Detecting Timing Leaks in Hardware
Semester Thesis

Audrey Lim

WS 06/07

Supervisors:
Boris Köpf, Prof. David Basin

Abstract

Timing leakage occurs when a program's running time depends on the program's secret parameters. This of course can be exploited by attackers to gain information on the secret key material. One possible and straightforward countermeasure for preventing such timing-based attacks is to ensure that the algorithm's execution time is independent of the secret values.
In order to detect timing leaks in clocked hardware implementations we want to make use of the hardware description language GEZEL and the model checker SMV, whereby a translation from GEZEL to SMV is needed.
In this semester thesis we implemented a compiler for translating programs written in GEZEL to the input language of SMV. The compiler gives us the possibility to automatically analyze hardware designs for potential timing leaks.

Contents

Chapter 1

Introduction

RSA, the today most commonly used public key cryptographic algorithm to secure electronic data transfer, is believed to be secure. Its security is based on its considerable mathematical strength, namely the computational infeasibility of factoring large numbers. Hence recovering the private key is equivalent to factoring the modulus used for de- resp. encryption and thus requires an extraordinary amount of computer processing power and time.

Although RSA is currently considered to be secure against direct attacks, one shouldn't overlook the fact that there exist other ways attackers may choose to recover secret key material. While running a program, side channel information, e.g. the running time, the power consumption or the electromagnetic radiation, may leak and give the attacker information on the inner-living of the system. Such (unintended) physical leakage caused by a naive implementation of a secure mathematical algorithm can be crucial in terms of security. Together with the attackers' knowhow it can be sufficient to extract the secret parameters from cryptographic implementations.

The process of attacking a system indirectly without breaking the mathematical algorithm itself is called *side channel attack*. The one we will focus on is the so-called *timing attack*.

Timing attacks exploit the timing variations in cryptographic operations. The amount of time used by a cyptosystem typically differs slightly from one system run to the other. The reason for these varing running times lays in the fact that the time needed to process depends on the input data, and possibly on the secret key material as well. By carefully measuring the running time of an algorithm, attackers may be able to deduce part of or even the entire secret parameters involved in the operation.

Timing side channels are a serious threat to cryptographic algorithms and should be avoided whenever possible. A conceivable and systematic countermeasure for preventing such timing-based attacks is to ensure that the algorithms' running times are independent of the secret parameters. Therefore algorithms should be tested by running them with all possible secret inputs in order to detect potential timing leaks in hardware.

As predicting running times of multi-purpose processors is a very difficult challenge, we will focus on clocked hardware implementations where the used hardware description language is GEZEL.

The problem of finding possible timing leaks could recently be reduced to a search prob-

lem on a product automaton. In order to solve this search problem, we want to make use of the existing model checker SMV and therefore need a translation from the hardware description language GEZEL to the input language of the model checker SMV.

Since it's a cumbersome task to translate each to be tested GEZEL program into its corresponding SMV program by hand, a compiler which does that job would be of great help. The goal of this semester thesis is to write a translator that takes a GEZEL program and compiles it into the corresponding SMV program which then can be checked for possible timing leakage.

The structure of the remaining part of this document looks as follows.
In Chapter 2 we introduce the models and languages. We first define security by introducing equivalence relations. Furthermore we will see that our problem can be reduced to a much simpler search problem. Afterwards we briefly describe the languages GEZEL and SMV. Chapter 3 will give us an insight to the design and implementation of this work. Some tests will be listed in Chapter 4. And finally we will come to the conclusion of this thesis in Chapter 5.

Chapter 2

Models & Languages

In this Chapter we first give the definition of a (deterministic) Mealy machine. We then define two security domains and introduce equivalences that are necessary to define secure information flow. The problem of deciding whether the information flow is secure can be reduced to a reachability problem on a special type of product automaton. The last two sections are dedicated to the languages, GEZEL and SMV, which we will use to detect whether information leakage occurs or not.

2.1 Mealy Machine

A Mealy machine is a finite state machine that produces an output for each transition, where the output is based on the machine's current state as well as on its input.

We will lay our focus on the determinstic case of a Mealy machine. Its formal definition is the following:

Definition *(deterministic Mealy machine).* A deterministic Mealy machine is a 6-tuple $M = (S, \Sigma, \Gamma, \delta, \lambda, s_0)$, where

- S is a finite set of states,
- Σ is the finite input alphabet,
- Γ is the finite output alphabet,
- δ: $S \times \Sigma \to S$ is the transition function,
- λ: $S \times \Sigma \to \Gamma$ is the output function,
- and $s_0 \in S$ is the initial state.

We use the deterministic Mealy machine to model the hardware circuits that are synchronized by a global clock signal. One transition corresponds to one clock cycle, i.e. during each clock cycle input signals are read and output signals are generated.

2.2 Defining Security

A system is considered to be secure if all system runs are indistinguishable even though the system might compute with different secret data. Vice versa, whenever the system shows distinguishable behavior while processing different secrets, information leakage may occur.

In the following we will introduce equivalence relations between inputs, outputs and states, respectively, in order to model to what extent an observer can distinguish system behavior.

To simplify the definition of security we will limit ourselves to a special case. We assume two security domains, *high* and *low*. According to these two domains, each input as well as output signal consists of a high and a low component:

- $\Sigma = \Sigma_H \times \Sigma_L$
- $\Gamma = \Gamma_H \times \Gamma_L$

Our definition of the terms *high* and *low* reads as follows.
We can think of each observer having a domain assigned and being only able to see variables that are within the same domain or lower. High variables therefore are observationally not available for the low observer, considering he can only see variables in the low domain. The high observer, however, is able to see everything (high and low).

For us, the high observer is of no interest. We will focus on the low observer.

Security means, we restrict the flow of information from the high into the low domain. That is, we demand for two inputs with arbitrary high- but same low input-parts, the low outputs to be indistiguishable. The high outputs, on the otherhand, are allowed to be different since by definition they can't be seen by the low observer anyway. Figure 2.1 visualizes the setting.

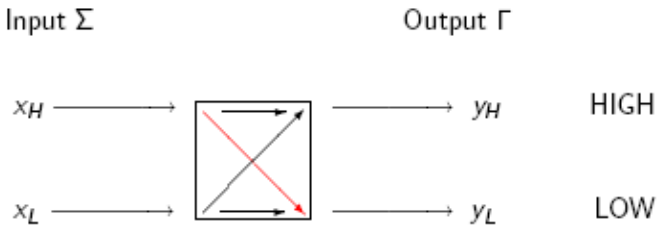

Figure 2.1: A system is secure if there is no information flow from the high into the low domain. On the other hand, if there was such a 'red' information flow, the low observer may gain information on the high input by just looking at the low output which indicates information leakage and thus an insecure system.

Definition *(Low-Equivalence).* Two inputs (resp. outputs) (h_1, l_1), $(h_2, l_2) \in \Sigma_H \times \Sigma_L$ (resp. $\Gamma_H \times \Gamma_L$) are said to be *observationally equivalent* iff $l_1 = l_2$.

A shorter notation for low-equivalence between two inputs (resp. outputs) x_1, x_2 is

$$x_1 =_L x_2$$

where '$=_L$' denotes equivalence on the *low* component.

Two states are said to be observationally equivalent if whenever the system is handed sequences of low-equivalent input it produces sequences of low-equivalent output, independent of the high input. The high input therefore doesn't affect the outcome of the low part in any way. The formal definition is the following.

Definition *(Observational Equivalence R).* Let $M = (S, \Sigma, \Gamma, \delta, \lambda, s_0)$ be a deterministic Mealy machine, where $\Sigma = \Sigma_H \times \Sigma_L$ and $\Gamma = \Gamma_H \times \Gamma_L$.
Two states s_1, s_2 are *observationally equivalent* (s_1 R s_2) iff $\forall$ l $\in \Sigma_L$ $\forall$ h1, h2 $\in \Sigma_H$:
$\delta(s_1,(h_1,l))$ R $\delta(s_2,(h_2,l)) \wedge \lambda(s_1,(h_1,l)) =_L \lambda(s_2,(h_2,l))$.

Definition *(Security).* Let $M = (S, \Sigma, \Gamma, \delta, \lambda, s_0)$ be a deterministic Mealy machine. We say that M has a *secure information flow* (or simply that M is *secure*) iff s_0 R s_0, where R is the relation from the previous definition.

Given a fixed *low* input value, a *secure system* can be fed with all possible *high* inputs and yet the *low* output will always be the same.

A more general definition of security can be found in [1].

2.3 Deciding Security

2.3.1 Reduction: Product Construction

Our next step is to apply the above defined models.

The problem of deciding whether two states are observationally equivalent can be reduced to a reachability problem on a special type of product automaton where every trace corresponds to a pair of traces of the original system.

Thanks to the transitivity of the observational equivalence it suffices to analyze each individual trace of the product automaton in order to determine whether the system is secure.

Definition *(Product Automaton).* Let $M_1 = (S_1, \Sigma_L \times \Sigma_H, \Gamma, \delta_1, \lambda_1, s_{0,1})$ and $M_2 = (S_2, \Sigma_L \times \Sigma_H, \Gamma, \delta_2, \lambda_2, s_{0,2})$ be deterministic Mealy machines.
Then $M_1 \times M_2$ is the automaton $(S_1 \times S_2,\ \Sigma_L \times \Sigma_H \times \Sigma_H,\ \{0,1\},\ \delta',\ \lambda',\ (s_{01}, s_{0,2}))$ where

$\delta' = ((s_1, s_2), (l, h_1, h_2)) = (\delta_1(s_1(l, h_1)), \delta_2(s_2, (l, h_2)))$ and

$\lambda' = ((s_1, s_2), (l, h_1, h_2)) = 1$ if $\lambda_1(s_1, (l, h_1)) =_L \lambda_2(s_2, (l, h_2))$
$= 0$ otherwise.

Theorem. The product automaton reaches a *falsifying* state when its output function λ' outputs 0. Reaching a falsifying state of the product automaton indicates information leakage and therefore an insecure system.

It is proven in [1] that deciding observational equivalence of states is equivalent to determining whether a falsyfying state can be reached in $M_1 \times M_2$.

2.3.2 smv encoding

The product autmaton can be encoded in a few lines of SMV code (see Figure 2.2).

```
MODULE main

VAR
  lo, hi1, hi2 : array(SIZE-1)..0 of boolean;
  sys1 : circuit; sys2 : circuit;

ASSIGN
  sys1.lo_in := lo; sys1.hi_in := hi1;
  sys2.lo_in := lo; sys2.hi_in := hi2;

SPEC !EF(!sys1.done = sys2.done)
```

Figure 2.2: Product construction in SMV.

Suppose `circuit` is a specification of an automaton. We instantiate it twice (as `sys1` and `sys2`) and provide both instances with the same low input `lo`, but different high inputs, namely `h1` and `h2`. In our case of detecting timing leaks, we consider the `done` flags, which signal termination, to represent the low outputs.

If the program `circuit` doesn't suffer timing leakage both instances' `done` flags are set to 1 simultaneously for every combination of `lo`, `h1` and `h2`, and the product automaton won't reach a falsifying state. Conversely, if we reach a state in which one instance's `done` flag is set to 1 before the other instance terminates we have found a falsifying state of the product automaton and as a result timing leakage.

In order to verify whether having reached a falsifing state or not, the SMV product automaton evaluates a CTL (Computational Tree Logic) expression each clock tick. Our formula to be satified, `!EF(!sys1.done = sys2.done)`, states that there exists no path where at some future state the `done` flags are set to different values.[1] The system is secure if the CTL formula is satified throughout the run of the product automaton. Whenever we get to the point where this expression is not satisfied, we reached a falsifying state and SMV generates a counterexample in the form of a trace or sequence of states, if possible.

[1] `EF` stands for Exist and Future.

2.4 The gezel language

GEZEL is a synchronous hardware description language that allows for cycle-true descriptions. Moreover, the output can be mapped to a physical implementation. Therefore the security guarantees obtained using GEZEL implementations are valid as well for the real-world hardware implementations.

A GEZEL program consists of modules where all modules are attached to a single, implicit clock. A module is composed of a controller and a datapath where a controller is always attached to a datapath and each datapath can only have one single controller.
A datapath (dp) is the place where definitions are made: port definition, register/signal definition and instruction definition. The instructions are defined as signal flow graphs (sfg) and each sfg collects a number of expressions.
The signal flow graphs specify only the available instructions, not the schedule. The scheduling of the instructions is done by the controller. There are 3 types of controllers: hardwired, sequencer and fsm. For our purpose we limit ourselves to the fsm type. The fsm controller has two tasks to fulfill in a finite state model, namely instruction sequencing and decision making. The fsm always resides in one of its states. Switching states is realised by state transitions where during a state transition one or more sfgs are selected to be executed.

The following example is a simple program that takes an input parameter `x` and outputs the first multiple of `x` greater than 100 (value stored in `res`) as soon as the `done` flag is set to 1.

```
dp my_dp(in x: ns(8); out res: ns(8); out done: ns(1)) {
 reg y: ns(8);
 sfg init {y = 0; res = 0; done = 0;}
 sfg sig0 {y = y + x;}
 sfg term {res = y; done = 1;}
}

fsm my_controller(my_dp){
 initial s0;
 state s1, end;
 @s0 (init) -> s1;
 @s1 if (y < 100) then (sig0) -> s1;
                  else (term) -> end;
 @end (term) -> end;
}
```

Figure 2.3: GEZEL Example.

2.5 The smv language

We saw that our problem of detecting timing leaks can be reduced to a search problem. Instead of implementing a search procedure in GEZEL by hand, we translate the GEZEL implementations to the input language of the symbolic model checker SMV and use SMV to automate the search on the product automaton from definition *Product Automaton*.

A SMV program consists of one or more modules where a module can be divided roughly into 3 parts.

1. Declarations of the state variables (see `VAR` block)
2. Assignments that define the valid initial states (indicated with `init()`)
3. Assignments that define the transition relation (indicated with `next()`)

The subsequent example is the SMV translation of the previous GEZEL program.

```
MODULE smvProg


VAR

 state : {s0, s1, end};
 x : array 7..0 of boolean;
 res : array 7..0 of boolean;
 done : array 0..0 of boolean;
 y : array 7..0 of boolean;


ASSIGN

 init(state) := s0;
 next(state) :=
  case
   state=s0 : s1;
   state=s1 & y<100 : s1;
   state=s1 & ~(y<100) : end;
   state=end : end;
  esac;
```

```
 next(res) :=
  case
   state=s0 : 0;
   state=s1 & ~(y<100) : y;
   state=end : y;
   1 : res;
  esac;

init(done) := 0;
next(done) :=
 case
  state=s1 & ~(y<100) : 1;
  state=end : 1;
  1 : done;
 esac;

next(y) :=
 case
  state=s0 : 0;
  state=s1 & y<100 : y + x;
  1 : y;
 esac;
```

Figure 2.4: SMV Example.

Chapter 3

Compiler Design & Implementation

As addressed before, the goal of this semester thesis is to build a compiler that translates a GEZEL- into an SMV-program for the reason mentioned earlier. This Chapter discusses the design and implementation of the 'GEZEL-to-SMV' compiler. We first have a glance at the system's architecture. Then we take a look at the class model and the operational description.

3.1 System Architecture

The architecture of the 'GEZEL-to-SMV' compiler can be subdivided into three main components:

- Data Structure:
  In order to store the information of a GEZEL resp. SMV program we need corresponding data structures. Since both languages show a linear pattern we decided the data structures to be abstract syntax lists rather than abstract syntax trees.

- Parser:
  To fill the GEZEL data structure, a parser for GEZEL programs is needed. Its job is to take an input GEZEL program in plain text and store its information into the provided abstract syntax lists.

- Transformer:
  This component transforms one data structure into the other and prints the target code.

We opted for the code to pass two data structures while going through the compiling process. This intermediate step is done to have the translation nicely subdivided into meaningful steps as well as to allow a more straightforward access to the information when bulding the final SMV program. An illustration of the whole procedure can be seen in Figure 3.1.

Figure 3.1: Compiling process. GEZEL code is received as input and gets stored in a data structure designed for GEZEL programs. Instead of directly producing the SMV output, the information first gets restored in a further data structure from which the SMV code is finally read and printed.

The classes needed for the first data structure are within the package *gezelDatastructure*, whereas the classes used for setting up the second data structure are gathered in a package called *smvDatastructure*. More information on these packages and their classes are given in the next section.

3.2 Class Model

We chose an object-oriented approach for the realization of the compiler. For each component discussed in the previous section, a seperate package is provided. An overview of all packages and some of their classes is given in Figure 3.2 in form of a UML-like diagram. Abstract classes are labeled with italic class names (*Variable*) whereas class names in upright font (Transition) represent concrete classes. Arrows symbolize an inheritance relationship and the lines ending in a diamond represent composition.

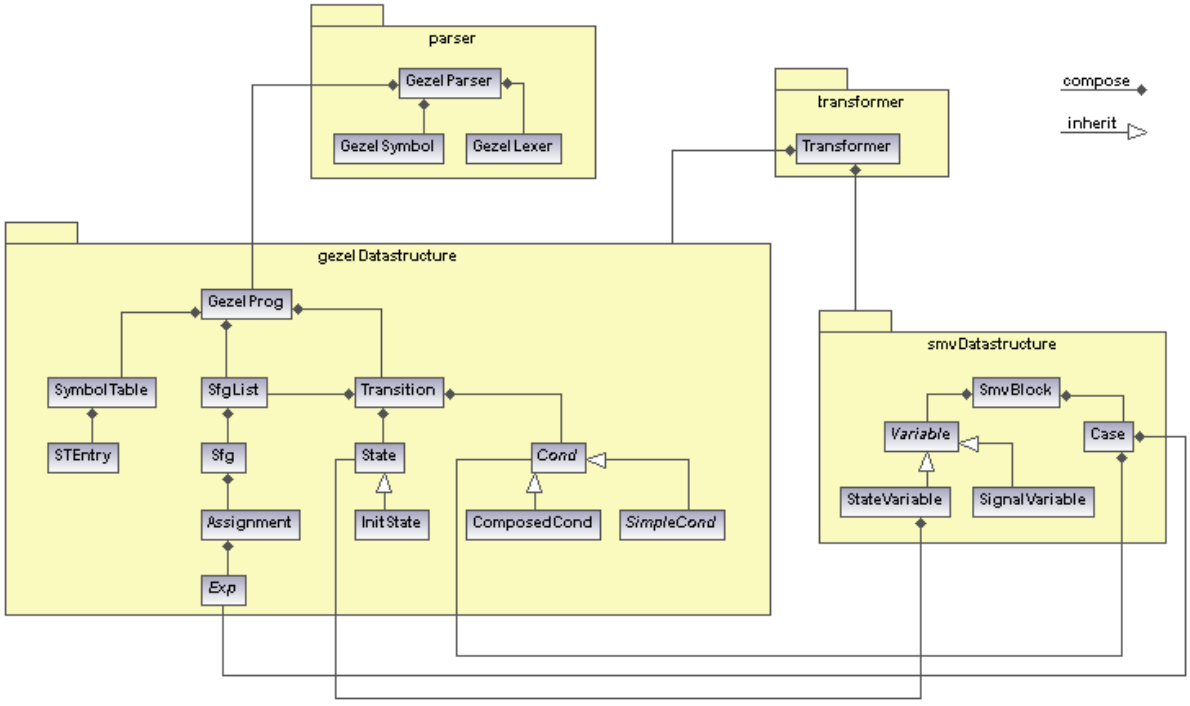

Figure 3.2: Class model. Shows the packages and some of their most important classes.

The following four paragraphs should give a rough idea of the provided packages.

parser package.
This package includes the parser and the lexer. More information on them is given in the next section.

gezelDatastructure package.
We came to the conclusion that the structure of a GEZEL program is best represented by means of lists. This arised from the fact that the GEZEL language consists of three major parts, namely definitions, instructions and transitions, which all show a linear pattern. The root class of this package is called *GezelProg*. It is composed of the three lists called *SymbolTable*, *SfgList* and *TransitionList*. All of them are built by the parser. The *SfgList* and the *TransitionList* are used to store the signal flow graphs (instructions) and the transitions, respectively. By using a *SymbolTable* it's a lot easier to do checks such as wheter an identifier has been used more than once or is being used before being declared. It stores all used identifiers declared in the program together with their 'type' (`dp`, `signal`, `sfg`, `fsm`, `state`, `initial`) and if needed some further information (e.g. number of bits in case of a signal).

transformer package.
While the first data structure stores the information received by the parser, the second data structure is set up by the transformer. The class *Transformer* is the only class within this package and contains two methods. The first one takes as input a GEZEL data structure and outputs the corresponding SMV data structure so that an easy accessible way to traverse and consequently build the SMV program is assured. The second method goes through the data structure built by the tranformer and prints out the SMV code.

smvDatastructure package.
Similar to GEZEL, SMV simply requires a linear data structure. An SMV program can be stored as a list of blocks where each block consists of a variable and its cases. A case corresponds to an assignment and a condition under which it is executed. Structures like *Exp* from the *gezelDatastructure* package are reused instead of implemented again.

3.3 Operational Description

There are 2 phases within building a compiler:

- the analysis phase which analyzes the source code (GEZEL program) and translates it into a suitable data structure
- the synthesis phase which takes the GEZEL data structure transforms it into an SMV data structure and produces the target code (SMV program)

We first discuss the analysis phase where the lexer and the parser come into play.

3.3.1 Analysis Phase

The first step of building the compiler included the construction of the lexer and the parser. Moreover, the design of the GEZEL data structure had to be specified.

The lexer is a program that takes GEZEL code as its input and splits it into tokens with certain values. These tokens are passed to the parser. The parser takes these tokens and builds the GEZEL data structure according to its specified grammar.

In order to generate the lexer and the parser we made use of the tools *JFlex* and *JavaCup*. They are the JAVA versions of *Lex* and *Yacc* which are typical lexer and parser generating tools generating C code. Both of these tools require a grammar file as input. So in order to build the lexer and the parser all we needed to do is to write these two grammar files.

```
...
">>"          {return symbol(MySymbol.RSHIFT};}
"^"           {return symbol(MySymbol.XOR};}
...
"initial"     {return symbol(MySymbol.INITIAL};}
"state"       {return symbol(MySymbol.STATE};}
"@"           {return symbol(MySymbol.AT};}
"->"          {return symbol(MySymbol.TARGET};}
"if"          {return symbol(MySymbol.IF};}
"=="          {return symbol(MySymbol.EQUAL};}
"then"        {return symbol(MySymbol.THEN};}
"else"        {return symbol(MySymbol.ELSE};}
...
```

Figure 3.3: Grammar file excerpt for generating the lexer with JFlex.

```
...
terminal String ID, COMP;
terminal Integer NUMBER;
...
non terminal State state_from_id, state_to_id;
non terminal VarExp lhs_expr;
non terminal SimpleCond cond_expr, condition;
...
gezel_desc ::= datapath fsmcontrol
               {: RESULT = new GezelProg(myTable,mySfgList,myTransList); :};
datapath   ::= DATAPATH dp_id dp_io LBRACE dp_def RBRACE;
dp_id      ::= ID:n
               {: STEntry sym = new STEntry(n, "DP");
                  myTable.insert(sym); :};
...
```

Figure 3.4: Grammar file excerpt for generating the parser with JavaCup.

The grammar file for *JFlex* contains the set of character sequences that may occur in the GEZEL program and defines their mappings to tokens. An extract of our grammar file for *JFlex* is shown in Figure 3.3.

The grammar file for *JavaCup* determines the structure of a GEZEL program. Furthermore, it specifies how the GEZEL data structure is set up. When a certain combination of tokens is encountered, the JAVA code within the corresponding braces is executed and builds up the GEZEL data structure. Figure 3.4 shows an extract of our *JavaCup* grammar file.

Having the data stored in the GEZEL data structure, we can move on to the so-called synthesis phase.

3.3.2 Synthesis Phase

As mentioned before in section 3.1, we decided to have not only one, but two intermediate representations of the data being processed. Not only does it make the compilation more understandable, it also simplifies the process of collecting the needed data by having it stored in an easy accessible way. The transformation from one data structure into the other is done by the transformer.

Let's have a look at the core transformation that has to be done.
Figures 3.5 and 3.6 show simplified excerpts of a GEZEL program and its corresponding SMV code, respectively. The biggest difference between GEZEL and SMV is their description of an automaton. GEZEL describes an automaton in terms of transitions, i.e. each transition of an automaton is declared explicitely. Line 6 in Figure 3.5 illustrates such a transition. In SMV on the other hand there is no explicit description of transitions. Instead it models an automaton by assigning each existing variable in the program a value every clock cycle. The values assigned depend on the automaton's current state and possibly some further conditions. Figure 3.6 shows this for the variable m.

While in GEZEL one focuses on the transitions in an automaton, in SMV one places emphasis on the variables. Thus the main challenge of the transformation is to turn the GEZEL transitions into these SMV blocks of conditional assignments for each existing variable in the program.

```
1: sfg sigA {m = 1;}
2: sfg sigB {m = 2;}
3: sfg sigC {m = 3;}
4:
5: initial s1;
6: @s1 (sigA) → s2;
7: @s3 if (i == 0)
8:       then (sigB) → s4;
9:       else (sigC) → s5;
```

Figure 3.5: GEZEL

```
next(m) :=
   case
     state=s1 : 1;
     state=s3 & i=0: 2;
     state=s3 & ∼(i=0): 3;
   esac;
```

Figure 3.6: SMV

After having built the SMV data structure, the last step consists of traversing it and producing the program in the target language.

Chapter 4

Tests

This Chapter shows two sample GEZEL-to-SMV translations.
In the first example the compiler receives a GEZEL code which does nothing spectacular, but includes a nested `if-then-else` clause. Thereon follows a more specific and topic-related GEZEL program which is taken from [1] and implements a finite field exponentiation algorithm.

```
dp prod(in x_in: ns(4); in i_in: ns(1);
        out done: ns(1); out res: ns(4)) {

reg x,p: ns(4);
reg i: ns(1);

sfg init {x = x_in; i = i_in; p = 0;}
sfg sig0 {p = 1; i = 0;}
sfg sig1 {p = 2;}
sfg sig2 {p = 3;}
sfg term {done = 1; res = p;}
sfg cont {done = 0; res = 0;}

}

fsm ctl_prod(prod) {

initial s1;
state s2,s3,end;

@s1 (init,cont) -> s2;
@s2 if (x[3])
    then if (i==0) then (term) -> end;
         else (sig0,cont) -> s2;
    else if (i==0) then (sig1,cont) -> s3;
         else (sig2,cont) -> s3;
@s3 (term) -> end;
@end (term) -> end;

}
```

Figure 4.1: Input to Test1.

```
MODULE smvProg

VAR

 state : {s1, s2, s3, end};
 x_in : array 3..0 of boolean;
 i_in : array 0..0 of boolean;
 done : array 0..0 of boolean;
 res : array 3..0 of boolean;
 x : array 3..0 of boolean;
 p : array 3..0 of boolean;
 i : array 0..0 of boolean;

ASSIGN

 init(state) := s1;
 next(state) :=
  case
   state=s1 : s2;
   state=s2 & i=0 & x[3] : end;
   state=s2 & ~(i=0) & x[3] : s2;
   state=s2 & i=0 & ~x[3] : s3;
   state=s2 & ~(i=0) & ~x[3] : s3;
   state=s3 : end;
   state=end : end;
  esac;

 init(done) := 0;
 next(done) :=
  case
   state=s2 & i=0 & x[3] : 1;
   state=s3 : 1;
   state=end : 1;
   1 : done;
  esac;

 next(res) :=
  case
   state=s1 : 0;
   state=s2 & i=0 & x[3] : p;
   state=s2 & ~(i=0) & x[3] : 0;
   state=s2 & i=0 & ~x[3] : 0;
   state=s2 & ~(i=0) & ~x[3] : 0;
   state=s3 : p;
   state=end : p;
   1 : res;
  esac;

 next(x) :=
  case
   state=s1 : x_in;
   1 : x;
  esac;

 next(p) :=
  case
   state=s1 : 0;
   state=s2 & ~(i=0) & x[3] : 1;
   state=s2 & i=0 & ~x[3] : 2;
   state=s2 & ~(i=0) & ~x[3] : 3;
   1 : p;
  esac;

 next(i) :=
  case
   state=s1 : i_in;
   state=s2 & ~(i=0) & x[3] : 0;
   1 : i;
  esac;
```

Figure 4.2: Output of Test1.

```
dp prod(in  x_in : ns(8); in a_in : ns(4);
        out done : ns(1); out prod1 : ns(8)) {

reg x : ns(8);
reg a : ns(4);
reg m : ns(8);
reg p,q,s : ns(8);
reg i : ns(3);
reg j : ns(4);

sfg init {
   x = x_in;
   a = a_in;
   m = 29;}

sfg sig0 {p=1;}
sfg sig1 {q=0;}
sfg sig2 {i=4;}
sfg sig3 {j=8;}
sfg sig4 {i=i-1;}
sfg sig5 {j=j-1;}
sfg sig6 {s=p;}
sfg sig7 {p=q;}
sfg sig8 {s=x;}
sfg sig9 {q=q<<1;}
sfg sig10 {s=s<<1;}
sfg sig11 {a=a<<1;}
sfg sig12 {q=q^m;}
sfg sig13 {q=q^p;}

sfg cont {
   done = 0;
   prod1 = 0;}

sfg term {
   done = 1;
   prod1 = x;}

}

fsm ctl_prod(prod) {

initial s1;
```

```
state s2, s3, s4, s5, s6, s8, s9, s10, s11,
      s12, end;

@s1 (init,sig0,sig2,cont) -> s2;

@s2 if (i==0)
    then (term) -> end;
    else (sig1,sig3,sig6,cont) -> s3;

@s3 if (j==0)
    then (sig7,cont) -> s8;
    else(cont) -> s4;

@s4 if (q[7])
    then (sig9,cont) -> s5;
    else (sig9,cont) -> s6;

@s5 (sig12,cont) -> s6;

@s6 if (s[7])
    then (sig5,sig10,sig13,cont) -> s3;
    else (sig5,sig10,cont) -> s3;

@s8 if (a[3])
    then (sig1,sig3,sig8,cont) -> s9;
    else (sig4,sig11,cont) -> s2;

@s9 if (j==0)
    then (sig4,sig7,sig11,cont) -> s2;
    else (cont) -> s10;

@s10 if (q[7])
     then (sig9,cont) -> s11;
     else (sig9,cont) -> s12;

@s11 (sig12,cont) -> s12;

@s12 if (s[7])
     then (sig5,sig10,sig13,cont) -> s9;
     else (sig5,sig10,cont) -> s9;

@end (term) -> end;
}
```

Figure 4.3: Input to Test2: Finite field exponentiation in $F(2^8)$.

```
MODULE smvProg

VAR

 state : {s1, s2, s3, s4, s5, s6, s8, s9,
          s10, s11, s12, end};
 x_in : array 7..0 of boolean;
 a_in : array 3..0 of boolean;
 done : array 0..0 of boolean;
 prod1 : array 7..0 of boolean;
 x : array 7..0 of boolean;
 a : array 3..0 of boolean;
 m : array 7..0 of boolean;
 p : array 7..0 of boolean;
 q : array 7..0 of boolean;
 s : array 7..0 of boolean;
 i : array 2..0 of boolean;
 j : array 3..0 of boolean;
```

Figure 4.4: Output of Test2 (1/2).

```
ASSIGN

 init(state) := s1;
 next(state) :=
  case
   state=s1 : s2;
   state=s2 & i=0 : end;
   state=s2 & ~(i=0) : s3;
   state=s3 & j=0 : s8;
   state=s3 & ~(j=0) : s4;
   state=s4 & q[7] : s5;
   state=s4 & ~q[7] : s6;
   state=s5 : s6;
   state=s6 & s[7] : s3;
   state=s6 & ~s[7] : s3;
   state=s8 & a[3] : s9;
   state=s8 & ~a[3] : s2;
   state=s9 & j=0 : s2;
   state=s9 & ~(j=0) : s10;
   state=s10 & q[7] : s11;
   state=s10 & ~q[7] : s12;
   state=s11 : s12;
   state=s12 & s[7] : s9;
   state=s12 & ~s[7] : s9;
   state=end : end;
  esac;

 init(done) := 0;
 next(done) :=
  case
   state=s2 & i=0 : 1;
   state=end : 1;
   1 : done;
  esac;

 next(prod1) :=
  case
   state=s1 : 0;
   state=s2 & i=0 : x;
   state=s2 & ~(i=0) : 0;
   state=s3 & j=0 : 0;
   state=s3 & ~(j=0) : 0;
   state=s4 & q[7] : 0;
   state=s4 & ~q[7] : 0;
   state=s5 : 0;
   state=s6 & s[7] : 0;
   state=s6 & ~s[7] : 0;
   state=s8 & a[3] : 0;
   state=s8 & ~a[3] : 0;
   state=s9 & j=0 : 0;
   state=s9 & ~(j=0) : 0;
   state=s10 & q[7] : 0;
   state=s10 & ~q[7] : 0;
   state=s11 : 0;
   state=s12 & s[7] : 0;
   state=s12 & ~s[7] : 0;
   state=end : x;
   1 : prod1;
  esac;

 next(x) :=
  case
   state=s1 : x_in;
   1 : x;
  esac;
```

```
 next(a) :=
  case
   state=s1 : a_in;
   state=s8 & ~a[3] : a << 1;
   state=s9 & j=0 : a << 1;
   1 : a;
  esac;

 next(m) :=
  case
   state=s1 : 29;
   1 : m;
  esac;

 next(p) :=
  case
   state=s1 : 1;
   state=s3 & j=0 : q;
   state=s9 & j=0 : q;
   1 : p;
  esac;

 next(q) :=
  case
   state=s2 & ~(i=0) : 0;
   state=s4 & q[7] : q << 1;
   state=s4 & ~q[7] : q << 1;
   state=s5 : q ^ m;
   state=s6 & s[7] : q ^ p;
   state=s8 & a[3] : 0;
   state=s10 & q[7] : q << 1;
   state=s10 & ~q[7] : q << 1;
   state=s11 : q ^ m;
   state=s12 & s[7] : q ^ p;
   1 : q;
  esac;

 next(s) :=
  case
   state=s2 & ~(i=0) : p;
   state=s6 & s[7] : s << 1;
   state=s6 & ~s[7] : s << 1;
   state=s8 & a[3] : x;
   state=s12 & s[7] : s << 1;
   state=s12 & ~s[7] : s << 1;
   1 : s;
  esac;

 next(i) :=
  case
   state=s1 : 4;
   state=s8 & ~a[3] : i - 1;
   state=s9 & j=0 : i - 1;
   1 : i;
  esac;

 next(j) :=
  case
   state=s2 & ~(i=0) : 8;
   state=s6 & s[7] : j - 1;
   state=s6 & ~s[7] : j - 1;
   state=s8 & a[3] : 8;
   state=s12 & s[7] : j - 1;
   state=s12 & ~s[7] : j - 1;
   1 : j;
  esac;
```

Figure 4.5: Output of Test2: Finite field exponentiation in $F(2^8)$ (2/2).

Chapter 5

Conclusion

5.1 Results

The core objective of this semester thesis was to design and implement compilers from the hardware description language GEZEL to the input languages of the model checkers SMV and SPIN. Starting this semester thesis without background in compiler design required some extra time which led to the decision to leave out the 'GEZEL-to-SPIN' compiler and instead focus entirely on the 'GEZEL-to-SMV' compiler.

However, part of the code of the 'GEZEL-to-SMV' compiler can be reused for building the 'GEZEL-to-SPIN' compiler. That is, the parser as well as the GEZEL data structure can be taken one-to-one from the 'GEZEL-to-SMV' compiler implementation. Code that has to be written from scratch includes the data structure of SPIN as well as the transformer.

The compiler can be run using the shell script that comes along with the program. Its usage is simple. It takes as input two parameters, the source file and the target file. For further information consult the included `readme` file.

5.2 Lesson learned

Having visited rather theoretical classes in the past, I hardly ever got the chance to implement programs that were longer than a few lines. This semester thesis gave me the opportunity to get some experience in programming and it even provided an insight into a world that I haven't been familiar with before at all, namely compiler design. With no background knowledge on building compilers it took me some time to get acquainted with the topic. Nevertheless, it was a pleasure and surely interesting to see into a totally different subject and get to learn something that I wouldn't have learned otherwise.

5.3 Future Work

For reasons mentioned earlier, we could not include all ideas we had in mind.

Further extensions to the currently existing work are:

- At present the compiler is able to interpret a subset of the GEZEL language. The next step therefore is to enhance the vocabulary of GEZEL expressions which will be accepted and processed by the compiler so that a wider range of programs can be translated.
- Implementation of a second compiler from the hardware description language GEZEL to the input language of the model checker SPIN by reusing part of the code from the GEZEL-to-SMV compiler implementation.

Bibliography

[1] Boris Köpf and David Basin. *Timing-Sensitive Information Flow Analysis for Synchronous Systems.* ETH Zürich, Switzerland, 2006.

[2] Gerwin Klein. *JFlex: The Fast Lexical Analyser Generator.* `http://www.jflex.de/manual.pdf`, 2004.

[3] Scott E. Hudson. *CUP: LALR Parser Generator For Java.* `http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`, 1999.

[4] Patrick Schaumont. *GEZEL language Reference 2.0.* `http://rijndael.ece.vt.edu/gezel2/index.php/GEZEL_Language_Reference`, 2005.

[5] K. L. McMillan. *The SMV language.* Berkeley, CA, USA, 1999.