\ & & \textbf{where} & c \rightarrow z = z_1 \\ & & & \neg c \rightarrow z = z_2 \\ & & & c = \mathsf{inst}_n(\alpha) \\ & & & z_1 = \mathsf{inst}_n(\beta) \\ & & & z_2 = \mathsf{inst}_n(\beta) \\ & & & z \text{ is a fresh variable} \end{array}$$

For the followed by operator $\rightarrow$, it is simply a matter of checking if we are in the initial time point, and for the **pre** operator, we instantiate the operand on the previous time point.

$$\begin{array}{lcl} \mathsf{inst}_n(\alpha\texttt{->}\beta) & = & \mathsf{inst}_0(\alpha) \text{ if } n = 0 \\ & & \mathsf{inst}_n(\beta) \text{ otherwise} \\ \mathsf{inst}_n(\texttt{pre } \alpha) & = & \mathsf{inst}_{n-1}(\alpha) \end{array}$$

Applying **pre** in the initial time point is undefined, since there is no previous time point.

Comparison operators are translated in a straightforward way, here exemplified with the less-than operator:

$$\mathsf{inst}_n(\alpha < \beta) \quad = \quad x - y \leq -1 \quad \textbf{where } x = \mathsf{inst}_n(\alpha), \; y = \mathsf{inst}_n(\beta)$$

4.1.2 An example

To illustrate how the translation works, here is a small example. The program is a single Lustre node with a counter, counting from 0 and upwards together with the property that the counter is non-negative.

```
node Counter() returns ( OK : bool );
    var C : int;
let
    C = 0 → pre C + 1;
    OK = C ≥ 0;
tel
```

For the initial time point, this would be translated to a $\mathcal{L}$-formula in CNF as

$$\begin{array}{l} \{\ C_1 \leq 0\ \} \\ \{\ -C_1 \leq 0\ \} \\ \{\ \neg \mathrm{OK}_1,\ (-C_1 \leq 0)\ \} \\ \{\ \mathrm{OK}_n,\ (C_n \leq -1)\ \} \end{array}$$

The first two clauses define C as 0 in the first time point. The last two correspong to the property. If OK is true, then C must be greater than or equal to 0. Otherwise C must be a negative number. For all other time points, the translation becomes

$$\begin{array}{l}\{\ C_n - C_{n-1} \leq 1\ \} \\ \{\ -C_n + C_{n-1} \leq -1\ \} \\ \{\ \neg \mathrm{OK}_n,\ (-C_n \leq 0)\ \} \\ \{\ \mathrm{OK}_n,\ (C_n \leq -1)\ \}\end{array}$$

4.2 Maximal instantiation

If large expressions are used in the program, the number of extra variables may become large. In "Maximal" instantiation, the unnecessary variables are eliminated. In the example

$$\mathsf{X} = \mathsf{Y} + \mathsf{Z} + 1;$$

The constraints that describe this definition becomes

$$\begin{array}{rl} X_n + Y_n + Z_n \leq & 1 \\ -X_n - Y_n - Z_n \leq & -1 \end{array}$$

In this case one variable and two constraints could be saved in this way.

4.2.1 Formalities

$$\begin{array}{lcl} \mathsf{inst}_n(\alpha + \beta) & = & \mathsf{inst}_n(\alpha) + \mathsf{inst}_n(\beta) \\ \mathsf{inst}_n(\alpha - \beta) & = & \mathsf{inst}_n(\alpha) - \mathsf{inst}_n(\beta) \\ \mathsf{inst}_n(\alpha * \beta) & = & \mathsf{inst}_n(\alpha) \cdot \mathsf{inst}_n(\beta),\ \text{if } \mathsf{inst}_n(\alpha) \in \mathbb{Z} \text{ or } \mathsf{inst}_n(\alpha) \in \mathbb{Z} \end{array}$$

The other cases are identical to "per operator" instantiation.

4.3 Guarded maximal instantiation

In the "Guarded maximal" method, only guarded constraints are used. With guarded contraints, two types of constraints are allowed in the formula $\mathbf{a}^t\mathbf{x} \leq b$ and $\mathbf{a}^t\mathbf{x} = b$. This means that definitions can use equality constraints instead of two inequality constraints, which leads to a "simpler" formula. Another advantage is that integer **if-then-else** expressions can be written in a more efficient way.

4.3.1 Formalities

The instantiation function inst returns a tuple $\langle e, x \rangle$ where x is the expression and e is an *enabler*. When the enabler is false, the constraints that

are associated with then expression x have no effect on satisfiability of the formula.

$$
\begin{array}{lll}
\mathsf{inst}_n(c) & = \langle e, c\rangle & \text{if } c \in \mathbb{Z} \\
\mathsf{inst}_n(v) & = \langle e, v_n\rangle & \text{if } v \in \mathcal{V}_{\mathbb{Z}} \\
\mathsf{inst}_n(\alpha + \beta) & = \langle e, x + y\rangle & \textbf{where} \quad \langle a, x\rangle = \mathsf{inst}_n(\alpha) \\
 & & \qquad\quad \langle b, y\rangle = \mathsf{inst}_n(\beta) \\
 & & \qquad\quad e \leftrightarrow a \\
 & & \qquad\quad e \leftrightarrow b \\
 & & \qquad\quad e \text{ is a fresh variable} \\
\mathsf{inst}_n(\alpha - \beta) & = \langle e, x - y\rangle & \textbf{where} \quad \langle a, x\rangle = \mathsf{inst}_n(\alpha) \\
 & & \qquad\quad \langle b, y\rangle = \mathsf{inst}_n(\beta) \\
 & & \qquad\quad e \leftrightarrow a \\
 & & \qquad\quad e \leftrightarrow b \\
 & & \qquad\quad e \text{ is a fresh variable} \\
\mathsf{inst}_n(\alpha * \beta) & = \langle e, x \cdot y\rangle & \text{if } x \in \mathbb{Z} \text{ or } y \in \mathbb{Z} \\
 & & \textbf{where} \quad \langle a, x\rangle = \mathsf{inst}_n(\alpha) \\
 & & \qquad\quad \langle b, y\rangle = \mathsf{inst}_n(\beta) \\
 & & \qquad\quad e \leftrightarrow a \\
 & & \qquad\quad e \leftrightarrow b \\
 & & \qquad\quad e \text{ is a fresh variable}
\end{array}
$$

For **if**−**then**−**else**, the instantiation becomes

$$
\begin{array}{lll}
\mathsf{inst}_n(\texttt{if } \alpha \texttt{ then } \beta \texttt{ else } \gamma) & = & \langle e, z\rangle \\
 & & \textbf{where} \quad \langle a, b\rangle = \mathsf{inst}_n(\alpha) \\
 & & \qquad\quad \langle c, x\rangle = \mathsf{inst}_n(\beta) \\
 & & \qquad\quad \langle d, y\rangle = \mathsf{inst}_n(\gamma) \\
 & & \qquad\quad e \leftrightarrow a \\
 & & \qquad\quad e \wedge a \leftrightarrow c \\
 & & \qquad\quad e \wedge \neg a \leftrightarrow c \\
 & & \qquad\quad c \rightarrow x = z \\
 & & \qquad\quad d \rightarrow y = z \\
 & & \qquad\quad e \text{ is a fresh variable}
\end{array}
$$

We enable the condition only if the if-then-else expression is enabled, and the "then"-expression is enabled only if the if-then-else expression is enabled and the condition is true. The "else"-expression in enabled only if the if-then-else expression is enabled and the condition is false.

Comparison operators are translated in this way, as exemplified by the

less-than ($<$) operator:

$$\begin{array}{lcll} \mathsf{inst}_n(\alpha < \beta) & = & \langle e, p \rangle & \textbf{where } \langle a, x \rangle = \mathsf{inst}_n(\alpha) \\ & & & \langle b, y \rangle = \mathsf{inst}_n(\beta) \\ & & & e \leftrightarrow a \\ & & & e \leftrightarrow b \\ & & & e \wedge p \rightarrow x - y \leq -1 \\ & & & e \wedge \neg p \rightarrow -x - y \leq 0 \\ & & & e, p \text{ are fresh variables} \end{array}$$

e is the enabler for the comparison, and p is a propositional variable which represents the truth value of the comparison.

4.3.2 An example

```
node Counter() returns ( OK : bool );
    var C : int;
let
    C = 0 → pre C + 1;
    OK = C ≥ 0;
tel
```

For the initial time point, this would be translated to a $\mathcal{L}$-formula in CNF as (simplified)

$$\begin{array}{l} \{\ \neg a,\ (C_1 = 0)\ \} \\ \{\ a\ \} \\ \{\ \neg b,\ (-C_n \leq 0)\ \} \\ \{\ \neg c,\ (C_n \leq -1)\ \} \\ \{\ d,\ \neg b\ \} \\ \{\ d,\ \neg c\ \} \\ \{\ \neg b,\ \neg c\ \} \\ \{\ \neg d,\ b,\ c\ \} \\ \{\ d\ \} \end{array}$$

The two first clauses say that C is 0. The next two are the constraints for the property, guarded by propositional variables b and c. Exactly one of them should be true, which is described in the last clauses.

Chapter 5

The decision procedure for $\mathcal{L}$-formulas

A simple decision procedure is based on the following observation. Treating the problem as a purely propositional problem, iff there is any propositional model for which the constraints are consistent, then the original problem is satisfiable. *Offline integration* uses a SAT solver to generate propositional models, and an integer programming package to check these models.

5.1 Offline integration

The original formula φ is translated into a purely propositional formula φ_p by replacing each integer constraint with a fresh propositional variable, called an *in-place* variable. Thus, in the example

$$\begin{array}{l}\{x \leq 0, y \leq 0\} \\ \{-y \leq -1\}\end{array}$$

if we replace $x \leq 0$ with p, $y \leq 0$ with q and $-y \leq -1$ with r, we get the propositional formula

$$\begin{array}{l}\{p, q\} \\ \{r\}\end{array}$$

This formula is submitted to a SAT solver, called Psat in the algorithm. In this example, the formula is satisfiable, for example with the model $\mathbb{M}_p = \{p, q, r\}$. From this model a constraint problem is created with the function generate by checking the valuation of the in-place variables.

$$C = \{x \leq 0, y \leq 0, -y \leq -1\}$$

Feasibility of the resulting constraint problem is checked in Csat. If the constraint problem is feasible, the solution can be merged with the SAT model to form a model for the original formula.

If the constraint problem is not feasible, like in the example above, then an explanation is generated by the function explain. It generates a propositional formula with the in-place variables explaining why the constraint problem was infeasible. This formula is added to the original propositional formula, and the procedure starts over. If the explain function returns the exmplanation p, q, r, then the cause $\{\neg p, \neg q, \neg r\}$ is added to the formula.

$$\{p, q\}$$
$$\{r\}$$
$$\{\neg p, \neg q, \neg r\}$$

The new clause prevents the SAT solver from finding the same model again.

This is repeated until either a feasible constraint problem is found, or the propositional formula becomes unsatisfiable.

Algorithm 5.1 Offline integration

Require: φ is a standard or guarded constraint formula
  loop
    $\mathbb{I}_p \leftarrow \mathsf{Psat}(\varphi)$
    if $\mathbb{I}_p = \emptyset$ **then**
      return *unsatisfiable*
    else
      if $\mathsf{Csat}(\mathsf{generate}(\varphi, \mathbb{I}_p))$ **then**
        return *satisfiable*
      else
        $\varphi \leftarrow \varphi \wedge \neg\mathsf{explain}(\mathsf{generate}(\varphi, \mathbb{I}_p))$
      end if
    end if
  end loop

5.1.1 Standard constraints

For standard constraints, the translation to a propositional formula is done by creating a fresh propositional variable for each constraints, and keeping track of which in-place variable corresponds to which constraint. Constructing the constraint problem is done by checking the valuation of each in-place variable. If it is true, the corresponding constraint is added to the constraint problem. If it is false, the negation of the corresponding constraint is added. Since all standard constraints are on the form $\mathbf{a}^t\mathbf{x} \leq b$, negating a constraint is simple:

$$\neg(\mathbf{a}^t\mathbf{x} \leq b) \leftrightarrow \mathbf{a}^t\mathbf{x} > b \leftrightarrow -\mathbf{a}^t\mathbf{x} < -b \leftrightarrow -\mathbf{a}^t\mathbf{x} \leq -b - 1$$

Algorithm 5.2 generate$(\varphi, \mathbb{I}_p)$ (Standard variety)

$C \leftarrow$ all constraints in φ
$A \leftarrow \emptyset$
for all $c \in C$ **do**
  $p \leftarrow$ in-place variable for c
  if $\mathbb{I}_p(p) = \top$ **then**
    $A \leftarrow A \cup \{c\}$
  else
    $A \leftarrow A \cup \{\neg c\}$
  end if
end for
return A

5.1.2 Guarded constraints

When using guarded constraints, all constraints are guarded by a propositional variable $p \rightarrow c$. These constraints are all on the top elevel, meaning that the guards and constraints only occurs in clauses like this:

$$\{\ \neg p,\ c\ \}$$

The formula is translated to propositional form simply by removing these guarded constraints. This is done by simply removing these clauses. Constraint problems are constructed by checking the valuation of each guard in the original formula. A constraint is added to the constraint problem iff the corresponding guard is true.

Algorithm 5.3 generate$(\varphi, \mathbb{I}_p)$ (Guarded variety)

$C \leftarrow$ all constraints in φ
$A \leftarrow \emptyset$
for all $c \in C$ **do**
  $g \leftarrow$ guard for c
  if $\mathbb{I}_p(g) = \top$ **then**
    $A \leftarrow A \cup \{c\}$
  end if
end for
return A

5.1.3 Soundness and completeness

The decision procedure described here is sound as long as the integer programming procedure Csat is sound, and the explanations all correspond to infeasible systems.

It is also complete, given a complete integer programming procedure [42]. Procedures based on branch-and-bound or branch-and-cut are not complete, and if one of those is used, then the abose algorithm will also be incomplete. There are only a finite number of SAT models to the propositional formula, and for each spurious model we add an explanation preventing the SAT solver from generating the same model again. The SAT solver will eventually either find a correct model, or the formula will become unsatisfiable.

5.2 Creating explanations of infeasible systems

There are many possibilities for an explain function. The most naïve version is to simply create a conjunction of all in-place literals, whose constraints have been asserted in the current SAT model. In this example

$$\begin{array}{ll} (1) & A - B \leq \ \ 1 \\ (2) & -A + B \leq -1 \\ (3) & -A \leq \ \ 0 \\ (4) & A \leq -1 \end{array}$$

we can see that the constraints 3 and 4 contraidicts each other, and the in-place variable for those constraints would be a better explanation than the in-place variables for all four constraints.

If reasons such as this can be used instread of using all in-place variables as explanations, the procedure may be more effective. With shorter explanations, we prune more of the model space of the SAT formula, and fewer spurious models will be generated. In integer programming, there are methods of finding these reasons. In chapter 6 these are described.

5.3 A small example

We are now ready to look at a complete example. We take a counter that counts from 0 upwards, and try to prove that the counter will always be a natural number.

```
node Counter() returns ( OK : bool );
    var C : int;
let
    C = 0 → pre C + 1;
    OK = C ≥ 0;
tel
```

With the "per operator" translation, we would get in step 0

$$\begin{array}{l}\{\ C_n - C_{n-1} \leq 1\ \} \\ \{\ -C_n + C_{n-1} \leq -1\ \} \\ \{\ \neg\mathrm{OK}_n,\ (-C_n \leq 0)\ \} \\ \{\ \mathrm{OK}_n,\ (C_n \leq -1)\ \}\end{array}$$

This is converted to a purely propositional problem by replacing the constraints with in-place variables $\{p_1 \mapsto C_n - C_{n-1} \leq 1, p_2 \mapsto -C_n + C_{n-1} \leq -1, p_3 \mapsto -C_n \leq 0, p_4 \mapsto C_n \leq -1\}$.

$$\begin{array}{l}\{\ p_1\ \} \\ \{\ p_2\ \} \\ \{\ \neg\mathrm{OK}_n,\ p_3\ \} \\ \{\ \mathrm{OK}_n,\ p_4\ \}\end{array}$$

The property corresponds to the variable OK_n, and to prove it we assume $\neg$OK and hope it will lead to an unsatisfiable formula. Psat has many models to choose from, lets say it first identifies $\mathbb{I}_p = \{p_1, p_2, p_3, p_4, \mathrm{OK}_n\}$. Since we are using standard constraints, the constraint problem generated from this model is

$$\begin{array}{lrcr}(1) & C_n - C_{n-1} & \leq & 1 \\ (2) & -C_n + C_{n-1} & \leq & -1 \\ (3) & -C_n & \leq & 0 \\ (4) & C_n & \leq & -1\end{array}$$

This is infeasible, with the IIS $\{3, 4\}$. The explanation then becomes $p_3 \wedge p_4$, and the clause $\{\neg p_3, \neg p_4\}$ is added to the formula. There is still one model left, $\mathbb{I}_p = \{p_1, p_2, \neg p_3, p_4\}$. The constraint problem now becomes

$$\begin{array}{lrcr}(1) & C_n - C_{n-1} & \leq & 1 \\ (2) & -C_n + C_{n-1} & \leq & -1 \\ (3) & C_n & \leq & -1 \\ (4) & C_n & \leq & -1\end{array}$$

Which is feasible, for instance with the solution $C_n = -1, C_{n-1} = -2$, and so the original formula is satisfiable with the model

$$\mathbb{M} = \langle\{\neg\mathrm{OK}_n\}, \{C_n = -1, C_{n-1} = -2\}\rangle$$

We now know that the property is not true in all states, so we move on to induction with depth 1. The base case becomes

$$\begin{array}{l}\{C_0 \leq 0\} \\ \{-C_0 \leq 0\} \\ \{\neg\mathrm{OK}_n, -C_0 \leq 0\} \\ \{\mathrm{OK}_n, C_0 \leq -1\}\end{array}$$

Which is unsatisfiable. For the step case the formula is decided in the same way, and it will also be unsatisfiable. Since both the base case and the step case is unsatisfiable, we can prove the property with depth 1.

5.4 Preprocessing constraints

It is possible to analyze the set of constraints in a formula before starting the decision procedure. First identical constraints can be identified, and the same in-place variable used for all identical instances of constraints [42].

This works for both standard and guarded constraint formulas, but for guarded constraints we need to introduce an extra variable. For instance in the formula

$$\{\neg g_1, x = 0\}$$
$$\{\neg g_2, x = 0\}$$
$$\{\neg g_1, \neg g_2\}$$
$$\{\neg g_2 \neg g_1\}$$

It is not possible to just replace g_2 with g_1, since that would make the formula unsatisfiable. Instead, we can add a fresh propositional variable g, and change the formula to

$$\{\neg g, x = 0\}$$
$$\{\neg g_1, \neg g_2\}$$
$$\{\neg g_2, \neg g_1\}$$
$$\{\neg g_1, g\}$$
$$\{\neg g_2, g\}$$

For standard constraints we simply create the same in-place variable for all occurencies of a particular constraint.

Secondly, it is possible to try to discover infeasible subsets of the constraints, and add explanations for these to the propositional formula [15]. In this way, the number of spurious SAT models can be reduced. Normally, all infeasible subsets of a certain maximal cardinality (typically 2) is discovered. This has been reported to be successful on random formulas [6].

5.5 Online integration

In online integration, we check *partial* interpretations instead of only complete models. Checking a partial interpretation is done in the same way as in offline integration, except that we do not consider those constraints whose in-place variables are not defined in the interpretation.

These checks are performed inside the SAT solver. Everytime the solver assigns a value to an in-place variable, the corresponding constraint is asserted. If the set of asserted constraints is infeasible, we have a conflict, which can be handled in the normal way in the solver. If a complete SAT model can be found with a feasible set of asserted constraints, we know the original formula was satisfiable. The SAT model together with the solution to the set of asserted constraints then form a model for the original formula.

5.5.1 Basic algorithm

The algorithm used for online integrations is basically the generate function described in 5.2 and 5.3. The DPLL based SAT solver is modified such that it keeps track of a set of constraints that are asserted. Initially that set is empty, and every time an in-place variable is assigned a value, the generate function is used to check if the corresponding constraint has become asserted. If it has the constraint is added to the set of asserted constraints, and if the set has become infeasible a conflict has been discovered. The reason for the conflict can be discovered using the methods in chapter 6 in the same way as in offline integration, and that reason is communicated to the SAT solver which generates a conflict clause and backtracks.

5.5.2 Making inferences

In the algorithm described here, no information is derived from partial models. In [28], the constraints that have been generated in the partial model is used to infer other constraints. With the constraints

$$X{=}Y$$
$$Y{=}Z$$
$$X{=}Z$$

if both $X = Y$ and $Y = Z$ is generated, then $X = Z$ must be true also. This can then be propagated to the SAT solver.

For guarded constraints, this does not work. If we have the satisfiable formula

$$\{a \rightarrow X = Y\}$$
$$\{b \rightarrow Y = Z\}$$
$$\{c \rightarrow X = Z\}$$
$$\{a\}$$
$$\{b\}$$
$$\{\neg c\}$$

In a partial model where both a and b are true, we can not infer c, since that would cause a conflict.

5.6 Combining online and offline integration

It is possible to combine offline and online integration by adding the cheap infeasibility detector from section 6.7 online, and using a complete ILP programming procedure offline.

Chapter 6

Analysing infeasible integer programming problems

When an integer programming problem is infeasible, it is often interesting to find out the reason for the infeasibility. This example

$$\begin{array}{ll} (1) & x_1 > 0 \\ (2) & x_2 - x_1 = 1 \\ (3) & 2x_3 - x_1 \leq 2 \\ (4) & x_2 > 1 \\ (5) & x_3 > 0 \\ (6) & x_1 < 0 \end{array}$$

is infeasible, and the reason of the infeasibility is the constraints 1 and 6, which are mutually exclusive. A reason such as this is called an *Infeasible Irreducible System*, or sometimes an irreducibly inconsistent system. In this chapter we describe a few different methods of identifying such reasons. Methods for IISs can be found in [20, 19] for LP problems, and , [29] for ILP. The methods for finding IISs that are applicable for ILP have been included here. These methods are useful for finding explanations of infeasible systems in the decision procedure. A cheap incomplete method of finding infeasible systems is presented last.

6.1 Infeasible irreducible system

Definition 6.1.1 *An* Infeasible Irreducible System *(IIS) is a minimal set of infeasible constraints.*

This means that a system is an IIS iff

- it is infeasible.

- it is not possible to remove any of the constraints from the set without making it feasible.

Any infeasible system contains at least one IIS. Lets look at a few examples. The system

$$\begin{aligned} (1) \quad & x_1 > 0 \\ (2) \quad & x_2 - 2x_1 = 1 \\ (3) \quad & x_2 \leq 1 \end{aligned}$$

is an IIS, since it is infeasible, and it is impossible to remove any of the constraints without making it feasible. The system

$$\begin{aligned} (1) \quad & x_1 > 0 \\ (2) \quad & x_2 - 2x_1 = 1 \\ (3) \quad & x_2 \leq 1 \\ (4) \quad & x_1 < 0 \end{aligned}$$

on the other hand, is not an IIS. It is possible to remove both 2 and 3 without making the system feasible. Note that if 4 is removed, neither 2 nor 3 can be removed. This means that there are two IISs for this system, $\{1, 2, 3\}$ and $\{1, 4\}$. It is also possible to have an IIS with only one constraint as in this example:

$$2x = 1$$

The system does not have any integer solutions, so it is infeasible, and an IIS.

6.2 Deletion filtering

The simplest IIS filtering algorithm works like this. We know that any infeasible constraint problem contains at least one IIS. If the problem is still infeasible when one of the constraints is removed, the remaining constraints contains at least one IIS. This idea can be applied iteratively, until it is impossible to remove a constraint without making the problem feasible. The IIS that is found is the one whose first member is dropped last from the constraint set. In this example

$$\begin{aligned} (1) \quad & x_1 < 0 \\ (2) \quad & x_2 - x_1 = 1 \\ (3) \quad & x_2 > 3 \\ (4) \quad & x_3 - x_2 = 0 \\ (5) \quad & x_1 > 1 \\ (6) \quad & x_3 < 1 \end{aligned} \tag{6.1}$$

deletion filtering will find the IIS $\{3, 4, 6\}$ if the constraints are removed from top to bottom. If the order is reversed, the algorithm will find $\{1, 2, 3\}$

instead. Removing constraints in the order in which they appear in the constraint problem is called *forward* filtering. Removing constraints from the bottom up is called *reverse* filtering. To find the IIS $\{1, 5\}$, a different ordering must be chosen. The algorithm must solve one constraint problem for each constraint on the original system C.

Algorithm 6.1 Deletion filtering

Require: C is an infeasible set of constraints
  for all $c \in C$ **do**
    temporarily drop c from the set
    if C is feasible **then**
      return c to the set
    else
      drop c permanently
    end if
  end for
Ensure: C is an IIS

Some of the constraint problems that have to be solved during filtering may be difficult, and in that case the algorithm can be modified [29]. If the constraint problem takes too much time to solve when a certain constraint have been removed, that constraint is marked as *dubious* and kept in the constraint set to ensure infeasibility. If the resulting set contains any dubious constraints, then the set may not be an IIS. It is still infeasible, and hopefully a good approximation of the IIS.

6.3 Additive filtering

A similar algorithm is called the additive filter. The basic idea is this. Starting with an empty set of constraints, and adding one constraint at a time until the set becomes infeasible, then the last constraint added is a member of an IIS. All constraints but the last one is dropped, and the procedure starts over. When the set of constraints which have been identified as being a member of an IIS is infeasible, this set is an IIS. Just as in deletion filtering there are two orderings, forward and reverse filtering. In example 6.1, using forward filtering would first identify constraint 3, then 2, and lastly 1, forming the IIS $\{1, 2, 3\}$. With reverse filtering, $\{3, 4, 6\}$ would be identified.

It is possible to improve on this algorithm by combining the additive filter with the deletion filter. When the first IIS member have been identified, the set I is infeasible and therefore containts at least one IIS. By applying the deletion filter on this set, this can be identified.

Algorithm 6.2 Additive filtering

t!

Require: C is an infeasible set of constraints
  $I \leftarrow \emptyset$
  repeat
    $T \leftarrow \emptyset$
    for all $c \in C \backslash I$ **do**
      $T \leftarrow T \cup \{c\}$
      if T is infeasible **then**
        $I \leftarrow I \cup \{c\}$
        break
      end if
    end for
  until I is infeasible
Ensure: I is an IIS

Algorithm 6.3 Additive/deletion filtering

Require: C is an infeasible set of constraints
  $T \leftarrow \emptyset$
  for all $c \in C$ **do**
    $T \leftarrow T \cup \{c\}$
    if T is infeasible **then**
      break
    end if
  end for
  $T \leftarrow \text{deletion}(T)$
Ensure: T is an IIS

6.4 Elastic filtering

An *elastic* constraint is a constraint with an extra, elastic variable(s) which allow the constraint to "stretch" its feasible region. An elastic variable is a real nonnegative variable which introduces elasticity to the constraint, in this way

Nonelastic	Elastic
$\mathbf{a}^t\mathbf{x} \leq b$	$\mathbf{a}^t\mathbf{x} - e \leq b$
$\mathbf{a}^t\mathbf{x} = b$	$\mathbf{a}^t\mathbf{x} + e - e' = b$

Any infeasible system can be made feasible by making all constraints elastic in this way. By setting the objective function to minimize the sum of these elastic variables, it is possible to identify members of an IIS. Since the solution is minimal, only constraints that belong to an IIS will be stretched. The set identified by the algorithm *contains* at least one IIS, but is not necessarily an IIS itself. In order to find an IIS using this algorithm, it must be coupled with another algorithm (which does return an IIS). The algorithm chosen is usually deletion filtering, which is applied to the the set of enforced constraints. In many cases, the set of enforced constraints resulting from elastic filtering is small compared to the original constraint set.

Algorithm 6.4 Elastic filtering

Require: C is an infeasible set of constraints
  make all constraints elastic by adding nonnegative elastic variables e_i
  Set objective function to min $\sum_i e_i$
  while C is feasible **do**
    Enforce the constraints where $e_i > 0$ by removing those elastic variables
  end while
Ensure: the set of enforced constraints contains at least one IIS

The resulting set contains a smallest cardinality IIS. It is easy to see why; For each iteration, at least one of the constraints in a smallest cardinality IIS will be stretched. When all of them have been streched, the set of enforced constraints is infeasible and the algorithm terminates. Because of this, the algorithm will iterate at most as many times as the cardinality of the smallest IIS in the original constraint set.

In example 6.1, elastic programming will first make all constraints elastic by adding elastic variables

$$
\begin{aligned}
(1) \quad & x_1 - e_1 \leq -1 \\
(2) \quad & x_2 - x_1 + e_2 - e_3 = 1 \\
(3) \quad & -x_2 - e_4 \leq -4 \\
(4) \quad & x_3 - x_2 + e_5 - e_6 = 0 \\
(5) \quad & -x_1 - e_7 \leq -2 \\
(6) \quad & x_3 - e_8 \leq 0
\end{aligned}
$$

Solving this problem using the objective function $\min \sum_i e_i$ will stretch e_1, e_4, e_5 and e_7. Those variables are then removed, and now the problem has become infeasible. The set of enforced constraints is

$$\begin{array}{ll} (1) & x_1 \leq -1 \\ (3) & -x_2 \leq -4 \\ (4) & x_3 - x_2 = 0 \\ (5) & -x_1 \leq -2 \end{array}$$

With deletion filtering, the IIS $\{1, 5\}$ can be identified.

The fact that the result from elastic filtering contains a smallest cardinality IIS does not mean that it will be identified. In example 6.1, if we rewrite all equality constraint into inequality, we would get the following problem:

$$\begin{array}{ll} (1) & x_1 - e_1 \leq -1 \\ (2.1) & x_2 - x_1 - e_2 \leq 1 \\ (2.2) & -x_2 + x_1 - e_3 \leq -1 \\ (3) & -x_2 - e_4 \leq -4 \\ (4.1) & x_3 - x_2 - e_5 \leq 0 \\ (4.2) & -x_3 + x_2 - e_6 \leq 0 \\ (5) & -x_1 - e_7 \leq -2 \\ (6) & x_3 - e_8 \leq 0 \end{array}$$

This problem is then solved using the objective function $\min \sum_i e_i$. The solution will stretch the elastic variables e_4 and e_7, and constraints 3 and 5 are enforced. Then the solver is run again, this time e_1, e_2 and e_6 are stretched. After enforcing the corresponding constraints, the problem becomes infeasible, and the set of enforced constraints is

$$\begin{array}{ll} (1) & x_1 \leq -1 \\ (2.1) & x_2 - x_1 \leq 1 \\ (3) & -x_2 \leq -4 \\ (4.2) & -x_3 + x_2 \leq 0 \\ (5) & -x_1 \leq -2 \end{array}$$

This set contains two IISs, $\{1, 5\}$ and $\{1, 2.1, 3\}$. Which of these will be identified depends on the filtering algorithm, and on the order of the constraints. With forward deletion filtering and the current order, the IIS is identified as $\{1, 5\}$. With backward filtering $\{1, 2.1, 3\}$ is identified.

6.5 Discussion

Of the algorithms discussed, elastic filtering will typically be faster, since the number of constraint problems that have to be solved to find the IIS is much smaller on average. There are two drawbacks with elastic filtering that may be important.

- In elastic filtering most of the constraint problems that have to be solved are feasible, and an optimal solution has to be found. For deletion filtering most problems will be infeasible. Also, optimal solutions are not necessary for either the deletion or additive filter.
- In cases where there are several IISs in the constraint problem, it may be difficult to control which of these elastic filtering identifies.

6.6 Finding more than one IIS

In cases where there are several IISs in a constraint problem, it may be useful to try to identify more than one of them. We will look at two simple ways of doing that by iteratively applying one of the previous filtering algorithms.

Definition 6.6.1 *Two IISs are said to* overlap *each other if they share at least one member [20].*

Definition 6.6.2 *A* cluster *of IISs is a maximal set of IISs such that each IIS overlaps at least one other IIS in the cluster [20].*

One simple way of finding several IISs is to locate one from each cluster. This can be done by iteratively locating one IIS using one of the previous algorithms, and removing that until the remaining constraint problem becomes feasible. The algorithm locates at least one IIS from each cluster in

Algorithm 6.5 Cluster filtering

Require: C is an infeasible set of constraints
  $T \leftarrow \emptyset$
  while C is infeasible **do**
    $I \leftarrow \text{filter}(C)$
    $T \leftarrow T \cup \{I\}$
    $C \leftarrow C \backslash I$
  end while
Ensure: T is a set of IISs.

the original system, and all IISs are non-overlapping. Instead of removing the entire IIS in each iteration, it would be sufficient to remove one of the members. This may lead to more IISs being identified, since overlapping IISs can now be identified. In example 6.1 we can identify $\{3, 4, 6\}$ and $\{1, 5\}$ with cluster filtering and forward deletion filtering. With overlapping filtering, it is possible to identify all three IISs in this way: First, we identify $\{3, 4, 6\}$ and remove 6. Then we identify $\{1, 5\}$ and remove 5. last we identify $\{1, 2, 3\}$ and remove 3. The remaining constraints $\{1, 2, 4\}$ are feasible and the algorithm stops.

Algorithm 6.6 Overlapping filtering

Require: C is an infeasible set of constraints
  $T \leftarrow \emptyset$
  while C is infeasible **do**
    $I \leftarrow \text{filter}(C)$
    $T \leftarrow T \cup \{I\}$
    $C \leftarrow C \backslash \{i\}$, where $i \in I$
  end while
Ensure: T is a set of IISs.

6.7 An incomplete infeasibility detector for ILP

This section describes a cheap procedure that can detect infeasible constraint problems, and also find approximations of IISs for infeasible problems.

6.7.1 Preliminaries

We have a total ordering on all variables $\prec$. A total ordering is one where if $i \neq j$, then either $v_i \prec v_j$ or $v_j \prec v_i$.

A constraint is on *normal form* if the greatest common divisor $\gcd(a_1, \ldots, a_n, b)$ of the factors and the right hand side is 1. Any constraint can be rewritten to normal form by dividing every coefficient and the right hand side by $\gcd(a_1, \ldots, a_n, b)$.

An equality is a *definition* if the coefficient of the greatest variable (in the chosen variable ordering) is 1 or -1. Definitions can be rewritten on the form $x_k = \sum_{i \neq k} a_i x_i + b$.

6.7.2 The algorithm

The basic idea is that if we have two constraints

$$x < b_1$$
$$x > b_2$$

these two constraints are infeasible iff $b_1 \leq b_2$. An algorithm that uses this fact take one constraint at a time, simplifies it by applying all known definitions and eliminating a number of variables. If the new simplified constraint is a definition, this definition is applied to all known constraints, eliminating one more variable. Then the new constraint is added to the set of known constraints. If any pair of known constraints that are of the form $\pm x \leq b$, contradicts each other, an infeasiblility has been detected. The algorithm can find explanations of infeasibilities by keeping track of which definitions are used when simplifying constraints. When a pair of conflicting constraints is discovered, the reason is then besides the pair, the definitions

Algorithm 6.7 Detect infeasibilities

Require: C is a set of constraints
  $T \leftarrow \emptyset$
  for all $c \in C$ **do**
    Use all definitions in T to simplify c
    if c is a definition **then**
      simplify all constraints in T with the definition c
    end if
    $T \leftarrow T \cup \{c\}$
    if T has a pair $(x \leq b_1, -x \leq b_2)$ where $b_1 < -b_2$ **then**
      return infeasible
    end if
  end for
  return unknown

that were used, directly or indirectly, to simplify those two constraints to their final form.

The order in which the constraints are added is important. There are two variants; Forward method where constraints are added in the order in which they appear in the set of constraints, and backward where the constraints are added in the reverse order. When an infeasibility has been detected by the procedure, it is possible to continue adding constraints from the constraint set, perhaps discovering more infeasible subsystems.

If a pair of constraints

$$\begin{aligned} x &\leq \ \ b \\ -x &\leq -b \end{aligned}$$

is discovered, this can easily be recognized as a definition $x = b$.

6.7.3 An extension

A possible extension is the recognize that the constraints

$$\begin{aligned} \mathbf{a}^t\mathbf{x} &\leq \ \ b \\ -\mathbf{a}^t x &\leq -b \end{aligned}$$

correspond to the equality constraint $\mathbf{a}^t\mathbf{x} = b$.

6.8 Preprocessing constraints

If every IIS can be detected during preprocessing, then the resulting propositional formula would be be a complete characterization of satisfiability of the original formula. That is, the propositional formula would be satisfiable if and only if the original formula is satisfiable. The SAT solver would then

not generate any spurious models, and it would not be necessary to switch back and forth between Psat and Csat iteratively in algorithm 5.1. Finding all IISs are however an expensive operation in the general case [1]. This is done in [45], using the Omega test.

It is usually not necessary to find all IISs, however. By looking at the IISs identified without preprocessing, we can see that a small subset of IISs are sufficient, if the "right" IISs can be identified.

6.9 Using the incomplete procedure

The incomplete procedure can be used in three different ways:

- In offline integration in combination with an ILP solver. For every SAT model, the generated constraints are first tested for infeasibility in the incomplete procedure. If the procudre could not find an infeasibility, the ILP solver is tried. The idea is that most constraint problems are infeasible, and the infeasibilities are usually simple enough to be detectable with the incomplete procedure.

- In online integration, instead of an ILP solver. The incomplete procedure is placed online in the SAT solver, and an ILP solver placed offline.

- In preprocessing. Before the SAT solver starts, all constraints are tested in the incomplete procedure. For all infeasibilities that are detected, a clause preventing the SAT solver from generating a model wich generates that specific subset of constraints is added to the formula.

[1] NP-hard in LP, worse in ILP, I think. ILP itself is NP-complete, so we can't verify a solution in polynomial time. In LP that is possible, since LP is polynomial. I have not found a proof for this, however.

Chapter 7

Analysis

7.1 Experimental setup

The tests were performed on a computer with a 1.8 GHz Pentium 4 processor with 512 kB cache, and 512 MB RAM running Linux.

7.2 Test suite description

The test suite is comprised of a number of Lustre programs together with properties on the programs. A test is one program/property pair. Since many programs have several properties, the same program is used in several tests. For programs which has more than one property, there is also a test where the property is the conjunction of all properties on the program. The test suite consists entirely of academic examples of Lustre programs. Several are for instance models of cache coherence protocols [9].

The tests are chosen such that they can be verified with any tool, given enough time. Those tests which can not be verified with one or more tools have been eliminated.

The test suite consists of 72 tests of varying complexity. All tests are fairly small, as can be seen in figure 7.1, which shows the number of integer and boolean variables for the tests.

7.3 The tool Rantanplan

Most of the ideas in the previous chapters have been implemented in a tool called Rantanplan [16]. The tool is based on Luke, and uses the Lustre parser and induction code as is. The SAT solver have been changed to MiniSat [24]. There are two reasons for this; Partly because MiniSat seems to outperform the old solver Satnik, but primarily because MiniSat is a well designed and well documented solver with a ready-made interface for implementing the necessary extensions. The integer programming package that is being used

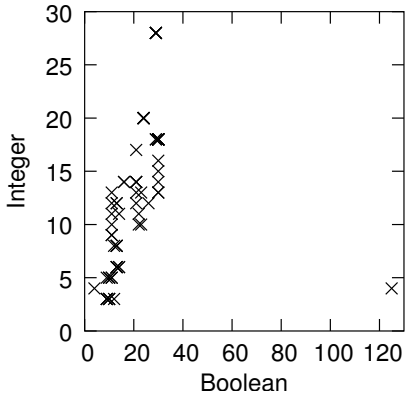

Figure 7.1: The number of boolean and integer variables for the different tests.

is GLPK, a free mixed integer linear programming package distributed by the GNU Project.

7.3.1 Formalizations

There are three formalizations "Per operator", "Maximal" and "Guarded maximal". All three formalizations are implemented.

7.3.2 Offline integration

There are two decision procedure which can be used offline; GLPK and the incomplete infeasibility detector, called Pooh [38]. Pooh comes in several variants: It can work both forwards and backwards. The algorithm can also be implemented to either only find one explanation of infeasibilities, or several.

7.3.3 Online integration

In online integration, it is possible to use either GLPK, or forward, single explanation Pooh or no procedure.

7.3.4 IIS filtering

A host of IIS filtering algorithms have been implemented: Deletion, reverse deletion, additive, reverse additive, additive/deletion, reverse additive/deletion and elastic/deletion filtering.

7.3.5 Multiple IISs

The filtering algorithms above can either be used to find a single IIS, or together with cluster filtering 2 to find a set of IISs.

7.3.6 Preprocessing of constraints

The constraints can be preprocessed before every step case, with either GLPK or Pooh. The versions of Pooh that can be used are forward and backward Pooh, both with multiple explanations.

Using the same in-place variable for all occurencies of a constraint is not done in Rantanplan.

7.4 Test plan

Not every combination of parameters is tested, some of them are left fixed. This is because the tests would take a inordinate amount of time with all combinations of parameters. Instead method has been fixed to "Guarded maximal".

NBAC was run with the following flags: `+eliminput --cudd "-maxmem 128"` The memory limit for CUDD has been increased to 128 MBytes since some examples exceeded the default limit. `+eliminput` is necessary because some properties reason about the input signals in the current time point.

Luke represents integers a vectors of bits using propositional literals. The size of these bit vector can be changed by the user, and for these tests, Luke was run with the default of 16 bit integers. A modified version of Luke version 0.4beta is used form comparison. The SAT solver has been changed to MiniSat, for performance reasons.

7.5 Analysis

Before starting to analyse the different tools, the goal of the analysis should be specified. The aim of the analysis is to answer these two questions:

- Is my tool competitive or complementary to Luke and NBAC?
- Which combinations of parameters gives good performance?

In order to answer this question, I will first look at my tool and try to find one or more good combination(s) of the different ideas covered in the earlier chapters. These combinations will then be compared to NBAC and Luke.

Parameter	P value
Offline	< 0.0001
Online	< 0.0001
Filter	< 0.0001
Multi IIS	< 0.0001
Preprocess	0.0451

Table 7.1: Dependence of parameters. A P value < 0.05 indicates that the parameter has an effect on performance.

Parameter	Offline	Online	Filter	Multi IIS	Preprocess
Offline	< 0.0001	< 0.0001	< 0.0001	< 0.0001	0.7533
Online		< 0.0001	< 0.0001	< 0.0001	< 0.0001
Filter			< 0.0001	< 0.0001	0.9986
Multi IIS				< 0.0001	0.8425
Preprocess					0.0451

Table 7.2: Tests of interaction between pairs of parameters. A P value < 0.05 indicates that a pair of parameters interact

7.5.1 Dependence on parameters

The first interesting question is: Does the parameters have an effect on performance in the test suite. In table 7.1 the probablilities that the parameters have no effect are shown. Since these probabilities are all well below 0.05 for all but Preprocess, we can safely say that the settings on all of these parameters matters to the performance on the test suite. For preprocessing the case is weaker.

7.5.2 Interaction between parameters

The next question may be if there are any interactions between the parameters, or if they are independent. For independent parameters, it is simple to find an optimal combination of parameters for the tool by optimizing each parameter individually. In table 7.2 the probabilities that a pair of variables do not interact are shown. We can see that there are no independent parameters, even if not all pairs interact. To see just how complex the interactions are, lets look at 3 term interactions. In figure 7.3, we can see that most triples interact. The triples that do not interact all include preprocessing. We can go further and test interaction between 4-tuples of parameters. Included in the figure is the only interesting 4-tuple where we have an interaction.

Since all parameters interact with at least one other parameter, find-

Parameters	P value
Offline, Online, Filter	< 0.0001
Offline, Online, Multi IIS	< 0.0001
Offline, Online, Preprocess	0.9229
Offline, Filter, Multi IIS	< 0.0001
Offline, Filter, Preprocess	1.0000
Offline, Multi IIS, Preprocess	1.0000
Online, Filter, Multi IIS	< 0.0001
Online, Filter, Preprocess	1.0000
Online, Multi IIS, Preprocess	0.9930
Offline, Online, Filter, Multi IIS	< 0.0001

Table 7.3: Tests of interaction between tuples of parameters. A P value < 0.05 indicates that those parameters interact

ing an optimal combination of values is not as simple as optimizing each parameter in turn.

7.6 Identifying candidates

I have tested the different combinations of parameters against the hypothesis that the medians of execution times of different parameters follow a normal distribution. This is visualized with the Q-Q plot in figure 7.2. Each circle in the plot is a certain combination of parameters. If the hypothesis was true, the observations would follow a straight line across the plot. Those combinations where this is not true deviates from this line. We can see that there a two clusters which deviates; The ones in the lower left, which are better than expected, and the ones in the upper right which are worse than would be expected. The combinations of parameters which are better than expected are the ones that are interesting. There are 126 different combinations in the lower left cluster (where the median is lower than 0.3).

It is interesting to also look at the 90th percentile. The median is the value where 50% of the examples have a shorter execution time and 50% have a slower execution time. The 90th percentile is the point where 90% of the examples have a shorter execution time and 10% a longer execution time. By plotting the median and the 90th percentile against each other, we can identify those combinations of parameters that are both fast on average (using the median) and also fast in the worst case (using the 90th precentile). The result is in figure 7.3. The plot has been truncated, and only shows those combinations of parameters where the 90th percentile is lower than 10 seconds. Three or four different clusters can be identified; The combinations which are fast overall and on difficult examples, those

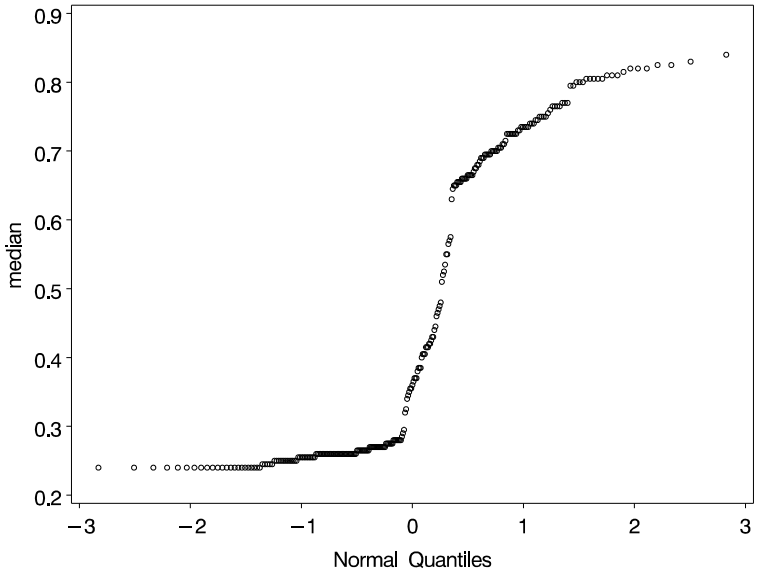

Figure 7.2: Tests if the medians of execution times for different combinations of parameters are follow a normal distribution.

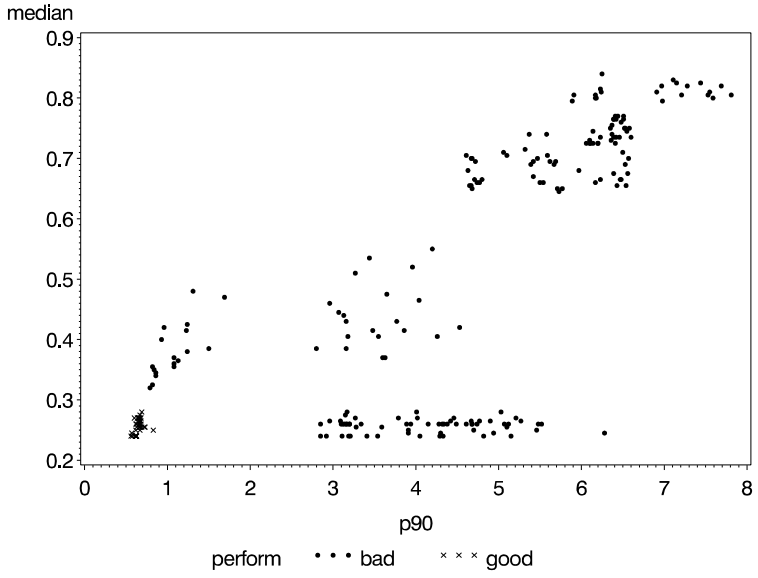

Figure 7.3: Scatter plot of the median and the 90th percentile of different combinations of parameters.

Offline	Online	Filter	Multi IIS	Preprocess
GLPK	Pooh	{deletion,rdeletion}	Yes	*
Pooh,GLPK	Pooh	{deletion,rdeletion}	Yes	*
PoohM,GLPK	Pooh	{deletion,rdeletion}	Yes	*
RPooh,GLPK	{None,Pooh}	{deletion,rdeletion}	Yes	*
RPoohM,GLPK	{None,Pooh}	{deletion,rdeletion}	Yes	*

Table 7.4: The "good" combinations of parameters. Pooh = forward Pooh, PoohM = forward multiple Pooh, RPooh = reverse pooh, RPoohM = reverse multiple Pooh, rdeletion = reverse deletion, None = no procedure

which are fast overall but slow on difficult examples, and those which are not fast overall. The fast combinations are indicated in the plot as crosses and the others are drawn as dots. In the interesting cluster, there are 42 different combinations of parameters. Looking more closely at those reveals an interesting pattern. The combinations are shown in table 7.4, and several conclusions can be drawn from this.

- For online integration it seems that the integer programming package GLPK is too expensive. The reason is that in online integration a large number of feasible intermediary ILP problems have to be solved, which when adding a few more constraints will usually either become infeasible, or the SAT solver will detect a conflict and backtrack. This problem seems to outweigh the benefit of detecting infeasible ILP problems "early".
- Elastic filtering is worse than deletion filtering. This does not agree with the literature on IIS filtering [20], where elastic filtering is claimed to be superior. There are two reasons for this difference. In elastic filtering, most of the (M)ILP problems that have to be solved are feasible, and an optimal solution has to be found. In deletion filtering on the other hand, most of the ILP problems are infeasible, and usually, the LP relaxation is also infeasible. These infeasibilities are much cheaper to detect than finding the optimal solutions in elastic filtering. This means that even if deletion filtering have to solve more ILP problems to find an IIS, each problem is so much cheaper that deletion filtering is still faster.
- For these specific combinations of parameters, it is worthwhile to locate several IISs, rather than just one. Interestingly, the combinations which give the worst performance also locates multiple IISs, but then in combination with elastic filtering.
- It does not matter if preprocessing is performed or not. This is of course only true for the specific preprocessing algorithms that have

Offline	Online	Filter	Multi IIS	Preprocess
GLPK	Pooh	{deletion,rdeletion}	Yes	None
Pooh,GLPK	Pooh	{deletion,rdeletion}	Yes	None
PoohM,GLPK	Pooh	{deletion,rdeletion}	Yes	None
RPooh,GLPK	{Pooh}	{deletion}	Yes	None
RPoohM,GLPK	{None,Pooh}	{deletion,rdeletion}	Yes	None

Table 7.5: The selected combinations of parameters. Pooh = forward Pooh, PoohM = forward multiple Pooh, RPooh = reverse pooh, RPoohM = reverse multiple Pooh, rdeletion = reverse deletion, None = no procedure

been tested here (forward and reverse Pooh, with multiple explanations).

Of these 42 combinations the ones with the lowest median and a 90th percentile below 1 is selected for comparisons with the other tools. There are 11 such combinations, shown in table 7.5.

7.7 Comparisons with NBAC

One way of comparing a number of tools against each other is to look at how many examples they can verify within a certain time. It is possible to calculate the probability that a tool has verified an example as a function of execution time. This is done in what is called *survival analysis*. The idea is to calculate the probability that a certain tool is able to verify a given example, as a function of execution time. In figure 7.4 these functions have been plotted for NBAC and the 11 "good" combinations of parameters which had the best median.

For most of the test suite, NBAC and my tool are more or less comparable, but there are some examples where NBAC is considerably slower. In figure 7.5 all examples which takes more than 10 seconds in NBAC have been removed. We can see that on some examples NBAC is faster and on others Rantanplan is faster.

7.8 Comparison with Luke

Luke can be tested in the same way. The result is shown in figure 7.6. The same conclusion as in the comprisons with NBAC applies here. There seems to be a large subset of the test suite where the tools are comparable, and a subset where Luke is very slow. The difference between the two is very sharp. Either Luke is fast, or it is very slow. There does not seem to be anything in between. If the examples which are difficult for Luke is removed

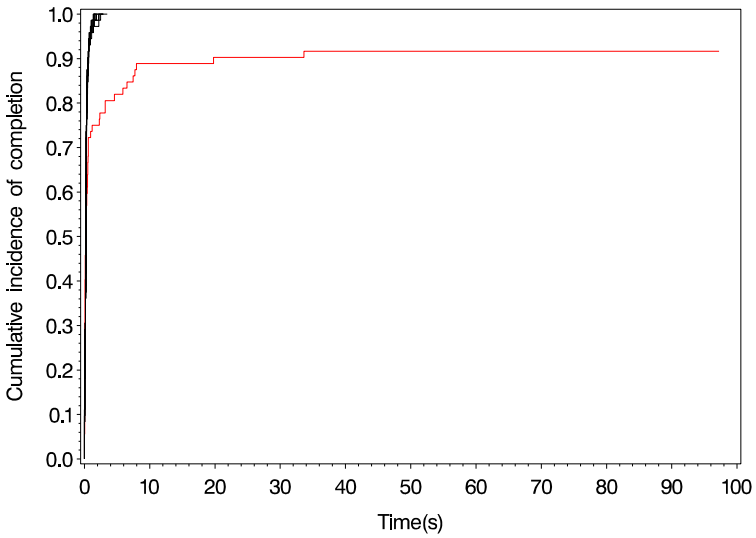

Figure 7.4: Comparison with NBAC. The vertical axis shows the probability that the tool has terminated on a given test as a function of time.

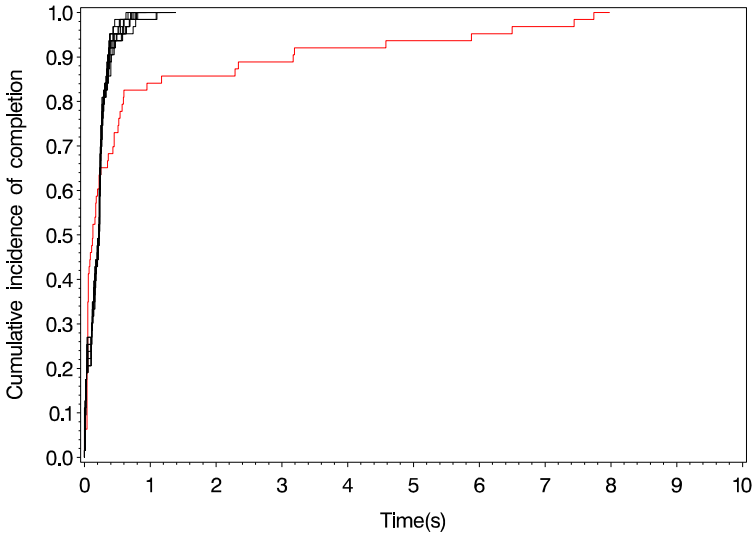

Figure 7.5: Comparison with NBAC on easy Lustre examples. The vertical axis shows the probability that the tool is still executing as a function of time.

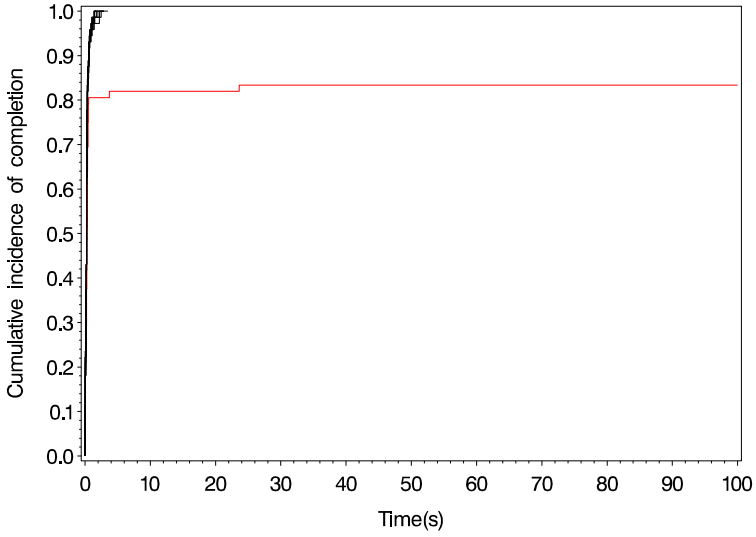

Figure 7.6: Comparison with Luke. The vertical axis shows the probability that the tool is still executing as a function of time.

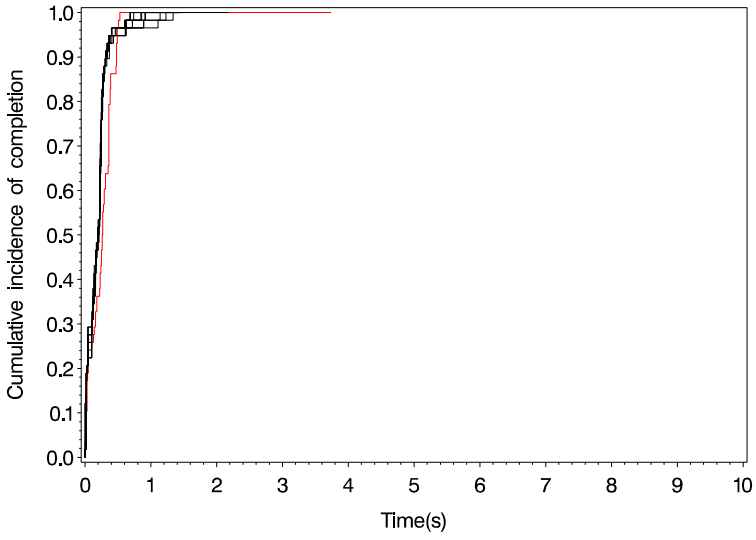

Figure 7.7: Comparison with Luke on easy Lustre examples. The vertical axis shows the probability that the tool is still executing as a function of time.

Comparison	Quotient
Rantanplan/Luke	0.68 [0.42, 1.07]
Rantanplan/NBAC	0.64 [0.41, 1.02]
NBAC/Luke	1.05 [0.66, 1.66]

Table 7.6: Comparison of Rantanplan and NBAC/Luke. The table shows the quotient of the geometric mean of execution times with a 95% confidence interval.

as in figure 7.7, we can see that Rantanplan seems slightly better on the remaining examples, but the difference is not very large.

In table 7.6, the percieved differences in the earlier plots are shown to be too small to indicate any statistically significant difference between the tools. The data used are the test which takes less than 10 seconds to prove in any tool.

7.9 Explanation of the differences

The examples where NBAC and Luke have trouble all have one thing in common: They use the sum of two or more variables in expressions. All other examples only add or subtract to variables by a constant, usually by 1.

NBAC also have trouble with conjunctions of several properties.

7.10 Other examples

The examples which can not be verified with all (three) tools are not part of the test suite, and this may bias the tests. The original test suite consisted of 137 tests. The reasons for exclusion are listed here.

7.10.1 Invalid properties

NBAC does not have built-in support for discovering that a property is invalid. A tool NBAC2LUCKY will soon be publicly available to interface to the symbolic simulator Lucky [34] to aid in finding counter-examples.

Induction is complete for invalid properties, so both Luke and Rantanplan can find the shortest possible counter-example for an invalid property. Luke outperforms Rantanplan on invalid properties with longer counter-examples, as can be seen in figure 7.8. The test in the figure has counter-examples ranging from 2 to 7 steps.

There are 13 invalid tests in the test suite.

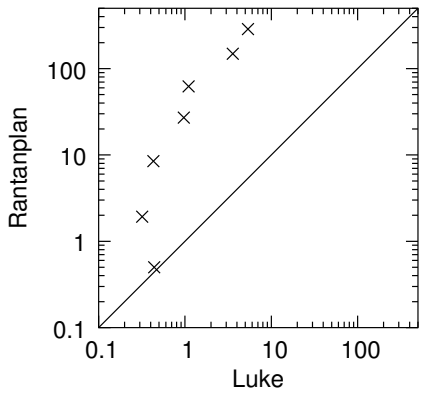

Figure 7.8: Comparison of execution times between Luke and Rantanplan on invalid properties with increasingly long counter-examples

7.10.2 Bounded integers

Some examples needed unbounded integers. These examples could be verified in NBAC and my tool, but in Luke, they produced a counter-example.

```
node Counter() returns ( OK : bool );
    var C : int;
let
    C = 0 → pre C + 1;
    OK = C ≥ 0;
tel
```

This example is verifiable in Rantanplan, but since Luke uses bounded integers the step cases will always fail, and eventually a counter-example will be found where the stream C overflows.

There are 10 examples which require unbounded integers in the test suite.

7.10.3 Limitations with GLPK

The integer programming package GLPK uses branch-and-bound and the Simplex algoithm, which is an incomplete procedure. Because of this, there are some Lustre programs which can not be verified in my tool. Below is an example which can not be verified in Rantanplan because of this limitation

```
node Problem( What : int ) returns ( R1 : bool );
   var Last4 : int;
let
```

```
    Last4 = ( 0 → pre Last4 mod 1000 * 10 ) + What;

    R1 = 0 ≤ What and What ≤ 9 ⇒ Last4 mod 10 = What;
tel
```

The property can easily be verified in Luke.

There are 5 test in the test suite where the constraint problems can not be verified in GLPK.

7.10.4 Non-linear expressions

Since Rantanplan uses integer linear programming, Lustre programs with non-linear expression can not be handled. There are no such examples in the test suite.

7.10.5 Incompleteness of induction

There are some examples where there is an infinite path of valid non-reachable states which lead to an invalid states. These examples can not be verified in my tool.

There are 14 such examples in the test suite.

7.10.6 modulo not supported in NBAC

The modulo operator is not supported in NBAC. There are 5 test in the test suite which uses the modulo operator.

7.10.7 Other problems

There are a few other problems where possible bugs which for reasons unknown can not be verified. There is for instance somewhere in the Haskell part of my tool a function where I and GHC does not agree on whether or not it is tail-recursive. On other examples NBAC seems to think that the problem is too large. The tool hits an internal limit on the size of the problems it can handle, but changing the limit does not seem to have any effect.

There are 18 such examples in the test suite.

Chapter 8

Related work

8.1 Verification of Lustre programs

There are several systems for verification of safety properties for Lustre programs. Some of these are described here.

LESAR

LESAR is based on model-checking, and capable of verifying safety properties involving boolean streams. There is some support for integers, but it is very weak, as can be seen in [33].

NBAC

NBAC is a model-checking tool based on abstract interpretation [35]. The tool supports both bounded and unbounded integers as well as booleans. Lustre programs can not consist exclusively of booleans however, this is possibly just an oversight.

Luke

Luke[1] is a tool for inductive verification of safety properties, where the decision procedure used is a SAT solver. It supports bounded integers by translating these to vectors of literals (bit vectors).

Gloups

Gloups [17] is an automatic generator of proof obligations from Lustre to PVS. The tool is not currently available.

[1] Available at the time of writing on `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/form/luke.html`

SCADE

SCADE is a commercial development environment from Esterel Technologies. The verification engine[1] uses a combination of Stålmarcks method, DPLL style SAT solving, BDDs, linear programming and constraint solving. Exactly how is not entirely clear from the paper, but presumably in a way similar to the approach taken here. The tool seems to support rational numbers and booleans. For bounded ratioals, a bit vector representation is chosen to make it complete with non-linear arithmetic.

8.2 Combining SAT solving with arithmetic

Several attempts have been made to combine propositinal logic with arithmetic. The relevant ones are descibed here.

MathSat

MathSat [7, 15, 42] combines propositional logic with arithmetic over the reals. The DPLL style SAT solver MiniSat [24] is combined with a hierarchy of several increasingly powerful solvers. The idea is to be able to detect infeasibilities with a cheap procedure, and avoid having to run expensive complete procedures unless absolutely necessary.

TSAT++

TSAT++ combines SAT solving with *difference constraints*. These are constraints on the difference between two variables $x - y \leq c$. In the constraint problems generated in Rantanplan, constraints will often have more than two variables, and the efficient decision procedure for difference constraints can not be used. The system uses preprocessing where all infeasible pairs of constraints are detected. This is reported as useful on random formulas, but the constraint problems generated in Rantanplan have a lot of structure.

ICS

ICS[23, 22] is a system which supports propositional logic extended with rationals, integers, bit vectors, uniterpreted functions, non-linear arithmetic, et.c. Both induction and bounded model chacking has been implemented on top of ICS, but it does not take advantage of an incremental SAT solver to increase efficiency.

HySat

The tool HySat [27, 12] uses a "pseudo-boolean" SAT solver[2] in combination with linear programming using GLPK for bounded model checking of discrete-continuous systems. The tool has uses three optimizations; Isomorphy inference and a custom tailored variable ordering in the SAT solver. Isomorphy inference means to detect subsets of constraints that are isomophic to the IISs that have already been discovered. Two sets of clauses are isomorphic if there is a substitution which makes them equal. All subsets of constraints which are isomorphic to an infeasible subset are also infeasible. A conflict clause for those subsets can be added to the SAT problem.

DPLL(T)

DPLL(T) [47] is a framework for integrating decision procedures into a SAT solver. The framework has been used to build a decision procedure for propositional logic extended with linear constraints. The distinguishing feature of DPLL(T) is the ability to propagate information from assignements of in-place variables. If a set of generated constraints implies one of more other constraints, then this information is propagated to the SAT solver.

8.3 Complete SAT characterization

In [45], a method of creating a complete SAT characterization of satisfiability of formulas with integer constraints is descibed. The method uses the Omega test [41]. The method is compared to ICS on a number of test suites with favorable results. Strichman notes that the time it takes to solve the SAT problem is very small compared to the time it takes to generate the SAT characterization.

[2] 0-1 programming

Chapter 9

Conclusions and future work

In this chapter some conclusion are drawn from the analysis, and a list of possible future work is outlined.

9.1 Conclusions

The result of the evaluation is this

- The method of combining SAT solving an integer programming is complementary to both the methods used in Luke and NBAC, at least for valid properties. This is despite the fact that the implementation of Rantanplan is not optimized in any way.
- There seems to be plenty of room for improvement on the tool as implemented here. Many of the suggestions in future work seems worthwhile to study.
- The tool is slower than Luke on larger induction depths and longer counter-examples. This difference may disappear if the procedure is improved, for instance with isomorphy inference (see 9.2.6).
- There is a class of Lustre examples where both Luke and NBAC is slow, but Rantanplan gives acceptable performance. This is not surprising, since by crafting the examples in just the right way, any given tool can show good performance. The examples in this case uses sums of two or more variables. As long as all expressions are addition or subtraction by a constant, the performance is comparable between the tools. An effort have been made here to not create "to many" of these examples. They have also been eliminated in the analysis in chapter 7.
- Implementing the procedure on top of MiniSat [24] is simple and straightforward. The SAT solver has a clear and easy to use interface for plugging in decision procedures, as well as good documentation.

9.2 Future work

There is a large number of possible extension or improvements to the procedure described in the thesis.

9.2.1 Use a complete ILP procedure

Integer programming packages based on branch-and-bound or branch-and-cut are not complete. There are complete methods based on Fourier-Motzkin elimination [41, 11]. They can either replace or complement a procedure based on branch-and-bound.

9.2.2 Support for more Lustre constructs

The Lustre language supported by the current tool is a fragment of Lustre. Some of the things that may be interesting to add support for are these:

Reals

Lustre supports real values in addition to booleans and integers. Adding support for reals would be relatively simple, since most integer programming packages (GLPK included) support mixed integer programming. A mixed integer programming (MILP) is a constraint problem where only a subset of the variables are required to be integer. Implementing this would become difficult in Luke, since floating point arithmetic is much more complicated to handle on the bit level.

Foreign functions

It is possible to call foreign functions in Lustre. Support for these may be added by simply modeling the result of a function call as a free variable.

Syntactic sugar

The is quite a bit of syntactic sugar that is not supported by Rantanplan. Some of the unsupported constructs are

- Datatypes
- Constants
- Vectors
- Recursive node definitions

Adding these would not pose any new problems, because of what they are; Syntactic sugar.

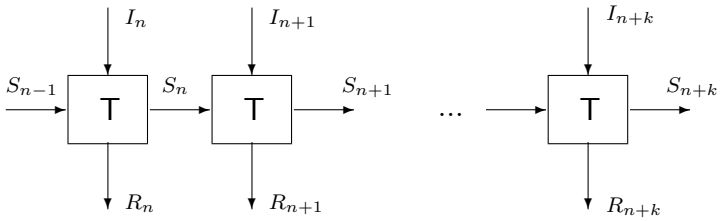

Figure 9.1: A sequence of transition functions

9.2.3 Incremental ILP

Placing an integer programming package online inside the SAT solver can be very expensive, since it has to solve a large number of intermediary feasible constraint problems. The only difference from one call of the package to the next is that one constraint have been added. An incremental version of such a package that takes advantage of having solved a nearly identical constraint problem before may reduce execution time.

9.2.4 Backwards instantiation

In Rantanplan, the fact that MiniSat is an incremental SAT solver is used to solve the increasingly larger SAT formulas required for the induction proofs. Transition functions are added one at a time, one every time the induction depth is increased. The transition functions that were used for the previous base case are kept, and the new transition function is placed "after" the last transition function. In figure 9.1 a new transition function would be added with input I_{n+k+1} computing the output R_{n+k+1}, and the new state S_{n+k+1}.

The idea is to add the new transition function "first", and let the transition which now has outputs R_{n+k} be used to computed R_{n+k+1} and the state instead S_{n+k+1}. When using an incremental SAT solver is it better to place it before the first instance instead. It would be worthwhile to investigate if that is true here as well.

9.2.5 Heuristics for faster IIS discovery

Many of the IISs that are discovered in offline integration are similar, or related to each other. It should be possible to take advantage of that in a heauristic which quickly identifies candidate members of an IIS in infeasible constraint problems.

9.2.6 Isomorphy inference

When a new transition function is added, the new constraints will contain subsets which are isomorphic to IISs which have already been identified. These subsets are also infeasible, and locating these is called *isomorphy inference* [27].

9.2.7 Complete SAT characterization

If during the preprocessing stage all IISs of the set of constraints can be discovered, then the SAT problem becomes a complete characterization of satisfiability for the original formula. This means that the propositional formula would be satisfiable if and only if the original formula is satisfiable.

Now it would not be necessary to integrate the ILP package with the SAT solver at all. Instead the SAT solver solves the propositional formula, and if the formula is satisfiable, there exists a model for the original formula. This can be found by solving the constraint problem generated from the SAT model. If the propositional formula is unsatisfiable, the original formula is also unsatisfiable.

Finding every IIS in the constraint set is an expensive operation. In linear programming it is an NP-complete problem. In integer programming it it worse. Integer programming itself is NP-complete, so there is no polynomial algorithm for verifying a solution.

It is probably not necessary to find *every* IIS, however. The structure of the formula will most likely prevent some IIS from being asserted in a SAT model. Because of this, there may be a method of finding enough of the IIS such that this search pays off in reduced execution time.

9.2.8 Improved incomplete procedure

In MathSAT, a hierarchy of increasingly powerful solvers are used [42]. In Rantanplan, only two solvers are used (Pooh, the incomplete infeasibility detector and the integer programming package GLPK). A similar idea can of course be used here.

9.2.9 Automatic strengthening of properties

It would be possible to automatically strengthen properties by discovering invariants in the Lustre code. This may reduce the induction dpeth required to prove a property, or even strengthen some properties that are otherwise impossible to verify with induction enough to make them provable.

Bibliography

[1] P. A. Abdulla, J. Deneux, G. Stålmark, H. Ågren, and O. Akerlund. Designing Safe, Reliable Systems using Scade. In *Proc. ISoLA '04: 1st International Symposium on Leveraging Applications of Formal Methods*, 2004.

[2] C. Aggarwal, R. K. Ahuja, J. Hao, and J. B. Orlin. Diagnosing infeasibilities in network flow problems. *Mathematical Programming*, 81:263–280, 1998.

[3] E. Amaldi, M. E. Pfetsch, and L. E. Trotter Jr. On the maximum feasible subsystem problem, IISs and IIS-hypergraphs. *Mathematical Programming*, 95:533–554, 2003.

[4] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based Procedures for Temporal Reasoning. In *Proceedings of ECP-99*, 1999.

[5] A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea. TSAT++: An Open Reasoning Platform for Satisfiability Modulo Theories. In *Proceedings of PDPAR*, 2004.

[6] A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. A SAT-based Decision Procedure for the Boolean Combination of Difference Constraints. In *Accepted to SAT 2004*, 2004.

[7] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Lecture Notes in Computer Science*, volume 2392, page 193. Springer Verlag, 2002.

[8] G. Audemard, M. Bozzano, R. Sebastiani, and A. Cimatti. Verifying Industrial Hybrid Systems with MathSAT. In *Electronic Notes in Computer Science*, volume 89, 2004.

[9] Automatic Verification of Parameterized Cache Coherence Protocols. Giorgio delzanno. In *Proccedings of CAV 2000*, 2000.

[10] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1993.

[11] U. Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, 1996.

[12] B. Becker, M. Behle, F. Eisenbrand, M. Fränzle, M. Herbstritt, C. Herde, J. Hoffmann, D. Kröning, B. Nebel, I. Polian, and R. Wimmer. Bounded Model Checking and Inductive Verification of Hybrid Discrete-continuous Systems. In *Proceedings GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 65–75, Kaiserslautern, Germany, February 2004.

[13] A. Bemporad and M. Morari. Verification of Hybrid Systems via Mathematical Programming. In *Lecture Notes in Computer Science*, 1569.

[14] P. Bjesse and K. Claessen. SAT-based Verification without State Space Traversal. In *Proceedings of FMCAD 2000*, 2000.

[15] M. Bozzano, A. Cimatti, G. Colombini, V. Kirov, and R. Sebastiani. The MathSAT Solver – a progress report. In *Proceedings of PDPAR 2004*, Cork, Ireland, July 2004.

[16] M. D. Bévère. *Sur la piste des Dalton*. Dargaud Lucky Productions, 1960.

[17] C. D. Canovas and P. Caspi. A PVS Proof Obligation Generator for Lustre Programs. In *Lecture Notes in Artificial Intelligence*, volume 1955, 2000.

[18] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining Constraint Solving and Symbolic Model Checking for a Class of Systems with Non-linear Constraints. In *Proceedings of CAV*, 1997.

[19] J. W. Chinneck. Finding a Useful Subset of Constraints for Analysis in an Infeasible Linear Program. *INFORMS Journal on Computing*, 9(2):164–174, 1997.

[20] J. W. Chinneck and E. W. Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

[21] L. de Moura and H. Rueß. Lemmas on Demand for Satisfability Solvers. In *Proceedings of SAT 2002*, 2002.

[22] L. de Moura, H. Rueß, and M. Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *Lecture Notes in Computer Science*, volume 2392, pages 438–455, 2002.

[23] L. de Moura, H. Rueß, and M. Sorea. Bounded Model Checking and Induction: From Refutation to Verification. In *Proceedings of Computer-Aided Verification*, 2003.

[24] N. Eén and N. Sörensson. An extensible SAT solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[25] N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. In *Proceedings of the First International Workshop on Bounded Model Checking*, 2003.

[26] M. Ernst, T. Millstein, and D. S. Weld. Automatic SAT-Compilation of Planning Problems. In *Proceedings of IJCAI*, 1997.

[27] M. Fränzle and C. Herde. Efficient Proof Engines for Bounded Model Checking of Hybrid Systems. In *Proceedings of FMICS'04*, 2004.

[28] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *16th International Conference on Computer Aided Verification (CAV)*, Boston (USA), July 2004.

[29] O. Guieu and J. W. Chinneck. Analyzing Infeasible Mixed-Integer and Integer Linear Programs. *INFORMS Journal on Computing*, 11(1):63–77, 1999.

[30] S. Gulwani and G. C. Necula. A Randomized Satisfiability Procedure for Arithmetic and Uninterpreted Function Symbols. In *Proceedings of CADE*, 2003.

[31] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[32] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[33] N. Halbwachs and P. Raymond. A Tutorial of Lustre. 2002.

[34] E. Jahier and P. Raymond. The Lucky language Reference Manual. Technical Report TR-2004-6, Verimag.

[35] B. Jeannet. Dynamic Partitioning in Linear Relation Analysis. Application to the Verification of Synchronous Programs. *Formal Methods in System Design*, 23(1):5–37, 2003.

[36] M. Ljung. Formal modelling and automatic verification of lustre programs using NP-Tools. Master's thesis, Royal Institute of Technology, 1999.

[37] J. Mikac and P. Caspi. How many times should a program be unfolded for proving invariant properties? Technical Report TR-2004-9, Verimag, 2004.

[38] A. A. Milne. *Winnie the Pooh*. Dutton and Company, 1926.

[39] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[40] M. R. Parker. *A Set Covering Approach to Infeasibility Analysis of Linear Programming Problems and Related Issues*. PhD thesis, University of Colorado at Denver, 1995.

[41] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[42] R. Sebastiani. Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms. Technical Report 0111-22, ITC-IRST, November 2001.

[43] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions. In *40th Design Automation Conference*, pages 425–430, 2003.

[44] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Lecture Notes in Computer Science*, volume 1954. Springer Verlag, 2000.

[45] O. Strichman. On Solving Presburger and Linear Arithmetic with SAT. In *Proceedings of Formal Methods in Computer-Aided Design*, 2002.

[46] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding Separation Formulas with SAT. In *Lecture Notes in Computer Science*, volume 2404, pages 209–222. Springer Verlag, 2002.

[47] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *Lecture Notes in Artificial Intelligence*, volume 2424, 2002.

[48] S. A. Wolfman and D. S. Weld. The LPSAT Engine & Its Application to Resource Planning. In *IJCAI*, pages 310–317, 1999.

[49] L. A. Wolsey. *Integer Programming*. Wiley, 1998.

[50] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiablility Solvers. In *Lecture Notes in Computer Science*, volume 2404, pages 17–36. Springer Verlag, 2002.