Dependency Analysis of Synchronous Programming Languages

Masterarbeit

Yu Bai

Betreuer:
Prof. Dr. Klaus Schneider
Dr. Jens Brandt

AG Eingebettete Systeme
Fachbereich Informatik
TU Kaiserslautern

30th November 2010
Erklärung

Hiermit erkläre ich, dass die vorliegende Masterarbeit wurde von mir selbstständig verfasst. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Kaiserslautern, 30th November 2010

Yu Bai
Acknowledgement

This thesis work was carried out in the AG Eingebettete Systeme which I gratefully acknowledge.

I would like to take this opportunity to express my gratitude to the persons who contributed to the completion of this thesis. I want to give a special thanks to my supervisors professor Schneider and doctor Brandt, for all their patience, advice, and support. Professor Schneider provided me the opportunity to work on this very interesting subject. We went through lots of inspiring discussions at the beginning of my work. Doctor Brandt paid constant energy and patience during the whole process to help me get through every tough problem I had and fixed every tiny little mistake in my thesis. I am thankful for the opportunity to learn from them and I sincerely appreciate their guidance.

I would also like to thank Mike, Daniel, Manuel and every member in the workgroup for creating a comfortable and fertile atmosphere for me to work in.
# Contents

1 Introduction ................................................................. 1  
  1.1 Motivation ............................................................ 1  
  1.2 Contribution .......................................................... 2  
  1.3 Overview of the Thesis ............................................... 2  

2 Fundamentals ............................................................... 3  
  2.1 Synchronous Programming Languages ............................. 3  
    2.1.1 The Synchronous Model of Computation ...................... 3  
    2.1.2 Quartz and AIF ................................................ 4  
    2.1.3 Compilation Problems ........................................ 7  
  2.2 Satisfiability Modulo Theories ..................................... 13  
    2.2.1 Satisfiability Modulo Theories Essentials .................. 13  
    2.2.2 The SMT solver Z3 ............................................ 15  
  2.3 Data Dependency Analysis for sequential programs ............ 17  

3 Optimization ............................................................... 23  
  3.1 Dependency Analysis of EFSM ....................................... 23  
  3.2 Generation of EFSM ................................................ 28  
    3.2.1 Generation of States and Transitions ....................... 30  
    3.2.2 Generation of Attached Guarded Actions ................... 36  
  3.3 Optimizations ....................................................... 37  
    3.3.1 Constant Propagation ......................................... 38  
    3.3.2 Invariant Inference .......................................... 45  
    3.3.3 Identification of Passive Code ............................ 58  
    3.3.4 Array Optimization ......................................... 61  
    3.3.5 Correctness of the Optimization ........................... 64  

4 Experimental Results .................................................... 65  
  4.1 Small Experiments ................................................ 65  
  4.2 Studies ............................................................... 74  

5 Summary ........................................................................ 77  
  5.1 Conclusion ............................................................. 77  
  5.2 Future Work ............................................................ 78  

A Supported Types of Quartz Expressions ................................ 79
1 Introduction

1.1 Motivation

Reactive systems [23, 37] by nature are demanded to fulfill strict requirements on correctness and efficiency, which makes the design and implementation not an easy work. For the past decades, synchronous languages like Esterel [10], Lustre [14, 24] have been proven a suitable method. The underlying reason is that the synchronous language and reactive system both share the same computation model, namely the synchronous model of computation. This gives not only the ease of convenient system description (regardless from HW/SW), but also the comfort of WCTE and formal verification of the system properties. Previously many works have been done successfully dealing with theoretical problems like causality analysis and schizophrenic that challenge the compilation of synchronous programs at most. Nowadays the research is focusing more about efficiency issues. For lifting the performance of the system to fulfill the strict requirements, the synchronous programs usually need to be optimized. Optimized programs usually have smaller size and have less computation. As a result, the optimized programs cost less storage, and reduce the time of reaction.

One important basis of optimization is to detect the data dependence of the program. By analyzing the data dependency relationships in the program, more information of the control flow and data flow of the program can be derived and used in improving the efficiency of the program’s performance. Another important usage of data dependency analysis could be the analysis of the correctness of the program. In principle, data dependency analysis helps us to know more about the program. The more knowledge we know about the program, the more safety and better performance we can assure.

While basic data dependency relationships can be detected by using simple syntactical methods, there are more underlying data dependency relationships that can be detected only by taking care of the semantical aspects of the program. By using Satisfiability Modulo Theory (SMT) based methods, many of these data dependency relationships can be discovered. This is exactly what this thesis trying to accomplish: to exploit the achievement in SMT and use it for better analysis of the data dependency in synchronous languages. The final goal of data dependency analysis in this thesis is the optimization of synchronous programs.
1.2 Contribution

The task in this thesis is to give a thorough analysis of the data dependency relationships for synchronous language, in particular Quartz programs [38, 39]. For the sake of simplicity and generalization the analysis is not directly based on Quartz programs, but an intermediate format called Averest-Intermediate-Format (AIF). The essential of AIF is a set of guarded actions. First the control flow and data flow of the AIF structure are extracted by translating the AIF structure into a semantically equivalent Extended Finite State Machine (EFSM). Then the data dependency analysis is done based on the EFSM in the form of satisfiability checking by utilizing an SMT solver. The result of the data dependency analysis is then used in various levels of optimization methods.

The optimization of synchronous programs here mainly focuses on two aspects: Dead-Code Elimination and Passive-Code Elimination. Based on this central idea, four optimization methods are developed. Finally some special considerations of arrays are also covered.

1.3 Overview of the Thesis

In Chapter 2 the prerequisite knowledge is introduced. Section 2.1 gives an introduction to synchronous languages, including the synchronous model of computation and the synchronous language we used in this thesis – Quartz. Two issues are discussed in more details – causality and schizophrenia. These two problems were main challenges for the compilation of synchronous programs. The solving of schizophrenia problem further leads to the generation of separated control flow / data flow, which is organized by the AIF structure as the input of the Extended Finite State Machine (EFSM) generation. The final section of this chapter gives an overview of the dependency analysis techniques used for classical compilers. Chapter 3 presents the main contribution of the thesis. The first section (section 3.1) introduces the dependency analysis of synchronous languages. Section 3.2 describes the generation of the EFSM based on the control flow and data flow of the program. This EFSM is then used as input for the optimization techniques which are developed in section 3.3. In section 3.3 the SMT techniques are utilized for the dependency analysis. The results are utilized directly by different optimization techniques: Inside-State Constant Propagation, Cross-State Constant Propagation, Invariant-Inference and Identification of Passive Code. Finally chapter 4 gives the benchmark of the optimization techniques and draws some conclusions.
2 Fundamentals

2.1 Synchronous Programming Languages

2.1.1 The Synchronous Model of Computation

A reactive system \[23, 37\] is the type of system that periodically reacts to the environment with stringent time requirement. The specifically demanded features include deterministic, concurrency and high reliability. In the past decades synchronous languages like Esterel and Lustre have been proposed and successfully implemented for modeling reaction systems. The underlying reason is that synchronous languages capture the essential principle of reactive systems – the synchronous model of computation.

The synchronous model of computation \[8, 23\] is the principle of synchronous programming languages. In the synchronous model, a program periodically samples inputs. Within each period, the system reacts according to the input. A reaction consists of finitely many computations, which are considered atomic and takes no time. An implication is that there are no data dependent loops in a reaction step. This is usually called perfect synchrony. The computations within each reaction are sometimes considered as micro-steps while each reaction period a macro-step. In this model, the concept of discrete logical time is used for modeling time, while within a discrete logical unit of time the computation takes no time. This is, of course, an ideal model, and in practice “no time” can be referred to as before the next reaction.

It is convenient to treat a synchronous program as an automaton in which the states refer to the locations of the control flow, and each reaction corresponds to a transition of the automaton. As automatons are synchronous when composed in parallel, a transition of the product actually consists of all the simultaneous transitions of each automaton. An explicit result is that the synchronous product of automatons preserves determinism, which is a desired property in system design. Another good reason for using automaton as the model of synchronous program is because of the FSM based mature verification techniques like Symbolic Model Checking \[16, 37\], which could provide guarantee for the demanded system properties.

In the later parts of the thesis, it is shown that the synchronous programs are translated into equivalent sets of guarded actions, which are then translated to extended
finite state machines (EFSMs) based on the idea described above. Actually the whole data dependency analysis methods and optimizations are based on the EFSM structure.

### 2.1.2 Quartz and AIF

This section is dedicated to the synchronous programming language Quartz [38][39], which is the programming language used in this thesis for the program examples. Quartz is a variant of Esterel with extended features like delayed assignments, delayed emissions, asynchronous concurrency, clock refinement etc. Its core statements are defined below:

**Definition 1** (Core Statements of Quartz). The set of core statements of Quartz is the smallest set that satisfies the following rules, provided that \( S_1 \) and \( S_2 \) are also basic statements of Quartz, \( l \) is a location variable, \( x \) is an event variable, \( y \) is a state variable, \( \delta \) is a Boolean expression, and \( \alpha \) a type:

- nothing
- emit and emit next (x) (immediate/delayed emission)
- \( y := \tau \) and next(y) := \( \tau \) (immediate/delayed assignments)
- \( l:\text{pause} \) (consumption of time)
- if \( \delta \) then \( S_1 \) else \( S_2 \) (conditional)
- \( S_1 ; S_2 \) (sequential composition)
- \( S_1 || S_2 \) (synchronous concurrency)
- \( S_1 ||| S_2 \) (asynchronous concurrency)
- choose \( S_1 \mid S_2 \) (nondeterministic choice)
- do \( S \) while \( \delta \) (iteration)
- suspend \( S \) when \( \delta \) (suspension)
- weak suspend \( S \) when \( \delta \) (weak suspension)
- abort \( S \) when \( \delta \) (abortion)
- weak abort \( S \) when \( \delta \) (weak abortion)
- \( \alpha; S \) (local variable)
- during \( S \) holds \( \delta \) (invariant assertion)

The data types of Quartz can be atomic data types like *boolean*, *integer*, etc. or composite data types like arrays and tuples. Besides the data type, the variable declarations also consists of the specification of the information flow and the storage type. The information flow of a variable falls in to the four types, which are: *input*, *control*, *output* and *local variables*. Together they are called *data variables* in the thesis. The storage type of Quartz variables can be either *event* or *memorized*. While the information flow of a variable is trivial, the storage type is worth some more words. An *event variable* can be emitted either in the current macro-step, or can be emitted from the previous macro step using a delayed emission. In either case, an event variable
2.1 Synchronous Programming Languages

does not store its value. In contrast to event variables, *memorized variables* are those store their values until they are assigned new values. A good analogy in hardware is that the memorized variables can be seen as registers, while the event variables can be seen as signals.

While the values of location variables are implicitly set by the control flow of the program, the values of the data variables are explicitly assigned by using assignments. There are two types of assignments: *immediate* and *delayed* assignments. An immediate assignment assigns a value to the corresponding variable within the same macro step while a delayed assignment assigns a value to the variable in the next macro-step. One thing to be noticed here is that the assigned value is always evaluated within the same macro-step. As a result of the existence of the memorized variables and delayed assignments, there are also possible data dependencies between the states of the EFSM. For more semantical information of Quartz it is suggested that the reader reference [38].

A statement $S$ of Quartz can be instantaneous, which means that it terminates at the exact time point when it starts. The instantaneous statements refer to the instant computations within a reaction step (i.e. a macro-step). Although these instantaneous statements do not take any time, it does not mean that they can be executed simultaneously. As there might be data dependencies between these statements, they have to be executed w.r.t. the underlying data dependency. Otherwise the result of the computation would be wrong. These steps of computation that exist within a single reaction step are called micro-steps. The other type of statements consumes time and would stop at a particular position of the control flow after its termination of execution. Once a program reaches a particular position of control flow by executing such a statement, the program ends its current reaction to the environment and waits for the next reaction to begin the next macro-step. The only basic statement that causes the control flow to stop is the *pause* statement. For this reason, pause statements are labeled with unique boolean location variables. A location variable $l$ is *true* if only the control flow is currently stopped at location $l$. As introduced before, a finite program is labeled with finitely many location variables. Because of concurrency, the program may stop at multiple location variables at the same time. The finitely many possible combinations of the boolean assignments of the location variables form the state space of the EFSM of the Quartz program.

A Quartz program is consisted of *modules*, with each module having the syntax below:

```plaintext
module Name($decl_1$, ..., $decl_n$) {
  BodyStmt
}
```

where each declaration of variables $decl_i$ has the syntax:

*StorageClass type* $name_i[n_1]...[n_{d_i}], ... , name_m[n_1]...[n_{d_m}]$
where “StorageClass” can be either event or memorized with “type” the data type of
the declared variable (with possibly specified array dimensions). The names of con-
trollable input variables and output variables have to be prefixed with the symbols “?”
and “!” respectively. The following shows a simple Quartz program implementing
a simple FSM:

```quartz
module ABRO ( event bool ?a , event bool ?b,
   event bool ?r , event bool !o) {
   loop
   abort {
   await (a) ; || await (b) ;
   emit o ;
   await ( r ) ;
   ) when ( r ) ;
   }
```

In the module named ABRO, there are three input variables a, b, r which are events. The variable o is the only output of the module, which is a boolean event. The Mealy
machine of the program is given in Figure 2.1. The program waits on event a and b
congruently. Any occurrence of a and b drives the program to a next state which then
waits for the other not appeared event. An output o is omitted after both a and b are
emitted. During this process, an occurrence of event r resets the state of the program
to its starting state regardless of the emission of the output o.

Figure 2.1: State machine of the program ABRO
2.1.3 Compilation Problems

As described above, perfect synchrony is a very good property of synchronous computation model. It provides a deterministic behavior, a convenient model, synthesis of synchronous circuits and formal verification. Anyway these advantages does come with prices – it brings some challenges to the compilation of synchronous programs, i.e. the compilation of schizophrenic programs and cyclic causal programs. In the following some words will be spend on both of them, because the first problem give rise to the compilation of synchronous programs which leads to the derivation of the separated control flow / data flow of the program. The separated control flow / data flow are the input of the EFSM generation and the data dependency analysis. The solving of the second problem contributes to the correctness of the compilation of synchronous programs which is taken as a precondition in this thesis. In the following each of the problems is reviewed. This would be a good start to understand the principles of the techniques used for the compilation of synchronous programs.

Causality Analysis

Causality analysis has its origin in cyclic circuit analysis. Kautz and Rivest [34] have proven that by allowing cycles in a combinatorial circuit, the circuit would be more space-economic than any acyclic circuit. A reasonable example is shown as follows:

\[
\{ z = \text{if } (c) \text{ then } \text{shift} \ (\text{add} \ (a, b), d) \\
\quad \text{else } \text{add} \ (\text{shift} \ (a, d), b) \}
\]

For an acyclic implementation, the component “shift” and “add” would have to be duplicated, which is definitely inefficient.

While this gives an important reason for cycles to stay in the design space of circuits, it also brings a problem – as cycles may exist in a circuit, there are possibilities that some of the outputs depend on themselves. These cycles make the circuit not combinatorial anymore and may introduce indeterministic behaviors to the circuit. For example [38], for the given formula: \( Y = \neg (X \lor Y) \), the truth table of the output \( Y \) is shown as follows:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Y(output)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
When \( X = 0, Y = 1 \), the output of \( Y \) is 0. This changes the state of the input to \( X = 0, Y = 0 \) which will produce the output \( Y = 1 \). This again will turn the state back to the original one. This oscillation is caused by the cyclic dependency of \( Y \) as both output and input. It is clear that while some cycles may be safe, some other cycles may lead to problems in a circuit \([20]\). For solving the problem, one can simply forbid the appearance of cycles for being conservative. A more preferable way is to allow the harmless cycles while forbidding the harmful ones. For doing this, we must figure out a way to tell the harmless cycles from those are not.

Perfect synchrony is an abstract way of seeing the synchronous property of synchronous circuits. As a result, in a synchronous programming language like Quartz, there are similar causal problems. As it is shown in the analysis of cyclic circuits, the existence of cycle doesn’t necessarily mean the existence of cyclic dependencies. This indicates that causality analysis of synchronous programs is not a syntactical issue, but a semantically issue. In \([12]\) the programs of different levels of causally correctness are categorized as follows:

**Reactive programs**: A synchronous program \( P \) is reactive if for each particular input there’s at least an output that leads to a consistent execution of the program. This means that there should be no input exists that can lead to an undefined output. For example:

```plaintext
Module P1(bool ?i, bool !x){
    if (x) emit x; else nothing;
}
```

In this program, \( i \) is the input and \( x \) is the output. This piece of program is reactive in the way that \( x \) could be either true or false, no matter what value \( i \) is. If \( x \) is true, then the condition is fulfilled and \( x \) would be emitted, which is consistent with the fact that \( x \) is true. Similarly for the case \( x = false \). Therefore, both cases of the input “\( x \)” lead to a consistent execution of the program.

**Deterministic Programs**: a synchronous program \( P \) is deterministic if for an input there is at most one output that leads to a consistent execution of the program. For example:

```plaintext
Module P2(bool x, bool y)
    if (x) {
        if (y) {emit x; emit y;}
    }
}
```

In this program, both \( x \) and \( y \) are inputs and outputs. For both \( x \) and \( y \) are true, they will be emitted. This leads to a consistent execution of the program. Otherwise for \( x \) and \( y \) both are false, neither \( x \) nor \( y \) would be emitted which also leads to a consistent
execution. For \( x = false \) and \( y = true \) or \( x = true \) and \( y = false \) there’s no consistent execution, since either both \( x \) and \( y \) would be emitted or both of them are not emitted, but not the case that one of them is emitted while the other is not.

**Logically Correct Programs**: a synchronous program \( P \) is **logically correct** if for an input there is exactly one output that leads to the consistent execution of the program, i.e. it is both reactive and deterministic. For example:

```plaintext
class P3(bool x) {
    if(x) emit x;
}
```

By examining every possible input value of \( x \), it is clear that for both \( x = true \) and \( x = false \) there is a consistent execution of the program respectively.

**Constructive Programs**: a synchronous program \( P \) is **constructive** if the output of the program can be constructively computed without speculation (case distinctions). For a more formal definition, the readers are suggested to refer to [12, 9]. The following example shows a constructive program:

```plaintext
class P4 (event &o) {
    if(o) nothing;
    emit o;
}
```

Notice that in this program, the sentence “emit o” is to be executed definitely. This makes the guard \( true \) which turns out to be consistent with the emission of \( o \). This consistent execution is by construction, rather than by guessing. On the other hand, consider the following program:

```plaintext
class P5 {
    if(o1) emit o1;
    if(o1) {
        if(o2) nothing; else emit o2;
    }
}
```

This program has neither input nor output. The only coherent assignment for \( o1 \) and \( o2 \) is for both of them to take \( false \). Anyway the logically correctness of the program is by guessing all possible assignments of the two variables. Such a program is logically correct but not constructive. It is called **self-satisfying**. Figure 2.2 shows an overview of the relationships among all kinds of programs discussed above.

In [29] Malik presented a first solution detecting non-combinatorial cyclic circuits. From then on many efforts have been put into solving the causality problems. The
The basic idea of Malik’s solution is to extend the boolean operations to a ternary extension with an additional value *Unknown*. Then by a fixpoint computation of the output of the circuit for each possible boolean input, one checks if there exists an output that is *Unknown*. If so, it means that the output doesn’t only depend on the inputs, but also the output itself. A dual rail coding is used for the implementation of the computation.

This solution is adapted and generalized by many following works. In particular, Berry [9] strengthened and formulated the cyclic combinatorial circuits in the sense of constructivity which is exactly the concept introduced before. Schneider et.al. [40] proved that for the different possible ternary extensions of a particular boolean operator, there always exists a maximal ternary extension in the sense that it can detect the most constructive circuits.

Symbolic algorithms for checking constructivity [9] have been implemented based on the solution described above utilizing BDDs [13]. In [32] the authors argued that symbolic algorithms are inefficient and difficult to be extended to non-Boolean data types that are used by high-level languages. They further proposed solutions that formulate constructivity as a satisfiability question and solve it by using SAT solvers. Similar works have also been done in [15, 6].

In [19] Edward managed to detect the combinatorial partial circuits of a given circuit by finding those input values that produce completely-defined gate outputs. These parts are then merged to form the complete combinatorial circuit of the original circuit.

As described above, the causality analysis of synchronous programs actually runs a process of cyclic dependency detection for the written variables of the program. Although this topic falls into the scope of the thesis, because of the limited size of the thesis it is not discussed here and rather used as an assumption. *The dependency analysis in this thesis is based on causal correct synchronous programs.*
Schizophrenia Problems

Schizophrenia problems refer to the creation of reincarnations of a local variable when the scope of the declaration of that local variable is left and re-entered during the same macro-step. In the synchronous language like Quartz it can be caused by a strong abortion statement with local declarations nested within a loop structure. For example [12]:

```plaintext
module Test ( event bool ?i , event bool !a,
              event bool !b)
{
  loop {
    bool c;
    abort {
      emit a;
      l:pause;
      emit b;
      emit c;
    } when(i);
  }
}
```

When the program is executed for the first time, the abort statement is entered and the emission of \(a\) is executed. If at this moment \(i\) doesn’t hold, because of the semantics of strong abortion the control flow would leave the abort statement. However as the abortion statement is inside the loop body, it will be re-entered and the emission of \(a\) will be re-executed. This means that \(a\) would always be emitted no matter when \(i\) holds. Moreover, consider the local variable \(c\). When the program is started for the first time, \(c\) is declared without emission. By the reaction of absence \(c\) would be set to \(false\) by default. Now assume the control flow is at location \(l\). After the pause \(c\) would be emitted. But at the same macro-step the loop would be re-entered again, which will cause the re-declaration of variable \(c\), again! By the analysis before we know that this newly declared variable \(c\) should retain its value \(false\), while the original \(c\) now has a value \(true\)! This means that we must duplicate the variable \(c\) and distinguish the original one from the reincarnation.

A deeper look into the problem reveals that we have to distinguish the cases when the loop is first entered and when the loop is left and re-entered since the reincarnation problem only occurs in the latter case. It means that we have two possible semantics for the same program based on different conditions. Moreover, it is also necessary to take micro-steps into consideration, since the different reincarnations of a particular local variable are due to the different micro-steps within one macro-step. Based on these facts, a compilation strategy is proposed in [42] [39] by Schneider et.al. In their papers, the compilation of the first case is referred to \(Surface\) compilation, while the latter case referred to \(Depth\) compilation. To distinguish different incarnations of a
local variable, duplications must be done. An observation is that the reincarnations of a local variable can only happen in the surface of a program (since the loop is re-entered and only the surface of the loop can be executed). This means that we only have to duplicate the declared variables in the surface.

Based on the separated compilation, two sets of control flow / data flow can be derived for the surface and depth of a synchronous program respectively. The control flow and data flow are formulated in the form of guarded actions. A guarded action has the form \((p, \text{act})\) where \(p\) is a condition and \(\text{act}\) is an assignment of a variable. For the control flow, the variable should be a location variable of the program. For the data flow, the variable should be one of those input / output variables or local variables. In the Averest compiler for Quartz, the separated control flow / data flow are organized with the information of the declared variables of the program and put into a structure called Averest Interchange Format (AIF). This data structure actually contains more information like the abbreviations of common sub-expressions, assertions, etc. But for this thesis, the relevant information is the control flow and data flow with the declared variables of the program. From now on we call the control flow / data flow together as guarded actions for simplicity.

For a particular Quartz program, Averest can compile the program into a structure called AIFSystem, while for sub-modules of the program Averest compile them into AIFModules. These AIFModules are then linked to form an AIFSystem to represent the final result of the compilation of the whole program. The EFSM generation and data dependency analysis are all based on the AIFSystem structure, which has the structure shown in Figure 2.3. While the shaded boxes are considered in the thesis, the rest are omitted. The entries Interface and Locals contain the declarations of the input / output variables and the local variables. The entries Init and Main contain the guarded actions for the surface and depth respectively. The AIFSystem file is the input of the EFSM generation.

Write Conflicts

Besides the causality problem and schizophrenia, another problem can occur in the context of perfect synchrony: write conflicts. Since in a macro-step, the computations are done concurrently, there might be the case that one variable is assigned to different values at the same time. For example:

```plaintext
module P5(bool o) {
    o = true;
    o = false;
}
```
In this program, two assignments are executed within one macro-step. This leads to a logical conflict of the state of variable $o$ – $o$ can not be true and false at the same time. In the following, it is assumed that the programs have no write conflicts.

2.2 Satisfiability Modulo Theories

In this thesis, data dependency analysis is formulated in the form of satisfiability checking in decidable theories. The task of satisfiability checking is performed by an SMT solver. This section will introduce the Satisfiability Modulo Theory (SMT) and the SMT solver Z3, which is used in the implementation of the satisfiability checking.

2.2.1 Satisfiability Modulo Theories Essentials

As defined in [45], an SMT instance is a formula in first-order logic where some functions and predicate symbols have additional interpretations by a given theory, and SMT is the problem of determining whether such a formula is satisfiable.

**Definition 2** (SMT instance). A **theory-specific predicate** is an expression in theory $T$, such that it evaluates to a boolean value (true or false) under $T$ when a $T$-value
is assigned to each of the variables in the expression. A Satisfiability Modulo Theory (SMT) instance on a theory $T$ is defined recursively as following:

1. A predicate on theory $T$ or a boolean variable is an SMT instance.

2. The logic $\land$, $\lor$ of two SMT instances or the logic not of an SMT instance is an SMT instance.

The “M” of SMT stands for modulo, which comes from the fact that SMT is the problem of determining satisfiability of formulas modulo background theories [17]. A background theory interprets and constraints the non-boolean symbols. For example, a typical background theory can be the theory of real numbers for linear arithmetic formula like: $x + 1 < y$. Here $x$ and $y$ have the interpretation of real numbers.

SMT solvers answer questions of satisfiability by decision procedures [17, 36]. Decision procedures tell whether a formula is inconsistent, satisfiable, or valid, or whether one formula is a consequence of others. A formula is satisfiable if there exists at least one interpretation of the variables (a model) such that the formula holds. If the formula holds for all models, we say the formula is valid. Otherwise, if the formula has no structure that can make it hold, the formula is unsatisfiable or inconsistent. Decision procedures may use heuristics for speed, but must always give the correct answer and terminate (i.e., must be sound and complete). For example, if we ask: “Does $4 \times x = 2$ follow from $x \leq y$, $x \leq 1 - y$, and $2 \times x \geq 1$ when the variables range over the real numbers?” The SMT solver must give us a correct yes or no answer.

While most of the common SMT approaches support only decidable theories, many realistic systems can only be modeled by means of non-linear arithmetics over the real numbers, which in general belong to undecidable theories. Thus, it is important to make sure that the problem we face can be formulated in a supported decidable theory before we put it into our SMT solver. For example the theory of real closed fields, and thus the full first order theory of the real numbers are decidable using quantifier elimination (due to Alfred Tarski [21]). The first order theory of the natural numbers with addition (but not multiplication) is also decidable. Since multiplication by constants can be implemented as nested additions, the arithmetic in many computer programs can be expressed using Presburger arithmetic [21], resulting in decidable formulae. Many useful theories are decidable [36]:

- uninterpreted functions
- linear integer and real arithmetic
- array theory
- fixed-size bit-vectors
- quantifiers
- ...
An SMT solver can be seen as an extension in the capability of problem solving in the sense of an SAT solver. While an SAT solver deals with satisfiability over propositional logic are decidable, an SMT solver concentrates more over the decidable theories introduced above. Thus, an SMT solver generalizes SAT solving by adding the ability to handle arithmetic and other decidable theories. Anything a SAT solver can do, an SMT solver can do better. The extension of SMT solvers over SAT solvers is in the sense that it supports predicates. The predicates can be classified w.r.t. the theories they belong to. For example, the predicate “!=” which means unequal can be classified into the theory of linear arithmetic, while a predicate with uninterpreted terms is classified into the theory of uninterpreted functions with equality \[35\]. Other theories of interests include the theories of arrays and list structures (useful for modeling and verifying software programs), and the theory of bit-vectors (useful in modeling and verifying hardware designs).

Anyway, most SMT solvers support only quantifier free fragments of their logics. This is due to the fact that generally speaking the use of quantifiers can make the theory undecidable. One way to cope with this problem is to use quantifier elimination techniques \[21\].

### 2.2.2 The SMT solver Z3

Z3 \[18\] is a state-of-the-art SMT solver from Microsoft Research which integrates a host of theory solvers in an expressive and efficient combination. It is used in this thesis because of its efficiency and its convenient interface to the host programming language F# \[1\] which is used for the implementations of the thesis.

In the following each decidable theory used in this thesis is illustrated with a Z3 example (some of the examples are taken from \[18\]). The examples are written in the SMT-LIB language \[7\] which is pretty much self-explanatory. A list of supported types of Quartz expressions can be found in the appendix of the thesis.

#### Example 1. Boolean Arithmetic

```plaintext
(declare-preds ((p1) (p2) (p3) (p4) (p5)))
(assert (=> p1 p2))
(assert (=> p1 p3))
(assert (=> p1 p4))
(assert (not p2))
(check-sat)
```

This example encodes the satisfiability checking for the boolean formula:

\[(p_1 \rightarrow p_2) \land (p_1 \rightarrow p_3) \land (p_1 \rightarrow p_4) \land p_2\]

The first line declares the predicates appearing in the formula. The 2nd to 5th lines encodes the implications and the negation sub-formulae. As these assertions are made
separately, it is implicitly indicated that these assertions should hold at the same time, which corresponds to the conjunction of the sub-formulae. The final line triggers the satisfiability checking of the encoded formula.

**Example 2. Integer Linear Arithmetic**

```lisp
define-fun is-power-of-two ((x BitVec[4])) Bool
(= bv0[4] (bvand x (bvsub x bv1[4])))
(declare-funs ((a BitVec[4])))
(assert (not (iff (is-power-of-two a)
or (= a bv0[4]) (= a bv1[4])
(= a bv2[4]) (= a bv4[4]) (= a bv8[4])))))
```

This example checks whether a 4-bit bitvector is 0 or a power of 2. The first line defines a function called “is-power-of-two”, which takes a 4-bit bitvector x and returns: ```x + (x – 1)```. It is clear that if x is power of two, then ```is – power – of – two(x)``` should return 0. The following lines checks for a given 4-bit bitvector a if it is power of two. If a is power of two, then it should be one of the bitvectors of 0, 1, 2, 4 or 8. The result returns Unsat which means the assertion is unsatisfiable. This is another way of saying the negation of the assertion is valid, which is the result we expected.

**Example 5. Array Theory**
Line 1 defines a function named A with a one-dimensional array type. Line 5 and line 6 states two assertions: line 5 asserts that the content of $a_1[x]$ is $x$, while line 6 asserts that $y$ is stored in location $a_1[x]$.

**Example 6 mix Array Theory with linear arithmetic**

This example checks the formula: $a_1[i + 1] \equiv a_1[j + 2] + 1$.

### 2.3 Data Dependency Analysis for sequential programs

Previously dependency analysis has been heavily studied for sequential programs [2, 26, 27, 35]. It’s been vital for optimization techniques like instruction scheduling and data-cache optimization. In particular, it determines the ordering relationships between instructions. We give the classical methods a short review here. The definitions are referenced from [31].

#### Types of Dependencies

If statement $S_1$ precedes $S_2$ in the given execution order, we mark it as $S_1 \prec S_2$. A *dependence* between two statements in a program is a relation that constraints their execution order. The following shows the types of dependencies:

1. *Control Dependence*: If $S_1 \prec S_2$ and the execution of $S_2$ depends on the result of the execution of $S_1$;
2. Flow Dependence: If $S_1 \prec S_2$ and $S_1$ writes a value that is read by $S_2$ (Read after Write);

3. Antidependence: If $S_1 \prec S_2$ and $S_1$ reads the value written by $S_2$ (Write after Read);

4. Output dependence: If $S_1 \prec S_2$ and both $S_1$ and $S_2$ write to the same variable (Write after Write);

The control dependence analysis needs the detection of control flow statements. This is typically done during the control flow analysis. Data dependence analysis starts from dataflow analysis within basic blocks. The following gives a possible way to formulate data dependences utilizing linear arithmetic and boolean algebra.

**Definition 3** (Dependence in Satisfiability). Assume that a sequence of statements that will be executed sequentially is given. The type of statement considered for dependence analysis is defined as:

- $S_i : v_j = f(v_{k_1}, ..., v_{k_n})$, $S_i$ is a natural number indicating the position of the statement, $f$ is a linear arithmetic function, $v_j$, $v_{k_i}$ are variables.
- $S := \{S_1, ..., S_n\}$ the set of all position variables as natural numbers within the basic block;
- Assignment $f_s : S \rightarrow \mathbb{N}$ where each $S_i$ is assigned natural $i$, thus $S_i = i$; $V := \{v_1, ..., v_k\}$ is the set of bool variables representing the appeared variables in the statements of the basic block;
- $RD := \{rd_1, ..., rd_n\}$ is the set of bool variables where $rd_i := v_{i_1} \lor ... \lor v_{i_m}$, $v_{i_j} \in V$ representing the variables read by the statement $S_i$;
- $RT := \{wt_1, ..., wt_n\}$ is the set of bool variables where $wt_i = v_j \land \neg V_{i_1} \land ... \land \neg V_{i_m}$, $v_j, v_{i_k} \in V$ representing the variable $v_j$ written by the statement $S_i$. The other variables are the rest of the variables in $V$.

Based on the above definition a statement $S_i : v_j = f(v_{k_1}, ..., v_{k_n})$ can be translated to:

$(S_i = i) \lor (rd_i = v_{k_1} \lor ... \lor v_{k_n}) \land (wt_i = v_j \land \neg v_{i_1} \land ... \land \neg v_{i_m})$

and the data dependences can be formulated as follows:

- **Flow Dependence (Read After Write):** there exists an assignment of $V$ such that for $S_i, S_j$ (if $S_i \prec S_j$) and $(wt_i \land rd_j)$ is true;
- **Antidependence (Write After Read):** there exists an assignment of $V$ such that for $S_i, S_j$, (if $S_i \prec S_j$) and $(wt_i \land rd_j)$ is true;
2.3 Data Dependency Analysis for sequential programs

1: a := b + c;
2: b := a – d;
3: c := b + c;
4: d := a – d;

(define-funs((S1 Int)(S2 Int)(S3 Int)(S4 Int)))
(declare-preds((a) (b) (c) (d)))
(declare-preds((rd1) (rd2) (rd3) (rd4)))
(declare-preds((wt1) (wt2) (wt3) (wt4)))
(assert(and (= S1 1)
  (= rd1 (or b c))
  (= wt1 (and a (not b) (not c) (not d)))))
(assert(and (= S2 2)
  (= rd2 (or a d))
  (= wt2 (and b (not a) (not c) (not d)))))
(assert(and (= S3 3)
  (= rd3 (or b c))
  (= wt3 (and c (not a) (not b) (not d)))))
(assert(and (= S4 4)
  (= rd4 (or a d))
  (= wt4 (and d (not a) (not b) (not c))))))
/*test S1 and S2 has RAW*/
(assert(and (< S1 S2) (and wt1 rd2)))

Figure 2.4: Example showing RAW dependence between $S_1$ and $S_2$

- **Output Dependence (Write After Write):** there exists an assignment of $V$ such that for $S_i, S_j$, $(wt_i \land wt_j)$ is true.

If there exists an assignment of the boolean variables that can make the boolean expression evaluate to true, we call this expression is satisfiable.

For example in Figure 2.4 As we can see that $(wt1 \land rd2) = (a \land (\neg b \land \neg c \land \neg d)) \land (a \lor d)$, there exist an assignment with $a = true$ that satisfies this formula. The other component of the formula $(S1 < S2)$ is translated to $(1 < 2)$ which holds obviously. Thus the assertion is satisfiable, which means that there is a RAW dependence between $S1$ and $S2$.

**Dataflow Analysis**

The most important dataflow analysis problems are listed here [31]:

- **Reaching Definitions:** for each basic block, which variable definitions are available at the entrance;
- **Available Expressions:** for each basic block, which expressions are available at the entrance;
• **Live Variables**: for a variable at a particular position, whether there is a use of the variable along some path from this point to the exit;

• **Upwards Exposed Uses**: dual problem of reaching definitions-determine which are the definitions for a particular usage of a variable;

• **Copy Propagation Analysis**: determines the path from the start of the copy to the furthest use of the defined variable while its value remains;

• **Constant-Propagation Analysis**: determines the path from the start of the constant assignment to the furthest use of the defined variable while its value remains;

• **Partial-Redundancy Analysis**: determines the computations that performed twice without the operands in between changed.

The traditional way of dataflow analysis is to formulate the interested objects as bit-vectors and utilizing lattice theory for a fixpoint computation [31].

**Loop Dependence Testing**

Experiments show that loops are the most frequently executed parts of a program. A good optimization of loop structures can usually lead to great improvement of the runtime performance. By analyzing the data dependencies in loop structures we find potentials for optimization.

A basic problem in loop dependence testing is the so called *array reference testing* – to test whether two array references point to the same array element. Based on the test, optimizations like loop interchange and vectorization transformation [3, 4] can be done safely. There are plenty of methods developed for array reference testing. To name some of them: GCD testing, Banerjee inequations, Omega testing, Power testing etc [30, 33, 44]. Most of these methods try to test particular properties of the reference formulae. If the tested property fails then it is sure that there is no data dependence. However these tests are usually only necessary conditions for the existence of data dependences which make the tests not exact. As this problem can be formulated by using linear arithmetic, it permits SMT to be used for testing satisfiability of the equality of the two references under given conditions. In this way, an exact answer can be derived. The types of loop dependences can be defined as follows [28]:

**Definition 4.** Statement $S_2$ has a loop carried dependence on statement $S_1$ if and only if $S_1$ and $S_2$ reference the same data element on different loop iteration vectors.

**Definition 5.** Statement $S_2$ has a loop independent dependence on statement $S_1$ if and only if $S_1$ and $S_2$ reference the same data element always on the same iteration vectors.

For a perfect nested loop shown in Figure 2.5, $f$ and $g$ are linear functions that map
2.3 Data Dependency Analysis for sequential programs

For $I_1 = l_1, \ldots, u_1$:
For $I_2 = l_2, \ldots, u_2$:
... For $I_n = l_n, \ldots, u_n$:

S1: ... $= A[f(I_1, \ldots, I_n)]$;
S2: $A[g(I_1, \ldots, I_n)] = ...$;

Figure 2.5: A perfect loop

iteration vectors to indexes of the array $A$. For testing whether $S_1$ and $S_2$ have loop
independent dependence or not, we have to answer the question that whether or not for
all possible iteration vector $(I_1, \ldots, I_n)$, $f(I_1, \ldots, I_n) = g(I_1, \ldots, I_n)$ is satisfiable. For
testing whether $S_1$ and $S_2$ have loop carried dependence or not, we have to answer the
question that whether or not there exist two different iteration vectors $(I_1, \ldots, I_n)$ and
$(J_1, \ldots, J_n)$ such that $f(I_1, \ldots, I_n) = g(J_1, \ldots, J_n)$ is satisfiable.

An example of checking loop dependencies is shown in Figure 2.6.

/* loop carried dependence testing*/
(declare-funs ((i Int) (j Int)))
(declare-funs ((a Int) (b Int)))
(assert (= a 3))
(assert (= b 2))
(assert (and (<= 1 i) (<= 100)))
(aSse rt (= (+ i a) (~ i b)))

/* loop independent dependence testing*/
(declare-funs ((a Int) (b Int)))
(declare-funs ((i Int)))
(assert (= a 3))
(assert (= b 2))
(assert (and (<= 1 i) (<= 100)))
(aSse rt (= (+ i a) (~ i b)))

Figure 2.6: Examples showing loop dependence detection
3 Optimization

This chapter describes the main contribution of the thesis. The first section gives the definition of *Extended Finite State Machine* (EFSM). Based on this definition, the dependency analysis of synchronous languages is discussed. The second section introduced the algorithm used for the generation of the EFSM. The rest describe the optimization techniques based on the dependency analysis in details.

3.1 Dependency Analysis of EFSM

Before we give the definition of EFSM, there’re still some prerequisites to be introduced. The basic element in synchronous dependency analysis is a *guarded action*. Previously in Section 2.1.2 an informal definition of guarded action was given. For the sake of analysis, we’ll define the concept more formally:

**Definition 6** (Guarded Action). A guarded action is a four-tuple \((\text{grd}, \text{lhs}, \text{rhs}, \text{assn})\) where \(\text{grd}\) is a boolean expression refers to as the guard, \(\text{lhs}\) a variable refers to as the variable that is assigned a value, \(\text{rhs}\) refers to as the right-hand-side expression and \(\text{assn}\) refers to as the type of the action (AssignNow or AssignNxt). Together the triple \((\text{lhs}, \text{rhs}, \text{assn})\) is the action.

There are two types of guarded actions: *control flow guarded actions* (CGA) and *data flow guarded actions* (DGA). The difference between a CGA and a DGA is that the \(\text{lhs}\) is a location variable for a CGA and a data variable for a DGA. As introduced in Section 2.1.2, the set of CGAs form the control flow of the synchronous program and the DGAs form the data flow of the program.

The EFSM is chosen as the representation on which the dependency analysis is performed. The reason choosing EFSM is the appropriate abstraction level. It is independent of any implementation language as well as target languages and hardware platforms. A source code level analysis would be difficult as the nested preemption-statements (abort, suspend etc.) and loops would make the structure of the source code very complex.

Informally, each state of the EFSM represents a subset of the control-flow labels (i.e. assignments of a set of location variables) while transition edges represent the
control flow of the original synchronous program. An EFSM transition takes place
when the control flow moves from one position to another. A Formal definition is
given as follows:

**Definition 7** (Extended Finite State Machine). An Extended Finite State Machine
(EFSM) is a tuple of \((s_0, S, T)\), where \(S\) is a set of states, \(s_0 \in S\) is the unique initial
state, and \(T \subseteq (S \times C \times S)\) a finite set of transition relations, where \(C\) is a set of
transition conditions.

**Observation 8** (Guarded Transition). *Definition 7* implicates that a transition \((s, c, s')\)
can take place only if the transition condition \(c \in C\) is evaluated to true.

A transition of an EFSM corresponds to a macro-step (or a reaction step) of the
reactive system modeled. The following definition describes an important property of
the transition of an EFSM:

**Definition 9** (Well-defined EFSM). An EFSM \((s_0, S, T)\) is well defined if only if it is
deterministic and complete, i.e. for any state \(s \in S\), and all transitions \((s, c_i, s'_i) \in T\),
\(\forall_i c_i\) evaluates to true and for any \(s_i, s_j\) where \(i \neq j\), \(c_i \land c_j\) evaluates to false.

This definition reflects that each state of the EFSM corresponds to a control position
in the control flow of the synchronous program. For a synchronous program, the global
control flow can only reside in one control position at any time. This means that for a
given EFSM state \(s\), if \(s\) has more than one successor state, at most one of the successor
states can be reached at the same time, i.e. at most one of the transition conditions w.r.t. the successors’ transitions can be evaluated to true. The EFSM \((s_0, S, T)\) can be
generated from the control flow guarded actions. The next section describes the EFSM
generation in detail.

**Fact 10.** At any time there is at most one transition that can take place in an EFSM.

For the data flow of the synchronous program, the information is encoded in the
DGAs. The following definition introduces the concept of the data flow in the context
of EFSM:

**Definition 11** (Attached Guarded Actions). For a given EFSM \((s_0, S, T)\), for each
state \(s \in S\), there is a set of attached data flow guarded actions (AGA) attached to that
state representing the micro-steps of that state.

The set of AGAs of each state is extracted from the data flow guarded actions. The
next section introduces this procedure in more detail. Because of the perfect synchrony
property, all the AGAs that can be executed within one state of EFSM are concurrent.
This simplifies the dependency analysis to some extend as there is no difference be-
tween guarded actions with different sequential orders. However we will see later that
the occurrence of guards brings some additional complexity in the analysis.
3.1 Dependency Analysis of EFSM

**Observation 12** (Guarded Action). An action is executed only when the guard is evaluated to true.

This observation can be easily deduced from the definition of the guarded action.

**Definition 13** (Reaction to Absence). A variable is assigned to a default value only if it is an event variable, and in the state there’s no action that can be executed to give the variable an assignment.

**Observation 14** (Deterministic Assignment). At any time, the variables within one EFSM state are all assigned with deterministic values.

This observation can be proved from perfect synchrony, absence of causality cycles or write conflictions, Observation 12 and Definition 13. The only point that is not covered is the assignment of a memorized variable. In the initial state the memorized variables with no assignments have type-consistent default values. If a memorized variable is neither defined in the current state nor is defined in the previous state, its value remained as the value in the previous state.

Observation 14 reveals the fact of the deterministic behavior of a reactive system. In any macro-step, the variables have deterministic values. This requires that the AGAs within each state are executed according to the data dependency. In other words, although the AGAs are instantaneous, they have to be executed in a particular order so that the values are correctly computed. For example, consider the following set of AGAs of a state $s$:

\[
\begin{align*}
AGA_1 &: True => x = y + 1 \\
AGA_2 &: True => y = 3
\end{align*}
\]

Apparently the two actions are all to be executed. But if we want to execute the assignment of AGA1, it is required that the value of $y$ is known already. This means that we must execute AGA2 before AGA1.

Besides the data dependencies among AGAs within a state, there are also data dependencies between states. There are two causes: the existence of delayed assignments and the memorized variables. If a variable is not assigned explicitly in the current state, then either it is assigned a default value if it is an event variable or it is assigned a value from the previous state by a delayed assignment or it maintains its value in the previous state if it is a memorized value. In the latter two cases the current value of the variable depends on the previous state. This causes a cross-state data dependency.

The data dependency can be formally analyzed by using *Action Dependency Graphs* (ADG) which is defined as follows:
**Definition 15** (Action Dependency Graph). An Action Dependency Graph w.r.t a set of CGAs and DGAs is a bipartite graph \((V, E)\) where \(V\) is the set of vertices and \(E\) the set of directed edges. A vertex represents a memorized variable, an event variable or a guarded action. An edge starts either from a variable vertex and points to a guarded action vertex, or starts from a guarded action vertex and points to a variable vertex. The former type of edges represents that the guarded action reads the variable (read edge) while the latter type of edge represents that the variable is written by the guarded action (write edge). For the write edges, it could either be an instant write or a delayed write.

The example in Figure 3.1 shows a Quartz program, its CGAs, DGAs and the corresponding ADG.

```plaintext
module Parallel (mem bool !y1, !y2) {
  event bool x1, x2;
  loop {
    w1: pause;
      { x1 = true; if (x2) y2 = true; }
      ||
      { x2 = true; if (x1) y1 = true; }
  }
}
```

**Figure 3.1:** Examples showing the guarded actions and the ADG of a program

**Definition 16** (Read and Write Dependencies). Let \(FV(t)\) denote the free variables occurring in the expression \(t\). Then the following definitions determine the dependencies from actions to variables:

For the given guarded action \(GA = (grd, lhs, rhs, assn)\):

- \(grdVars(GA) := FV(grd)\)
- \(rhsVars(GA) := FV(rhs)\)
3.1 Dependency Analysis of EFSM

\[
\begin{align*}
rdVars(GA) & := \text{grdVars}(GA) \cup \text{rhsVars}(GA) = FV(\text{grd}) \cup FV(\text{rhs}) \\
wrVars(GA) & := lhs
\end{align*}
\]

while the dependencies from variables to actions are determined as follows:
\[
\begin{align*}
grdActs(x) & := \{ GA | x \in \text{grdVars}(GA) \} \\
rhsActs(x) & := \{ GA | x \in \text{rhsVars}(GA) \} \\
rdActs(x) & := \{ GA | x \in \text{rdVars}(GA) \} \\
wActs(x) & := \{ GA | x \in \text{wrVars}(GA) \}
\end{align*}
\]

In an ADG, the incoming edges of a variable vertex \(x\) from the vertices of guarded actions represent those write actions (\(wActs(x)\)) and the out-going edges represent the read actions (\(rdActs(x)\)). The incoming edges of a vertex of guarded action \(GA\) from the vertices of variables represent those read variables (\(rdVars(GA)\)) while the out-going edge represents the write variable (\(wrVars(GA)\)).

**Observation 17** (Unique Write Variable). For any guarded action vertex of an ADG, there is only one out-going edge point to a variable vertex.

This observation is simply proved by the fact that for each guarded action, there is only one written variable.

More information can be derived from the ADG. For example we can calculate that which guarded actions can be executed simultaneously without considering their execution order. But there is also information not encoded inside an ADG. In particular, the absence of reactions and the relayed values of memorized variables are not covered in ADGs. The ADG is built based on variables and guarded actions, anyway the one thing that truly affects the value of a variable is the execution of an action. If it can be guaranteed that a guarded action \(GA\) is to be executed, we know that the written variable \(lhs\) definitely depends on the variables in \(rhsVars(GA)\). But if we are not sure of the execution, it might be the case that the action is never executed and the data dependency never exists. To make more precise analysis, we need to take care of the guards to check the possibility of the actions’ execution.

In particular, by Observation 12 it is clear that the execution of a guarded action \(GA\) depends on the truth value of the guard, which in turn depends on \(grdVars(GA)\). The assigned value depends on the \(rhsVars(GA)\). The following facts give the condition of when a guard can be evaluated to true:

**Fact 18** (Reachable State). For a given EFSM \((s_0, S, T)\), a state \(s \in S\) is reachable only if \(s\) is the initial state \(s_0\) or there exists a transition \((s', c, s) \in T\) such that \(c\) is satisfiable in state \(s'\) and \(s'\) is reachable.

**Fact 19** (Satisfiability of Guard). For a given EFSM \((s_0, S, T)\) and the corresponding AGAs, for a guarded action \(GA\) in a set of AGAs of a state \(s\), the guard \(g\) can be evaluated to true only if the state \(s\) is reachable and in \(s\) there exists an assignment of variables \(grdVars(GA)\) such that \(g\) is evaluated to true.
Fact 20 (Satisfiability of Transition Condition). For a given EFSM \((s_0, S, T)\) with transition \(t = (s, c, s') \in T\), the transition condition \(c\) is satisfiable, i.e. can be evaluated to \(true\) if only \(s\) is reachable and in state \(s\) there exists an assignment of variables \(FV(c)\) such that \(c\) evaluates to \(true\).

By the above facts, we can see that the dependency relations in an EFSM are determined by the guards and transition conditions. Once the truth value of a particular guard or a transition condition is fixed, the corresponding data dependencies are fixed. In other words, in the ADG there might be edges that can be eliminated once the guards are evaluated to \(false\) and there might be vertices that can be eliminated once the transition conditions are evaluated to \(false\). This observation is the main motivation of the optimization techniques discussed in the following sections.

3.2 Generation of EFSM

In Section 2.1.3 the \(AIFSystem\) structure is introduced. Among its many parts, there are two parts that are of interest: \(init\) and \(main\). These two parts store the compiled control flow and data flow of the program. For the following Quartz program

```plaintext
module Test(bool ?i1 ,?i2 ,
       bool !y) {
  event x1, x2;
  loop {
    y = x2; w1: pause;
    y = x1; w2: pause;
  }
  ||
  loop {
    next(x1) = i1 & i2; x2 = i1 | i2 ;
    w3 : pause;
  }
}
```

the corresponding \(AIFSystem\) structure is listed as follows in Figure 3.2.

In the \(AIFSystem\) structure in Figure 3.2, the parts \(iface\) and \(local\) list all the variables used in the program. In the parts \(local\), \(w1\), \(w2\) and \(w3\) are the location variables. All the other variables are data variables. Both the parts \(init\) and \(main\) have control flow and data flow fields. These are the places storing the compiled guarded actions of the program.
system Test:
    interface:
        i1: input bool
        i2: input bool
        y: output memorized bool
    locals:
        w1: label bool
        w2: label bool
        w3: label bool
        x1: local event bool
        x2: local event bool
    abbreviations: ...
    init:
        control flow:
            true => next(w1) = true
            true => next(w3) = true
        data flow:
            true => y = x2
            true => next(x1) = i1|i2
            true => x2 = i1|i2
        assertions:
    main:
        control flow:
            w2 => next(w1) = true
            w1 => next(w2) = true
            w3 => next(w3) = true
        data flow:
            w2 => y = x2
            w1 => y = x1
            w3 => next(x1) = i1|i2
            w3 => x2 = i1|i2
        assertions:
        absence reactions:
        invar: ...
        specifications: ...

Figure 3.2: The AIFSystem structure of the Test program
3.2.1 Generation of States and Transitions

The states of the EFSM can be derived from the control flow of the program. The algorithm used here is based on the method introduced in [25]. As already discussed in Section 2.1.3, the EFSM states are encoded by the location variables, and the state transitions are encoded in the control flow of the program represented by CGAs. We can derive the initial state by definition, and put it into the CGAs to run a state transition of the encoded state machine to derive the successor states and transitions, i.e. a symbolic simulation. At first there are two sets of CGAs given in an AIFS System structure for surface and depth of the program respectively. They can be combined into one set of CGAs easily by using the next definitions:

**Definition 21 (Control Flow Guarded Action).** A control flow guarded action is of the form $(p, vl, true, AssignNext)$, where $p$ is the guard of the CGA and $vl$ is the location variable.

**Definition 22 (Combined Control Flow Guarded Actions).** Given two control flow guarded actions: $(grd_1, lhs_1, true, AssignNext)$ and $(grd_2, lhs_2, true, AssignNext)$, if $lhs_1$ and $lhs_2$ are the same location variable $lhs$, the combined control flow guarded action is $(grd_1 \lor grd_2, lhs, true, AssignNext)$.

By combining the CGAs in init and main a single set of CGAs which fits all control flow positions can be achieved. Now the whole algorithm can be described informally as follows:

During the whole computation a queue of states is maintained. Initially there is only one state in the queue – the initial state. The algorithm starts from the head of the queue and runs through the queue. For each state in the queue the successor states can be computed from the CGAs. In this way the transition relations of a state can be computed. In the meantime the algorithm checks if there is any new state generated in the successor states. If so, the new state is put at the end of the queue. As there are finitely many location variables and finitely many CGAs, the queue has a finite length and the algorithm terminates. Once the procedure runs through every state in the queue, the computation is finished.

**Generation of the Successor States**

We introduce a location variable $st$ representing the starting position of the program. With the aid of the starting location variable $st$, the initial state of an EFSM can be defined as follows:

**Definition 23 (State of EFSM).** Given the set of location variables $Vl$ of an EFSM, a state of EFSM is encoded by using the location variables, i.e. each state of EFSM is a function assignment of the variables: $\{st\} \cup Vl \rightarrow \mathbb{B}$. 
Given this definition, the label of state $s$ can be defined as: $\text{LabelGrd}(s) := \land_{l \in L}(l)$ and the AGAs of state $s$ can be defined as:

$\text{GrdAct}(s) := \{(\text{grd}, \text{lhs}, \text{rhs}, \text{assn})| \text{grd} \rightarrow \text{LabelGrd}(s)\}$.

**Definition 24 (Initial State).** An initial state of an EFSM is encoded by setting the starting location variable $st$ to $true$ while all other location variables are set to $false$.

Given the above definitions, the algorithm generating the successor states for a given state can be described as shown in Figure 3.3. In Figure 3.3, current state and list of

![Diagram](image)

Figure 3.3: Compute the successor states

CGAs are the original inputs. A detailed explanation is given below:
1. The values of the location variables (i.e. the encoding) can be derived directly from the current state.

2. Assume the assignments of the location variables of the current state is $A$. For a CGA $GA$, assume the guard is $grd$. Let $[grd]_A$ be the result of a partial evaluation of $grd$ using $A$. $GA$ is related if $grd$ fulfills: $[grd]_A \neq false$. The set of CGAs can then be divided into related CGAs and unrelated CGAs. If $GA$ is related, $[grd]_A$ is either evaluated to $true$ or to a simplified expression (containing no location variables). A related CGA is also called a cofactor.

3. For an unrelated CGA, it is sure that the location variable will be assigned to $false$ at the successor state.

4. For the cofactors, we have to check the truth value of the guard. If it is evaluated to $true$ / $false$, then the corresponding location variable will be assigned to $true$ / $false$ respectively at the successor state. These are the set of pairs (location variable, boolean value) in Figure 3.3. If it is evaluated to an expression whose boolean value can’t be fixed, then we have to take both truth value into consideration, i.e. a case distinction is needed. These are the set of pairs (local variable, guard) in which guard is the expression whose boolean value can’t be fixed.

5. For the case distinction, we can compute every possible case of the unfixed location variables. Together with those location variables that have been fixed the set of all possible boolean assignments to the location variables can be generated. These assignments are the successor states.

One point that needs a little more clarification is Step 2. Only a cofactor provides the possibility of assigning the location variable to $true$. Informally, in the original program a particular location variable $vl$ is to be set to $true$ at the next macro-step only because the control flow can reach this location $vl$ from the current location $v$ (which is set to $true$ currently). The location variable $v$ is called an active location. This indicates that for a particular CGA, if the $lhs$ variable is to be set to $true$, there must be an active variable in the guard so that the guard $grd$ is satisfiable. Let current state be $s \in S$, $ActiveLocation(s)$ be the set of active locations in state $s$, then: $\forall l.l \in ActiveLocations(s),\ grd \rightarrow l$.

Now we examine in detail the process of case distinction. As shown in the above explanation, the case distinction deals with the location variables whose truth value are still not fixed. A basic idea is to consider both possible cases of each particular variable. Thus if there are $n$ unfixed location variables, there should be $2^n$ many different cases. These location variable assignments with the fixed location variable assignments together form the $2^n$ successor states. A further observation is that once we assigned an unfixed location variable to a particular boolean value, we take the assumption that the guard of this variable is taken the same boolean value. This might in turn help us to fix the values of some other unfixed location variables. For example:
where \( w1 \) is assigned to \( true \), the assignments of \( w2 \) and \( w3 \) depend on the value of \( op \) which is still not fixed. But if we take the assumption that \( w2 \) is \( true \), then we assume already that \( op \) holds as well. This will help us to deduce the fact that under this condition \( w3 \) is to be assigned \( false \).

Based on this observation, the algorithm of case distinction can be designed as shown in Figure 3.4. The input is the list of \((location, variable)\) pairs. Following is a detailed explanation:

1. For the head of the list, two cases are generated – the \textit{guard} evaluates to \( true \) and \( false \) respectively. Assume the guard is \( guard_0 \).

2. For each pair in the tail of the list, the guard \( guard_i \) is checked based on the case distinction made for \( guard_0 \). For the case that the guard in the head is \( true \), a conjunction of \( guard_0 \) and \( guard_i \) is computed. If the conjunction evaluates to \( false \), then it is sure that the corresponding location variable should be set to \( false \). Otherwise the truth value is still not known and the result of the conjunction is saved for the next iteration.

3. For the case that the guard in the head is \( false \) the treatment is similar. A disjunction of \( guard_0 \) and \( guard_i \) is computed first. If the disjunction evaluates to \( true \), then it is sure that the corresponding location variable should be set to \( true \). Otherwise the result of the disjunction is saved for the next iteration.

4. If there are any unfixed location variables left, a new iteration for the computation is performed.

This algorithm terminates because there are finitely many unfixed local variables. An upper bound of the iteration would be \( 2^n \) with \( n \) the number of unfixed local variables. For \( guard_0 \) is \( true \), the conjunction computation is:

\[
(\text{guard}_0 \land \text{guard}_i) = (true \land \text{guard}_i) = \text{guard}_i
\]

If the conjunction evaluates to \( false \), then it is only because \( \text{guard}_i \) is \( false \). The result can not be \( true \), since \( \text{guard}_i \) is unfixed. Thus, we only get two possible results – either the conjunction evaluates to \( false \) or it evaluates to an expression whose value is still unknown. The explanation is similar for the disjunction computation.

Given the algorithm of calculating the successor states, the complete algorithm of EFSM generation is depicted in Figure 3.5. Back to the example introduced at the beginning of this section, we can derive the combined CGAs:

\[
\begin{align*}
w2 & => next(w1) = true \\
w1 & => next(w2) = true \\
w3 & => next(w3) = true
\end{align*}
\]
Figure 3.4: Case distinction of unknown expressions
Create an empty Queue for the states of the EFSM

set the initial state as the head of the queue

Empty queue of EFSM states

queue of EFSM states

Get the head of the queue

state

Generate the successor states of the current state

List of old states

List of new states

Empty List?

Yes

END

Put the states in the back in the queue

No

Figure 3.5: Extended Finite State Machine Generation
The computation of successor states is shown in Figure 3.6:

Figure 3.6: Example showing states generation

### 3.2.2 Generation of Attached Guarded Actions

The DGAs are not executed in every EFSM state but only when they are activated. The way of checking whether a DGA should be attached to a particular state $s$ is to check whether the guard of the DGA reads any of the active variables of the state. If so, it is sure that the DGA is executed in the state.

Figure 3.7 shows the generated EFSM of the program Test introduced at the beginning of this section:

Figure 3.7: Example showing the EFSM of the program Test
3.3 Optimizations

Based on the EFSM generation discussed previously, in this section, we focus on the central point of the thesis – optimizations. The goal of EFSM optimization is to reduce the size of the EFSM – the number of the states and transitions and the number of the AGAs of states. The EFSM generation has already considered the guards of the guarded actions in the abstraction level of propositional logic, i.e. to treat every sub-expression containing non-boolean sub-expressions but returning a boolean value as a boolean variable. Facts 19 and 20 indicate the situations of when to execute the actions. In particular, a guard that is evaluated to true brings no constrictions to the execution of the guarded action – the action will always be taken place at every cycle. An action whose guard is evaluated to false will never take place. A guard evaluated to an expression whose value is unknown will take place when the expression is satisfied. In the last case, the boolean-based EFSM generation method simply makes case-distinctions on the unknown-value expression and considers both situations.

A closer look at the sub-expressions of the guards may give us more information of the guard, and thus helps us to make more precise decisions. It is usually the case that the sub-expressions of a boolean guard are actually arithmetic expressions with other types. For example:

$$(x + y < 3) \land (x > 5) \land (\neg \text{op})$$

where the first two conjuncts $(x + y < 3)$ and $(x > 5)$ are actually linear arithmetic expressions. But in the level of boolean-based EFSM generation (propositional logic) they are treated as boolean variables (propositions). By taking a closer look into the sub-expressions, we actually bring the level of analysis into predicate logic.

**Observation 25.** By Definition 2, a guard or a transition condition is an SMT instance of some theory.

Here we can take advantage of the state-of-the-art SMT solvers helping us to answer the question of decision problems in the context of some decidable theories, for example, integer linear arithmetic theory.

The idea is clear: to treat the guards that are considered formerly as pure boolean expressions in a more precise way (or in a lower abstraction level) – as SMT instances, and check the satisfiability of all the predicates that belong to the decidable theories, which can be solved by an SMT solver. For example, by using an SMT solver we know that $(x < 5) \land (x > 6)$ is unsatisfiable, i.e. no assignment of variable $x$ exists such that the boolean expression is satisfied. In this case the case-distinction procedure can be saved. By Facts 19 and 20, when knowing the guard is unsatisfiable we again can eliminate this guarded action as we know that the action is never executed.

Based on this basic idea, a two-step SMT based EFSM optimization is developed. The first step is called dead code elimination. In this step an SMT solver is utilized for
checking the satisfiability of the guards of AGAs and the transition conditions in the
given EFSM. Those unreachable states as well as unsatisfiable AGAs are eliminated.
The second step is called passive code elimination. Based on the first step, the AGAs
that have no contribution to the computation of the outputs, i.e. passive codes, are
detected and eliminated. In the following sections the two steps are examined in detail.

3.3.1 Constant Propagation

The idea for constant propagation comes from the classical constant propagation tech-
niques [43]. If we can fix the value of a variable, we therefore reduce the size of
dependent variables of an expression. Moreover, there might be guards whose truth
values can already be fixed. In this section, two methods of constant propagation are
developed. The first is called internal state constant propagation. As indicated by
its name, the method only considers the constants within a given EFSM state. In the
second method cross state constant propagation, the propagation of constants across
states is added.

Basic Idea

In constant propagation, the environment of the given state $s$ is established for helping
to check the satisfiability of the guards of AGAs as well as the transition conditions.

Definition 26 (Environment). For a given EFSM with $V := \{x_1, ..., x_n\}$ the set of
variables, where $x_i$ has domain $D_i$. An environment $E$ of an EFSM state is defined as
the mapping of the variables of the state to their domains respectively, i.e. for $x_i \in V,$
$E(x_i) = v$ with $v \in D_i$.

Note that in this definition, the set of variables $V$ already contains the incarnated
variables. Thus, an environment can be partial, i.e. some variables may have no values
mapped. This happens for example when there are incarnated variables existed in the
EFSM, and the environment of the initial state has no mapping between an incarnated
variable and any value within its domain.

The environment is complete iff all the variable assignments are known. At compile
time, as the inputs are not known yet, and many local variables and output variables’
calculations depend on the input values, the environment is usually not complete. However it might be possible to compute part of the environment instead of
the complete environment. Thus, the idea of internal state constant propagation is to
gather whatever useful information inside a given state to compute a partial environ-
ment and to use this partial environment to compute the satisfiability of the guards.
3.3 Optimizations

Obviously, this method is not complete, since the truly complete environment can only be derived in the run-time. Only then, every guard can be evaluated to a deterministic value.

Program state and EFSM state

In general, a state of a program is usually defined by the assignment of the data variables, i.e. an environment. A problem using environment as the state of the program is state explosion. Theoretically when the variables have infinite data types the number of states in a program may even be infinite.

In this thesis, however, the state of a program has another meaning. As defined in Definition 23, location variables are used for the encoding of an EFSM. A state is actually representing a particular location of the control flow of the program. Since finitely many location variables are used for a finite program, the number of states of a program is also finite. However by using this definition, a state of the program may have more than one environment. Consider the following example:

\[
\text{loop } \{
\text{l: pause; }
\text{next(x) := x + 1; }
\}
\]

Assume that \( x \) is a variable of natural number. There is only one state in the program (except for the initial state), but infinitely many possible environments for this state as \( x \) enumerates through the domain of natural numbers. To solve the decision problem in such a context is somehow tricky. Consider a simple variant based on the previous example:

\[
\text{loop } \{
\text{l: pause; }
\text{next(x) := x + 1; }
\text{if}(x > 100) \ y := \text{true; }
\text{else } y := \text{false; }
\}
\]

To find out when the inequation \( (x > 100) \) holds, one needs to consider every possible environment. As there are infinitely many environments, the question can’t be answered by literally examining all environments. For that we might establish an induction-like proof, or formulate it simply as a bounded model checking problem [5].

Anyway, in our case, we don not really need to know when the inequation holds. We just want to know that if it holds for all cases or not. In this case, the problem is much
simpler. Back to the context of constant propagation, we try to find the particular invariant such that it holds for all environments of a state or not. Formally, a congruence relation can be defined on the environments modulo the constants of one state. In an equivalence class of environments of a state, the values of the constant should be the same.

**Definition 27 (Congruence of Environments).** For a given EFSM with the set of variables $V = \{x_1, ..., x_n\}$, $s$ is a state of the EFSM and $E$ the set of environments of $s$. A congruence of $s$ is the set of variables: $W \subseteq V$ that $W := \{x_i | x_i \in V, \forall E_m \in E, \forall E_n \in E, E_m \neq E_n, E_m(x_i) = E_n(x_i)\}$. The variables in $W$ are called constants.

The purpose of constant propagation is to compute the congruence of a state, and use this congruence to help deciding the satisfiability of the guards of AGAs as well as the transition conditions.

**Internal State Constant Propagation**

A simple starting point for the computation of congruences is to gather information inside one state. In the context of synchronous languages, we can gather the following different types of information in one state:

1. **Constants directly encoded in the guard expression.**

   For example: for the condition $(x > 5) \land (x < 3)$, 5 and 3 are constants that are encoded directly. They can be used immediately for SMT decision making.

2. **Constants that can be deduced directly.**

   For example:

   $$\begin{align*}
   \text{true} & \Rightarrow x := 1 \\
   (x > 3) & \Rightarrow y := \text{true}
   \end{align*}$$

   In this example $x := 1$ will be executed deterministically. Thus, $x = 1$ holds forever. This could help us deduce the fact that condition $(x > 3)$ is never satisfiable.

3. **Constants that can be deduced by reaction of absence.**

   For the same example in 2, based on type 2 we already know that $(x > 3)$ is never satisfiable, which means that $y := \text{true}$ will never be executed. Thus we can further deduce the fact that $y = \text{false}$ always hold because of the reaction to absence (if no other possible assignment is executed on variable $y$).

However, a subtle case comes into play here: not all reaction to absence should be considered. Only those that can be deduced from the first two types should be taken
care of. This is because that a state in EFSM may have more than one environment. The information we are trying to gather here should be constants that remain in every possible environment of the state. Only then can we assure that the guard being considered is truly valid or never satisfiable in the state under all environments. Consider the previous loop example:

```plaintext
loop {
  l: pause;
  next(x) := x + 1;
  if(x > 100) y := true;
  else y := false;
}
```

Initially in state \( l \) the equation \( x = 0 \) holds because of the reaction to absence. Based on this equation we can then deduce that \( (x > 100) \) does not hold and \( y = \text{false} \) holds. However, this result only effects the state in the initial environment. As the environment evolves, \( x \) will be greater than 100 in the future, and the results deduced in the initial environment will be invalid.

The computation of the congruences depends on both the data flow within one state and the power of the SMT solver. Therefore, it is possible that there are some constants in the congruence that can not be identified. We call the sub-set of a congruence a partial congruence. An empty congruence is called an initial congruence.

The computation of partial congruences can be formulated as a fixpoint computation. For every state in a given EFSM, start from an initial congruence (no variable is mapped to a value):

1. Extract all dataflow guarded actions and then evaluate the guards according to the current congruence (type-[1] constants). Check the satisfiability of the evaluated guards. For those that are unsatisfiable, the guarded actions can be eliminated and reaction to absence can be considered (type-[3] constants). For the valid one, a symbolic execution of the action can be performed (possible type-[2] constants). Both situations will update the partial congruence.

2. If the partial congruence is modified from the previous one, then repeat Step [1]. Otherwise a fixpoint is reached and the computation is over.

Based on the partial congruence calculated in the fixpoint computation, the control-flow guarded actions, i.e. the state transitions can be further examined. For those transitions whose guards are unsatisfiable, the transition can be eliminated.
Collecting Reactions to Absence

The reactions to absence consider those variables whose guards of guarded actions are all unsatisfiable. Thus, for a particular variable, all the related guarded actions have to be checked. The procedure of internal-state constant propagation shown in Figure 3.8 maintains a list of pairs \((\text{variable}, \text{bool})\), which indicates the variables possibly suffering from reactions to absence. If a variable’s guard is satisfiable, the pair \((\text{var}, \text{false})\) is then updated to the map indicating that the variable does not react to absence, since its action can be executed.

Otherwise, if a variable’s guard is unsatisfiable, the pair \((\text{var}, \text{true})\) is put into the list. Before doing this, we have to make sure that the pair \((\text{var}, \_\_\_)\) is not in the list already. If an action that can be executed has been considered before, then there should be a pair of \((\text{var}, \text{false})\) inside the list already. The current unsatisfiable action then leaves it as it was before. If there was a \((\text{var}, \text{true})\) already inside, which means another unsatisfiable action was considered before, then there is no need to update the same entry. Only if there is no entry for \(\text{var}\) inside, the update is carried out.

Cross-State Constant Propagation

Previously, the discussion was limited to the partial congruences derived inside a particular state. Now we extend the method of constant propagation to cross-state constant propagations. Consider the example in Figure 3.9. In the above example, if only state \(L_1\) is considered, there is no information of the variable \(op\) that can be derived. If the predecessor states of \(L_1\) are considered, it can be deduced that \(\text{next}(op)\) is set to 2. Then it is clear that in state \(L_1\) the condition \((op > 5)\) is never satisfied, which leads to the reaction to absence that determines the value of \(y\) to be false.

This time the constants to be gathered are those that affect the successive states. For each state, not only the immediate assignments of the variables are stored, but also the assignments for the successive states.

Conservative Constant Propagation

One thing to be noticed in cross-state constant propagation is that not every state can propagate constants. This is shown in Figure 3.10. For state \(S_3\), it has two predecessors \(S_1\) and \(S_2\), which both contain constants. If \(Expr_a\) and \(Expr_b\) both are satisfiable then variable \(m\) is not a constant in state \(S_3\). Otherwise, if we can prove that one of \(Expr_a\) and \(Expr_b\) is never satisfiable, then it is sure that the other expression should be valid and the corresponding constant can be propagated. This example indicates that it is necessary to take special care of those states having more than one predecessor.
3.3 Optimizations

Figure 3.8: Workflow of internal-state constant propagation
Figure 3.9: Example of cross-state constant propagation

Figure 3.10: Conservative cross-state constant propagation
3.3 Optimizations

Here a simple and conservative method is adapted – constants are not to be propagated to the states having multiple predecessors. It means that, again, only a subset of the constants in the congruence – a partial congruence of a state is computed. While being simple, this conservative method is reasonable enough to be adapted. A detailed analysis is given in the next section.

The complete procedure of cross-state constant propagation can now be described as follows:

1. For all states of the EFSM, do the internal-state constant propagation. Then for each state collect the set of constants to be propagated to the successor states.

2. For each state, if the set of constants to be propagated to its successive states is not empty, try to propagate the constants to its successive states. For each successor of the state, if it has more than one predecessor, then it gives up the update.

3. If the updated partial congruences of the EFSM equal the previous congruences, stop the computation. Otherwise go to Step 1 for a new iteration.

Figure 3.11 shows the procedure.

EFSM Reduction

The final purpose of the task is to carry out possible EFSM reductions – to eliminate unreachable states and transitions as well as those unsatisfiable AGAs. The internal-state constant propagation has already identified those AGAs with guards evaluated to false. They can be safely eliminated as they are never executed. The partial congruences can also be used as constraints for checking the satisfiability of the transition conditions. Once the transition conditions to a state are identified as unsatisfiable, the state is unreachable and can be safely removed. This might lead to further unreachable states. Again a fixpoint computation can be established for eliminating the newly created unreachable states.

3.3.2 Invariant Inference

In the last section a simple optimization technique – constant propagation was discussed. This method does a per-state collection of constants and propagates them in to the successive states in a conservative way. Then an SMT solver can use this information for satisfiability checking of the guards and transition conditions to detect dead codes. In this method, the performance of the optimization depends on how many dead codes it is able to detect. This, in turn, depends on the number of constants found in the congruences of states. As more constants are found, more guards’ truth value
Set of initial congruences

For each state:
  Do Internal-State Constant Propagation
  Collect constants for successor states

Set of partial congruences
Set of constants for successor states

For each state:
  Do update of successor states
  For each successor state:
    If having multiple predecessors, give up the update

Set of updated partial congruences

Any update?

Yes

END

No

Figure 3.11: Workflow of cross-state constant propagation
could be fixed. Based on the same idea, a more powerful technique – *Invariant Inference* is developed in this section. The basic idea comes from the works in invariant detection [22] in program analysis. In invariant detection, methods are developed for calculating invariants in a program for better understanding of the program. These invariants could also be well suited for satisfiability checking. Anyway, here we’ll not calculate the invariants explicitly, but rather utilize them in an implicit way. We try to establish an equation system which encodes the program as well as the invariants of the program. Then the satisfiability of the target formulae can be checked under the context of the equation system.

There are more things needed to be clarified before we go into details of the method *Invariant Inference*. From Facts [19] and [20] it is clear that the core problem to be solved is to check the satisfiability of a guard. It is best to know that a guard is valid or unsatisfiable, since then the execution of the corresponding action is determined. It is of no help if a guard is satisfiable, since it gives no help in determining the execution of the action. In order to determine the satisfiability of the guards before runtime, the only information that can be utilized is from the program itself. Therefore the more information is collected, the satisfiability of the more number of guards can be determined. In Figure 3.12 an example shows that this is not an easy task.

![Figure 3.12: Example of the complexity of constant propagation](image)

By the EFSM given in Figure 3.12, it is clear that both *Initial State* and *S2* contain constants that might be propagated to their successor which in case to be the single state *S1*. Anyway this could not be done, otherwise there would be a write conflict in *S1*. But to forbid the constants to be propagated is however conservative. Since *Initial State* is the start of the execution, it is sure that the transition of *Initial State*→*S1* occurred at first. This means that the constant in *Initial State* is propagated to *S1* during the execution. If this constant propagation in turn triggers a chain reaction that can make some of the transitions following it unsatisfiable, more dead codes can be identified. In this example if the constant *m = 1* is propagated to *S1*, an SMT solver
can utilize this constant to deduce the fact \( c = \text{false} \), which means that the transition \( S1 \rightarrow S2 \) would never happen. Thus, the constant in \( S2 \) never had a chance to be propagated to \( S1 \).

This example shows that the order of execution should be considered when checking satisfiability for loops. In the following, an analysis similar to control flow analysis is employed for showing the complexity of satisfiability checking. Two basic structures – branch and perfect loop are discussed.

**Assertion Systems**

The execution of an EFSM can be illustrated by a graph. An EFSM can be seen as a directed graph which is simply called an EFSM graph. In this graph, the vertices are the states while the edges are the transition relations. A path in the graph corresponds to an execution sequence of the program. Since each edge is labeled with a transition condition, not every path corresponds to a valid execution of the program. A path containing edges that are unsatisfiable is called a fake path.

The satisfiability of a particular guard or transition condition of an EFSM is examined in the context of paths. A path gives a set of constraints to the formula being checked. Only in the context of path is the satisfiability of a formula meaningful.

**Definition 28 (Assertion System).** For a given EFSM \((s_0, S, T)\) with corresponding AGAs, graph \(g\) is the EFSM graph, \(p\) is a path of \(g\). The assertion system \(AS\) of \(p\) is the corresponding equation system extracted from \(p\). \(AS\) is generated as follows:

- For a given path \(p\): \(< s_1, s_2, \ldots, s_n >\) where \(s_i\) is the \(i\)th state in \(p\) and \(s_i \in S\), label each state in the sequence by the function \(L(s_i) = i\).

- Let \([\delta]^w\) be a substitution of a formula \(\delta\) by replacing the free variable \(x\) by \(y\). For each AGA \((\text{guard}, \text{lhs}, \text{rhs}, \text{assn})\) of state \(s_i\), rename the free variables occurred in it by substitutions:
  \[\text{guard}' = [\text{guard}]^w_{v_i}, \text{rhs}' = [\text{rhs}]^w_{v_i},\]
  \[\text{lhs}' = \text{if} (\text{assn} = \text{AssignNext}) \text{then} [\text{lhs}]^w_{v_{i+1}} \text{ else } [\text{lhs}]^w_{v_i}.\]
  for each transition \((s_i, c_i, s_{i+1})\), rename the transition condition \(c_i\) by \(c'_i = [c_i]^w_{v_i}\).

- For each renamed AGA \(k\) : \((\text{guard}', \text{lhs}', \text{rhs}', \text{assn})\), build an implication formula:
  \[AGA_k := \text{guard}' \rightarrow (\text{lhs}' = \text{rhs}')\] and for all renamed AGAs let \(\text{Conj}_i = \bigwedge_k AGA_k\) for state \(s_i\). For all transition conditions, let \(TC = \bigwedge_i c'_i\).

- Define the assertion system of \(p\) as: \(AS_p = (\bigwedge_{i=1,\ldots,n} \text{Conj}_i) \wedge TC\).

Given the definition of assertion system w.r.t. a particular path, the Facts 19 and 20 can now be stated in the sense of assertion systems:
Fact 29 (Satisfiability of an expression in a path). For an EFSM \((s_0, S, T)\), given a path \(p = < s_1, s_2, ..., s_n >\) and its assertion system \(AS_p\), the boolean expression \(e\) is either a guard of an AGA of state \(s_n\) or a transition condition of transition \(t := (s_n, e, s_{n+1})\) where \(t \in T\) and \(s_{n+1}\) is the successor of \(s_n\). \(e\) is satisfiable w.r.t. \(AS_p\) if there exists at least one model \(T\) such that \(T \models (AS_p \land e)\). \(e\) is valid w.r.t. \(AS_p\) if for any model \(T\) of \(AS_p\), \(T \models e\); \(e\) is unsatisfiable w.r.t. \(AS_p\) if for any model \(T\) of \(AS_p\), \(T \not\models e\).

By Definition 28 it is clear that for path \(p = < s_1, s_2, ..., s_n >\) and its assertion system \(AS_p\), \(AS_p\) is satisfiable means that \(s_n\) is reachable from \(s_1\). Under this constraint, if \(e\) as a guard in \(s_n\) is satisfiable, then Fact 19 is fulfilled. The same argument can be adopted when \(e\) is a transition condition for \(s_n\).

Branch Based Satisfiability Checking

The satisfiability checking in a branch-structured EFSM is relatively easy. Now the definition of a branch is given as follows:

Definition 30 (Branch-structures EFSM). Given an EFSM with \(g\) the corresponding EFSM graph, if \(g\) is acyclic then \(g\) is called a branch.

Figure 3.13 shows an example of a branch and all its paths going through state \(S4\).

Figure 3.13: Example of a branch and its paths:(a) a branch and (b) the paths going through \(S4\)
Optimization

It is obvious that for a branch, there exist finitely many paths in it. In Figure 3.13, the right hand side enumerates the three paths that end at state S4. As there are finitely many paths only, an obvious way to check the satisfiability of a guard within a branch is to check its satisfiability within every path that can reach the state it resides in.

Figure 3.14 shows an example of a branch-EFSM and its corresponding assertion system. This method will be discussed in more detail later in Section 3.3.2.

Loop based Satisfiability checking

An EFSM graph can be very unstructured in the sense that it contains complex loop structures. This is quite probable when there are abortion or suspension statements nested in loop statements. Loop structures are more difficult to analyze in the context of paths, since once a loop is introduced there might exist infinitely many possible paths in it. In this section only the simplest loop structure – a basic loop is discussed. Anyway from the discussion one can see that even for the simplest structure, the analysis would already be too complex to be carried out.

Definition 31 (Basic Loop). Given an EFSM, g is the EFSM graph. Define an end state of g to be a state with no successors. g is said to be a basic loop if it contains a unique end state, a unique strongly connected component within which includes one state, or the entry state, having in-degree being 2, and one state, or the exit state, having out-degree being 2, and all the other states of the strongly connected components having both in-degree and out-degree being 1.

There is a unique path from the initial state to the entry state, being the entry path, and one unique path from the exit state to the end state, being the exit path. The
strongly connected component forms the only loop. The path starts from the entry state enumerating all states in the strongly connected component to the predecessor of the entry state is called the loop path. If the loop is a self-loop with only one state \( s \), the loop path is then the path \( < s, s > \).

Figure 3.15 shows a basic loop and its corresponding entry path, loop path and exit path.

![Figure 3.15: Example of a basic loop with: (a) the basic loop and (b) its entry path, loop path and exit path](image)

By Definition 9 it is clear that if the basic loop is a well-defined EFSM, then all the transitions except for the transitions from the exit state should have the transition condition to be true. In the above example, the only two transitions that have non-true transition conditions are \( S3 \rightarrow S5 \) and \( S3 \rightarrow S4 \). Given all these definitions and facts, the satisfiability of guards and transition conditions can now be discussed as follows:

### Satisfiability in Entry Paths

Checking the satisfiability of a guard in the entry path is simple, because the entry path is the only path that could go through the state which contains the guard. For simplicity, the path from the initial state to the state that contains the AGA with the guard is called the entry path w.r.t. the guard, or the entry path as well. Let the assertion system of the
entry path be $AS_p$. The following table gives the satisfiability checking of the guard in its entry path:

<table>
<thead>
<tr>
<th>$AS_p \land \text{guard}$</th>
<th>SAT</th>
<th>UNSAT</th>
<th>VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td>guard</td>
<td>SAT</td>
<td>UNSAT</td>
<td>VALID</td>
</tr>
</tbody>
</table>

The first row of the table lists the satisfiability of the formula $(AS_p \land \text{guard})$, and the second row of the table lists the satisfiability of $\text{guard}$. It is clear that the entry path of the guard is the earliest executed path and the only path that go through the guard. In fact this path can be seen as a simple branch. The satisfiability checking here can then be reduced to a branch-based satisfiability checking.

**Satisfiability in Loop Paths**

Checking the satisfiability of a guard in a loop path is more complex, because the execution order of paths needs to be considered. A state inside a loop path can be reached in two ways: either by a first execution of the loop path – originated from the initial state, or by an $n$th execution of the loop path. Considering these two cases as different cases is important. From Fact 19 it is clear that if the guard is satisfiable, the corresponding state must be reachable. To check the reachability of the state, one should check additionally the path that starts at the initial state and goes all the way to the state in the loop path, i.e. the first-reached path, rather than the loop path itself only.

**Definition 32** (First-reached path). Given a basic loop and a guard of a state $s$ in the basic loop, $g$ is a guard of an AGA of $s$. The first-reached path $p$ of $g$ is the path starts from the initial state to the state $s$ in which no state is repeatedly visited.

In Figure 3.15 the first-reached path of state $S_4$ is $<S_1,S_2,S_3,S_4>$. To check the satisfiability of a guard in state $S_4$, one should first consider if the guard’s first-reached path is satisfiable. If it is not satisfiable, it means that $S_4$ is not reachable. Thus the execution of the program can only be $<S_1,S_2,S_3,S_5>$. If it is satisfiable, the satisfiability of the guard can then be checked against the first-reached path. Assume that the assertion system of the first-reached path is $AS_{fp}$, the assertion system of the loop path is $AS_{lp}$. The following table gives the satisfiability checking of a guard in a loop path:

<table>
<thead>
<tr>
<th>$AS_{fp}$</th>
<th>UNSAT</th>
<th>SAT</th>
<th>UNSAT</th>
<th>SAT</th>
<th>VALID</th>
<th>VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AS_{lp} \land \text{guard}$</td>
<td>-</td>
<td>UNSAT</td>
<td>SAT</td>
<td>SAT</td>
<td>VALID</td>
<td>SAT</td>
</tr>
<tr>
<td>$AS_{fp} \land \text{guard}$</td>
<td>-</td>
<td>UNSAT</td>
<td>SAT</td>
<td>SAT</td>
<td>VALID</td>
<td>SAT</td>
</tr>
<tr>
<td>$\text{guard}$</td>
<td>UNSAT</td>
<td>UNSAT</td>
<td>?</td>
<td>SAT</td>
<td>VALID</td>
<td>?</td>
</tr>
</tbody>
</table>
In the table, from top to bottom, the four rows list the satisfiability of the formulae $AS_{fp}$, $AS_{fp} \land guard$, $AS_{lp} \land guard$ and $guard$. “guard” is the guard of an AGA in a state in the loop path. “−” means the satisfiability of the item can be any of the three and “?” means that the satisfiability is unknown. It is clear that when the $AS_{fp}$ is unsatisfiable, the guard is also unsatisfiable, since the corresponding state in which the guard resides is not reachable. Even if $AS_{fp}$ is satisfiable, it is not guaranteed that the guard is satisfiable – the guard still has to be checked against $AS_{fp}$. If the formula $(AS_{fp} \land guard)$ is unsatisfiable, it means that the guard is not satisfiable in the first round of the loop. However if the formula $(AS_{lp} \land guard)$ is also not satisfiable, then it is sure that the guard is never satisfiable.

The tricky point is that the unsatisfiability of the two formulae $(AS_{fp} \land guard)$ and $(AS_{lp} \land guard)$ is a sufficient condition of the unsatisfiability of the guard. In other words, knowing that guard is unsatisfiable, it is not sure that the formula $(AS_{lp} \land guard)$ is also unsatisfiable. To understand the reason, it is necessary to have a deeper look into the meaning of the satisfiability of the assertion system $AS_{fp}$. The satisfiability of the loop path’s assertion system means that there exist models of the assertion system. These models are all the possible executions’ instances of the loop. Anyway since the loop path does not take the beginning condition into consideration, it might be possible that the models include some execution instances that are never to be executed in the real run of the program. For example in Figure 3.16.
It is clear that the loop path will only execute based on the condition declared in the second line of the program (a): \( i < 5 \). In the assertion system of the loop path it is obvious that \( i \_S1 \) and \( i \_S2 \) can be any number less than \( 5 \), while in the first-reached path \( i \_S1 \) is fixed to 3 and \( i \_S2 \) is fixed to 4. The models of the assertion system of first-reached path are restricted to the first two rounds of the loop’s execution, while the models of the assertion system of loop path consider more than enough execution instances.

Due to the imprecision of the assertion systems explained above, the satisfiability checking of the guards incomplete. When the formula \((AS_{fp} \land guard)\) is unsatisfiable but \((AS_{lp} \land guard)\) is satisfiable, it is not guaranteed that the guard is satisfiable, since we do not know that if the models that made the second formula satisfiable are ever real execution instances at run time! The same argument applies for the case when \((AS_{fp} \land guard)\) is valid but \((AS_{lp} \land guard)\) is satisfiable, since it is also not clear that whether those models that made the second formula unsatisfiable can really be carried out in the run time. For the case when \((AS_{fp} \land guard)\) is satisfiable, it is clear that the guard is satisfiable, because the guard is already satisfiable in the first round of the loop’s execution. For the case that both \((AS_{fp} \land guard)\) and \((AS_{lp} \land guard)\) are valid it is also guaranteed that the guard is valid, because the validity of the two formulae has covered all possible execution instances of the loop.

Another way that might worth trying is a reachability computation. By formulating the program into a model-checking problem, one can test the reachability of a specified state. In our context, this state should be the state where the guard holds (for testing the guard’s satisfiability). In order to check that the guard is valid, it is necessary to verify that the guard holds in all reachable states. Anyway it is also too complex to be implemented.

**Satisfiability in Exit Paths**

Checking the satisfiability of a guard in an exit path is as difficult as checking the satisfiability in the loop path, since there are different cases that a state in the exit path can be reached, and each case needs to be taken care of.

**Definition 33** (Loop Reached Path). *For a given basic loop of an EFSM, state \( s \) is a state in the exit path. \( grd \) is a guard of AGA of \( s \). The loop-reached path of \( grd \) is the path starting from the exit state of the basic loop, enumerating through all the states in the loop path, then reaching the exit state again and enumerating the exit path until state \( s \).*

Assume the assertion system of the loop-reached path is \( AS_{lrp} \). The table follows gives the satisfiability checking of a guard in an exit path:
### 3.3 Optimizations

<table>
<thead>
<tr>
<th>( AS_{fp} )</th>
<th>( UNSAT )</th>
<th>( SAT )</th>
<th>( SAT )</th>
<th>–</th>
<th>–</th>
<th>–</th>
</tr>
</thead>
<tbody>
<tr>
<td>( AS_{fp} \land guard )</td>
<td>–</td>
<td>( UNSAT )</td>
<td>( UNSAT )</td>
<td>( SAT )</td>
<td>( VALID )</td>
<td>( VALID )</td>
</tr>
<tr>
<td>( AS_{lrp} \land guard )</td>
<td>–</td>
<td>( UNSAT )</td>
<td>( SAT )</td>
<td>–</td>
<td>( VALID )</td>
<td>( SAT )</td>
</tr>
<tr>
<td>( guard )</td>
<td>( UNSAT )</td>
<td>( UNSAT )</td>
<td>?</td>
<td>( SAT )</td>
<td>( VALID )</td>
<td>?</td>
</tr>
</tbody>
</table>

An argument similar with the “Satisfiability in Loop path” can be applied for the table above. The detailed analysis is omitted here for the sake of avoiding repeated contents.

An alternative way dealing with loops (not implemented in this thesis) is loop unrolling. By loop unrolling, the loops in the programs are unrolled several times to form a sequential piece of code. Then for this particular sequential program the properties of interest can be examined easily. A disadvantage of this method is that the result only applies for the specified number of unrolling. Beyond this number of unrolling nothing is guaranteed.

### Complex Cases

Usually a realistic program contains loops that are more complex than a perfect loop. For simplicity the state in the strongly connected component with incoming edges outside the component is still called an *entry state*, and the state with out going edges outside the component is still called an *exit state*. In the corresponding EFSM graph, the strongly connected component may have multiple predecessors / successors for the entry state / exit state as well as multiple entry states / exit states. This often makes the situation much more complex. Consider the example in Figure 3.17. In the example (a) shows a part of an EFSM graph, where the entry state \( S_3 \) has two predecessors and the strongly connected component has two exit states. This gives rise to the four different executions of the loop as listed in (b) of the figure. Given these four cases of executions, it is necessary to consider their order of execution, since an earlier execution of a case might disable guards of other cases of executions and make them never executed. It is even worse when the structure of the strongly connected component is more complex, which means that there are more cases to be considered. Loop unrolling is also helpless in this situation, since the strongly connected component is more unstructured.

### Procedure

The implementation of Invariant Inference is based on the branch-based satisfiability analysis. This implementation tries to divide an EFSM graph into partitions such that each partition forms a branch. Then a single assertion system can be derived from the branch. The question of satisfiability can be answered within this assertion system easily.
Definition 34 (Extended Basic Block). (EBB) An extended basic block of an EFSM graph \( g \) is a sub-graph of \( g \), which forms a tree \( t \). The root of \( t \) is either the initial state of \( g \) or a vertex with in-degree > 1. All the other nodes in \( t \) should have in-degree = 1 and the successors of the leaves either have out-degree = 0 or have in-degree > 1.

The concept of extended basic block is also useful in control flow analysis of sequential programs [31]. Figure 3.18 shows an example of an EFSM graph and its corresponding extended basic blocks.

The extraction of EBBs is relatively easy. Through a breath-first search of the EFSM graph the corresponding EBBs can be derived. During the breath-first search, a set of EBBs is maintained. The search started from the initial state of the graph. For each node we first check whether the node is the initial state of the graph or its in-degree is larger than 1. If any of the two conditions holds, by Definition 34 the node should be a root of a new EBB. In this case, a new EBB structure is created, and the node is set as the root of the EBB. Then the EBB is put into the set of EBBs. If not, it is just a child node of some existing EBB. In this case, the node is added to the corresponding existing tree of EBB. By the end of the breath-first search, we should have all the EBBs of the graph.

Once all the EBBs are collected, the corresponding assertion systems can be built based on them.
Definition 35 (Assertion System of an Extended Basic Block). For a given EFSM \((s_0, S, T)\) graph \(g\) with its corresponding AGAs, \(B\) the set of extended basic blocks, \(b\) an extended basic block in \(B\). The assertion system of \(b\) is the corresponding equation system \(AS\) extracted from \(b\). \(AS\) is generated as follows:

- Let the sequence: \(< s_1, s_2, ..., s_n >\) be the corresponding order of the breadth-first searching of \(b\). Label each state in the sequence by the function \(L(s_i) = i\).
- For each AGA : \((\text{guard}, \text{lhs}, \text{rhs}, \text{assn})\) of state \(s_i\), rename the free variables occurred in it by substitutions:
  \(\text{guard}' = [\text{guard}]_{v_i}, \text{rhs}' = [\text{rhs}]_{v_i}, \text{lhs}' = \text{if (assn} = \text{AssignNext}) \text{then [lhs]}_{v_{i+1}} \text{ else [lhs]}_{v_i}\).
  If the AGAs of the final state in the path contain AssignNext statements, then the written variable \(\text{lhs}\) of the corresponding AGA is renamed by \(\text{lhs}_{end}\). For each transition \((s_i, c_i+1, s_{i+1})\), rename the transition condition \(c_{i+1}\) by \(c'_{i+1} = [c_{i+1}]_{v_i}\).
- For each renamed AGA_k : \((\text{guard}', \text{lhs}', \text{rhs}', \text{assn})\), build an implication formula: \(\text{AGA}_k := \text{guard}' \rightarrow (\text{lhs}' = \text{rhs}')\) and for all renamed AGAs let \(\text{ Conj}_k = \bigwedge_{k} \text{ AGA}_k\) for state \(s_i\).
- For each leaf node \(s_d\), let \(\text{AS}_d = \text{ Conj}_d\); for each non-leaf node \(s_e\) of \(b\), let the successors of \(e\) be \(\{s_i, s_j, ..., s_r\}\) and \(\{c'_i, c'_j, c'_r\}\) be the corresponding renamed transition conditions. Let \(\text{AS}_e = (\bigvee_{m=i, j, ..., r}(\text{AS}_m \land c'_m)) \land \text{ Conj}_e\). 

Figure 3.18: Example of EBB: (a) an EFSM graph, (b) the extended basic blocks
Let \( s_r \) be the root of \( b \), define the assertion system \( AS \) of \( b \) as: \( AS_r \).

It is not difficult to see that each EBB is a branch. The satisfiability checking can be done similarly as the way discussed in branch based satisfiability checking. The only difference here is that instead of doing a satisfiability checking with the assertion system of each path within a branch, we do it with a complete assertion system built for the whole branch.

After satisfiability checking with each EBB, there are possibly nodes and edges that can already be eliminated. Thus, a fixpoint computation can be carried out for the satisfiability checking until there is no more change of the EFSM graph. The procedure of the fixpoint computation is shown in Figure 3.19.

From Figure 3.19 it is shown that the satisfiability checking for the transition conditions is performed first, then the checking of the guards of AGAs is executed after the shape of the EFSM graph is fixed. The reason is that by Fact 19 that for each executable AGA, the state it resides in must be reachable in the first hand. If this state is even not reachable, then there is no need to check its satisfiability. This helps us to design a more efficient implementation.

Compared with constant propagation, invariant inference is a more powerful method and demands more computation. In practice it is up to the programmer’s choice which technique to use. Depending on how large the size of his or her programs would be, he or she might choose constant propagation for a faster and limited optimization or choose invariant inference for a slower but more precise optimization.

### 3.3.3 Identification of Passive Code

In Section 2.1 the synchronous model of computation is introduced. This model of computation demands that for every reaction (i.e. macro-step) the inputs and outputs of the reactive system are deterministic. For the computation of the outputs there could be many local variables defined. These local variables must also be synchronized during each reaction, even some of them might not be needed for the computation of the outputs. Consider the example shown in Figure 3.20. From the last two AGAs in state \( S_1 \) it is clear that the output depends on the four variables \( a, b, c \) and \( d \) as well as the boolean variable \( op \). However, only one of the two AGAs can be executed. From the initial state it is known that \( op \) is set to \( true \) in state \( S_1 \). Thus the last AGA is not executed. This fact infers that variable \( c \) and \( d \) are not needed and the computation of these two variables can be safely eliminated.

The data dependency analysis introduced in Section 3.1 has provided us some useful operations like the computation of read and write dependencies in Definition 16. We can use these operations to compute for each state of the EFSM a set of required
3.3 Optimizations

Figure 3.19: Fixpoint computation of invariant inference
variables, i.e. the set of variables required in this state to be able to compute the outputs. In the following some auxiliary functions are defined:

- **UseNowVars** \( (s) \) := \{ x \in V | \exists \alpha \in GrdAct(s). \exists y \in reqvars_s, y \in wrVars(\alpha) \land x \in rdVars(\alpha) \} 

- **UseNxtVars** \( (s) \) := \bigcup_{s' \in suc(s)} \{ x \in V | \exists \alpha \in GrdAct(s). \exists y \in reqvars_{s'}. next(y) \in wrVars(\alpha) \land x \in rdVars(\alpha) \} 

- **DefNowVars** \( (s) \) := \{ x \in V | Check(\bigvee_{(grd,lhs,rhs,AssignNow) \in GrdAct(s)} grd) \} 

- **DefNxtVars** \( (s) \) := \{ x \in V | Check(\bigvee_{(grd,lhs,rhs,AssignNext) \in GrdAct(s)} grd) \} 

- **StoreVars** \( (s) \) := \{ x \in MemVars(V) | x \notin DefNxtVars(s) \land \exists s' \in suc(s). x \in reqvars_{s'} \land x \notin DefNowVars(s') \} 

where:

- **UseNowVars** \( (s) \) contains the variables \( x \) that are read by an action that immediately writes a required variable \( y \) in state \( s \).

- **UseNxtVars** \( (s) \) contains the variables \( x \) that are read by an enabled delayed assignment of state \( s \) to a variable \( y \) that is required in a successor state \( s' \) of \( s \).

- **DefNowVars** \( (s) \) is the set of variables having a guarded action with an immediate assignment that must become true in state \( s \) (‘must become true’ is estimated by the test procedure \( Check(\phi) \) below).

- **DefNxtVars** \( (s) \) is the set of variables having a guarded action with a delayed assignment that must become true in state \( s \) (again, ‘must become true’ is estimated by the test procedure \( Check(\phi) \) below).
• *Check*(\(\phi\)): In principle, the previous two properties would be the disjunction of guards. *Check*(\(\phi\)) conservatively approximates \(\phi\): it will only return true if \(\phi\) is true; if the heuristic cannot determine a value, it conservatively returns false. The precision of this function depends on the dead-code elimination procedure. If dead code elimination has been performed before, it is possible that more guards are identified to be valid, and thus changed to true.

• *StoreVars*(\(s\)) is the set of memorized variables that must be stored in state \(s\) due to the reaction of absence, i.e., the memorized variables \(y\) that are not defined in \(s\) due to an enabled delayed assignment, but that are required in a successor \(s'\) of \(s\) where \(y\) is also not immediately defined.

Given the auxiliary functions, the set of required variables can now be defined as the least fixpoint of the following equation system:

**Definition 36 (Required Variables).** Given \(V_O\) the set of output variables, the following system of mutually dependent fixpoint equations describes the desired \(reqvars_x\) sets for the EFSM states \(\{s_1, s_2, \ldots, s_n\}\):

\[
\begin{align*}
reqvars_{s_1} &\overset{\mu}{=} UseNowVars(s_1) \cup UseNext(s_0) \cup StoreVars(s_1) \cup V_O \\
&\vdots \\
reqvars_{s_n} &\overset{\mu}{=} UseNowVars(s_n) \cup UseNext(s_n) \cup StoreVars(s_n) \cup V_O
\end{align*}
\]

Based on the required variables, two kinds of AGAs can be eliminated. For a state \(s\):

\{(grd, lhs, rhs, AssignNow)|AGA \in GrdAct(s), lhs \notin reqvars\}

\{(grd, lhs, rhs, AssignNext)|AGA \in GrdAct(s), \forall s' \in Succ(s), lhs \notin reqvars_{s'}\}

The first set of AGAs computes the values of variables that are not used in the current reaction. The second set of AGAs computes the values of variables that are not used in the next reaction. The identification of passive code can be performed without dead code elimination, but as shown in the next chapter, better performance can be achieved if it is carried out after dead code elimination.

### 3.3.4 Array Optimization

Array-intensive computations usually cost the most of time and space. Many optimization techniques have been developed concentrated on the computation of arrays. The basis of many optimizations is the array reference testing [28, 44, 30]. However, previously the techniques invented for array reference testing are not complete, since they only test special properties of the dependency relationships. This makes the methods necessary conditions for the existing of dependencies. In Section 2.3 it is already
shown that array reference testing can be formulated in the form of a satisfiability-checking problem. In this section, an SMT-based array reference testing method for EFSM is introduced. Based on this method, an array normalization procedure is implemented. Generally, this procedure detects those equal array references and changes them to syntactically the same form. Based on this procedure it is shown that the optimization of identification of passive code is able to achieve better performance.

**Basic Idea**

An array assignment usually has more unfixed variables than normal assignments, since except for the free variables of the right-hand-side expression and the variable indicating the name of the array, the array index is possibly also a variable. Thus, even if the assigned value can be fixed, it is usually not known which element of the array it is assigned to. If the semantically equivalent array accesses can be identified, they can be normalized to the syntactically equivalent normal forms. Based on the syntactical normalization many further optimizations can be performed.

As array elements are memorized variables, a deterministic assignment in a previous state would not help in the current state since the same array element that has been fixed might be assigned to a new value this time. To make sure that the assigned element in the current state is not the previously fixed element (so that one can be sure that the previously assigned array element retains its value), one has to make sure that the two array elements do not have the same index, or the guard of the current assignment is unsatisfiable. Instead of considering the propagation of array values between EFSM states, in this section we only concentrates on the array reference testing within a particular EFSM state.

Another issue is that sometimes array elements of different arrays could share the same content. Checking the equal array elements of two different arrays needs more computation. For two arrays having \( m \) and \( n \) elements respectively, \( m \times n \) comparisons are needed. As this rarely happens, we will save the effort for checking equality of those elements in the same array.

**SMT based array reference testing**

The SMT based array reference testing presented here is based on the invariant inference introduced in Section 3.3.2. The reason is that once the assertion systems are established, they automatically take the array expressions into consideration. To check the equivalence of the array elements one only needs to add corresponding assertions and do a satisfiability checking.
### 3.3 Optimizations

#### Array Element Normalization

Two syntactically different array accesses might have the same value. SMT can help us detecting this fact. Once they are identified to be always equal, we can rewrite one of them to the other, which in turn making them syntactically equalized. If they are identified to be equal under some cases or even not equal, their syntactical form should be remained. Anyway for those unequal array accesses, they should be recorded as well. This can be a basis for many further optimizations like:

- **For Identification of Passive Code** For example:
  
  \[ \text{A}[f] = T1; \]
  \[ Y = \text{A}[g]; \]

  Let \( Y \) be needed. If we detect that \( A[f] \neq A[g] \) then it is safe to say that the computation of \( A[f] \) is not necessary.

- **For more precise determination of satisfiability of guards.** If an *invariant inference* procedure is done, then there’s no need to do this step. Else, this step might help deciding guards of the form: \( A[f(...)] < A[g(...)] \).

- **For eliminating redundant computations.** For example:
  
  \[ \text{A}[f] = T1; \]
  \[ \text{A}[g] = T2; \]

  Once it is identified that \( A[f] = A[g] \) and there is no write confliction, \( T1 \) or \( T2 \) can be eliminated safely, since they must evaluate to the same value.

#### Procedure

The procedure of array normalization within an EFSM state is relatively simple. Let the set of arrays of an EFSM be \( A \). For a given state of the EFSM:

1. For each \( \text{array}_i \in A \), generate the set of all array accesses appeared in the AGAs: \( S_{\text{array}_i} \).
2. For each \( \text{array}_i \in A \), check the equivalence of the array accesses using an SMT solver and the corresponding assertion system. Then group them in equivalence groups \( G_{eq}^k \subseteq S_{\text{array}_i} \) where: \( \forall a_i, a_j \in G_{eq}^k \), \( a_i \equiv a_j \) and \( \forall a \in G_{eq}^k \), \( b \in G_{eq}^l \), \( k \neq l \), \( a \neq b \).
3. For each equivalence group, choose randomly a normal form and map all other accesses to it. Then put all normal forms to set \( G_{ie} \), representing the inequivalence group.
4. Replace the array accesses by their normal forms in the AGAs of the given state.
3.3.5 Correctness of the Optimization

**Definition 37** (Correctness of the Optimization). Let $P$ be any AIFS\textit{System} structure. Let $Q$ be the optimized AIFS\textit{System} structure of $P$. An optimization is correct if only for any assignment of $P$’s input variables, the outputs computed by $P$ and $Q$ are exactly the same.

This definition indicates that for any correct optimization, the optimized program still computes the same result for the same input, just as the program before optimization does.

**Correctness of Dead Code Elimination**

The correctness of the Dead Code Elimination is based on the Facts \[18\] \[19\] and \[20\]. Since a guarded action is never executed when its guard is invalid, the elimination of the guarded action does not have any effect on the computation of any input. The same argument applies also for the elimination of unreachable EFSM states.

**Correctness of Passive Code Elimination**

The correctness of the Passive Code Elimination can be proved by analyzing the assistant functions defined in Section \[3.3.3\]. First, if a local variable is not required for the computation of any output variables, then the guarded action that computes the local variable can be safely eliminated. Then by analyzing the computations of the assistant functions it is clear that the set of required variables computed is a sub-set of the complete required variables, i.e. the fixpoint computation of Definition \[36\] is an approximation. For more details of the proof, the reader can refer to \[11\].
4 Experimental Results

This chapter gives the results of this thesis. The first section gives some small examples, which illustrate the effects and potential of the optimization techniques presented in the previous chapter. The second section depicts some results in running the optimizations on practical examples, which give a preliminary impression on the optimizations’ practical usage.

The implementation of the algorithms introduced in this thesis is based on the .Net Framework 4.0.30319. The programming language of implementation is F# 2.0.0.0, with assistant libraries from F# PowerPack 1.9.9.9. A set of Averest 1.9.9.10 libraries (rewritten in F#) is utilized for the support of the AIFSSystem related functions and translations of the expressions. For the satisfiability checking, a set of .Net managed APIs (packed in a .dll file) implementing the functions of the SMT solver Z3 are used. The hardware platform is an ASUS laptop with the CPU Intel Duo T7500 and 2GB memories.

Five modules are implemented as F# libraries, which could as well be integrated into Averest libraries:

<table>
<thead>
<tr>
<th>library</th>
<th>functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDD.dll</td>
<td>Efficient boolean operations implemented in Binary Decision Trees</td>
</tr>
<tr>
<td>EFSSM.dll</td>
<td>Generation of the Extended Finite State Machines</td>
</tr>
<tr>
<td>SMTSolving.dll</td>
<td>Checking satisfiability of Z3 formulae</td>
</tr>
<tr>
<td>InvarInference.dll</td>
<td>Generation of assertion systems and performing invariant inference</td>
</tr>
<tr>
<td>IDPassCode.dll</td>
<td>Performing identification of passive codes</td>
</tr>
</tbody>
</table>

4.1 Small Experiments

In this section experimental tests are shown for illustrating the functions of the optimizations. With each example, first the Quartz program is listed, then the original EFSSM and corresponding optimized EFSSM are depicted.

Internal-State Constant Propagation

Consider the following Quartz program:
module Test_ISCP(event !o) {
    nat x, y;
    x = 0;
    if(x < 3) {
        l1: pause;
        x = 4;
        y = x + 1;
    } else {
        l2: pause;
        x = 5;
        y = x - 1;
    }
    if(x > 10) o = true;
}

Figure 4.1 shows the corresponding EFSMs. In the figure it is shown that in (b) State 2

is kept while State 1 is eliminated. This is because the optimizer checks the fact $x = 0$
in the initial state, which makes the transition condition $!(x < 3)$ invalid.

Cross-State Constant Propagation

Consider the following Quartz program:

module Test_CSCP (int{256} ?a, ?b, s) {
    int{256} i1, i2, o, t1, t2;
    bool op;
    loop {
        w1: pause;
        t1 = i1 + i2 ;
        t2 = i1 * i2 ;
    }
if ( op ) o = t1; else o = t2;
}
||
loop {
    w2: pause;
    i1 = a;
    i2 = b;
    op = false;
    next(i1) = o;
    next(i2) = s;
    next(op) = true;
    w3: pause;
    s = o;
}
}

Figure 4.2 shows the corresponding EFSMs. In the figure it can be seen that in State 1

(a) (b)

Figure 4.2: Example of cross-state constant propagation: (a) the original EFSM of the program Test_CSCP, (b) the optimized EFSM

the AGA \( op \Rightarrow o = t2 \) in (a) is transformed to \( True \Rightarrow o = t2 \) in (b) and the AGA \( op \Rightarrow o = t1 \) is eliminated. These changes are due to the fact \( op = false \). Moreover, in State 2 the AGA \( op \Rightarrow o = t1 \) of (a) is replaced by \( True \Rightarrow o = t1 \) in (b) and the AGA \( !op \Rightarrow o = t2 \) is eliminated. These changes are due to the constant \( op = true \) propagated from State 1.
Branch-based Invariant Inference

Consider the following Quartz program:

```quartz
module Test_InvInf (int{256} ?a, ?b, s) {
    bool op;
    int{256} t1, t2, t3;
    w1: pause;
    {
        next(op) = true;
        if(op) t1 = 3; else t1 = 5;
        w4: pause;
        if(op) t3 = 11; else t3 = 22;
    }
    ||
    |
    |
    if (t1 > 10) {w2:pause; next(t2) = 1;}
    else {w3:pause; next(t2) = 5;}
    w5: pause;
    if (t2 > 4) t3 = t2;
    if(t3 > 10) {w6:pause; next(t2) = 11;}
    else {w7:pause;next(t2) = 22;}
}
w8: pause;
if (t2 < 20) {t3 = 1;}
}
```

Figure 4.3 shows the corresponding EFSMs. For comparison, the figure also shows the result of a cross-state constant propagation in (b). It can be seen from (b) that State2, State3, as well as State7 are optimized. For State2 and State3 it is because the constant op = true is propagated from their predecessor State1. For State7 it is due to the reactions to absence. In (c), it can be seen that the structure of the EFSM is dramatically changed: both State2 and State5 are eliminated. The two AGAs !op => t1 = 3 and op => t1 = 5 imply that the transition condition 10 < t1 of State1 → State2 is invalid. In this case, the value of t2 can also be fixed in State4. This, in turn, can be used in State4 to deduce the fact t3 = t2 = 5 is valid. Thus, the transition condition 10 < t3 of State4 → State5 can be identified as false and State5 recognized as an unreachable state. Finally in State7 the fact t2 < 20 => t3 = 1.

Dead code elimination can be very helpful in optimizing loops. The following piece of code shows a nested loop structure.

```quartz
module Test_Loop (int{16} ?a, int{16} !c) {
    int{16} b;
    int{16} i, j, k;
    i = 0; j = 0; k = 0;
```
Figure 4.3: Example of cross-state constant propagation: (a) the original EFSM of the program Test_InvInf, (b) the EFSM optimized by cross-state constant propagation, (c) the EFSM optimized by invariant inference.
while (k < 4) {b = a + 1; next(k) = k+1; l0:pause;}
while (i < 4) {
    j = 0;
    while (j < 4){
        b = a + 10;
        if(b < 3) k = j + 1;
        next(j) = j + 1;
        l1:pause;
    }
    next(i) = i + 1;
    l2:pause;
}
c = b;

Figure 4.4 shows the initial state of the EFSM of the program Test_Loop.

Figure 4.4: The initial state of the EFSM of the program Test_Loops

A loop in a program usually has an initial condition for the iterations. For example, in the program Test_Loop, all the iteration variables i, j and k are set to 0. This information is encoded in the guarded actions of the initial state. However, when generating the EFSM, the case distinction procedure is done in the level of propositional logic, which means that the initial condition is not considered. In this case there
might be many redundant guarded actions as well as transition conditions as shown in Figure 4.4. By using the dead code optimization techniques of this thesis, these unsatisfiable guarded actions can be identified and eliminated. Figure 4.5 shows the original EFSM structure and the optimized EFSM structure of the program Test_Loop.

![Figure 4.5: Optimizing loops: (a) the original structure of the EFSM of program Test_Loops and (b) the optimized structure of the EFSM](image)

Passive Code Elimination

The original and optimized EFSMs of the program Test_InvInf is shown in Figure 4.6.
Figure 4.6: EFSMs of the program Test_InvInf: (a) the original EFSM, (b) the EFSM optimized using invariance inference, (c) the EFSM optimized using passive code elimination based on (b) and (d) the EFSM optimized using passive code elimination based on (a)

The different results of (c) and (d) shows an interesting phenomenon – passive code elimination often achieves better performance with dead code elimination. The reason might be that originally there are many guarded actions considering unsatisfiable cases. Without a dead code elimination procedure, these unsatisfiable cases bring more data dependencies. By considering the redundant data dependencies, the passive code elimination procedure can not be fully utilized.

Array Optimization

Consider the following Quartz program:

```quartz
module Test_Array (int{16} ?a, int{16} !c) {
  int{16}[3] b;
  nat{5} i, j;
  i = 0; j = 0;

  b[i] = a; b[j] = b[i];
  next(j) = j+1;
  next(i) = 0;
  w0:pause;
```
4.1 Small Experiments

\[
b[j] = b[i] + 1; \\
\text{next}(j) = j + 1; \\
\text{next}(i) = 1; \\
\text{w1:pause;} \\
b[j] = b[i] + b[i+1]; \\
c = b[j]; \\
\]

Figure 4.7 shows the original and the optimized EFSMs respectively.

In Section 3.3.4 a normal form is randomly chosen for the equivalent array accessions. However, the example in Figure 4.7 shows that different choices might influence the performance of the optimization. In Figure 4.7 (b) \( b[i+1] \) is chosen as the normal form in State 2, and in (c) \( b[j] \) is chosen to be the normal form in State 2. The procedure of passive code elimination produces the same EFSM based on (c), but reduced a guarded action in State 1 of (b). The reason is that array normalization only considers the situations within a state while passive code elimination depends mostly on cross-
state data dependencies. If the normal form is improperly chosen, the data dependency between succeeding states might still remain. Anyway, with an appropriate normalization strategy, it is shown by the example that passive code elimination can achieve a better result.

4.2 Studies

This section shows the results achieved for larger programs, which are taken from the set of Averest benchmarks. The tested files are all of AIFSystem format. The original AIFSystem files are compiled from the corresponding Quartz programs respectively. These original AIFSystem files are then treated as inputs for the optimization. After optimization, the AIFSystem files are generated. Figure 4.8 gives an overview of the workflow.

As can be seen from Figure 4.8, the final AIFSystem files are generated in three ways:

1. with no optimization;
2. with **dead code elimination**

3. with **dead code elimination** and **passive code elimination** together.

*Invariant Inverence* (IVIF) and *Identification of Passive Code* (IDPC) are chosen to represent the **dead code elimination** and **passive code elimination** respectively.

The comparison is done based on the size of the output *AIFSystem* files. If the optimized version has smaller size than the version with no optimization, then we have a positive result. The reason to choose size rather than run-time as the metric of comparison is that the run-time of the programs varies as inputs vary. It is difficult to test all possible inputs for a program. Meanwhile, the reduction of the size should lead to a reduction of time, since the computations of the program are reduced. It is also noticeable that the optimizations introduced in this thesis won’t increase the size as the procedures only do eliminations.

The files with no optimization have different sizes ranged from dozens of KBs to dozens of MBs. The following table shows the **test results**:

<table>
<thead>
<tr>
<th>Tested Program</th>
<th>Original Size</th>
<th>IVIF</th>
<th>IVIF+IDPC</th>
<th>Reduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car Velocity</td>
<td>193KB</td>
<td>145KB</td>
<td>145KB</td>
<td>24.8%</td>
</tr>
<tr>
<td>Island Traffic Control</td>
<td>60,863KB</td>
<td>3,407KB</td>
<td>3,407KB</td>
<td>94.4%</td>
</tr>
<tr>
<td>Light Control System</td>
<td>25KB</td>
<td>25KB</td>
<td>25KB</td>
<td>NO</td>
</tr>
<tr>
<td>Logical Sensors v1</td>
<td>33,712KB</td>
<td>33,188KB</td>
<td>30,915KB</td>
<td>8.2%</td>
</tr>
<tr>
<td>Logical Sensors v2</td>
<td>4,817KB</td>
<td>4,639KB</td>
<td>4,163KB</td>
<td>13.5%</td>
</tr>
<tr>
<td>Logical Sensors v3</td>
<td>493KB</td>
<td>430KB</td>
<td>263KB</td>
<td>46.6%</td>
</tr>
<tr>
<td>Wheels Free</td>
<td>18KB</td>
<td>18KB</td>
<td>18KB</td>
<td>NO</td>
</tr>
<tr>
<td>Wheels Slip</td>
<td>75KB</td>
<td>75KB</td>
<td>75KB</td>
<td>NO</td>
</tr>
<tr>
<td>Wheels Velocity</td>
<td>338KB</td>
<td>338KB</td>
<td>338KB</td>
<td>NO</td>
</tr>
<tr>
<td>XAcceleration</td>
<td>27KB</td>
<td>21KB</td>
<td>21KB</td>
<td>22.2%</td>
</tr>
<tr>
<td>YawRate</td>
<td>384KB</td>
<td>119KB</td>
<td>119KB</td>
<td>69.0%</td>
</tr>
</tbody>
</table>

In the table above, the tested programs “Wheels Free, Wheels Slip, Wheels Velocity, XAcceleration, YawRate” are all sub-modules of the program “Logical Sensors”. The program “Logical sensors” itself is too big for a global optimization. The idea here is to split it into smaller programs and do a local optimization. The three versions “v1, v2, v3” are different configurations that only contain parts of the sub-modules. The right-most column shows the proportions of the reductions from the original-sized *AIFSystem* files to the “IVIF+IDPC” optimized *AIFSystem* files. The run-time of all the tested examples are able to finish within 5 minutes.

The testing of the practical programs shows the potential of the optimization techniques. In particular we can learn from the optimization that:

- **Dead code elimination** and **Passive code elimination** together usually achieve better results.
• It might be good to optimize sub-modules when the project is big – to optimize parts of the system then put them together.

• For some big examples the effect is dramatic.

• For some small examples there is simply no effect.

For large programs, the optimization usually leads to better result. It might be the case that for large parallel loops, the original EFSM may contain lots of fake transitions. These transitions can be eliminated, which causes a chain effect that leads to more reductions of the system.

Another problem revealed here is that EFSM based method is not suitable for large projects. The AIFSystem files generated for the EFSMs usually blows 10 to 100 times larger than the original AIFSystem files. The reason is that EFSM is flatterned while the original codes are modular. The flatterened parallelism might lead to a state explosion. For example in the original program “Logical Sensors” there are about 40 to 50 location variables. With such amount of location variables, the state space would be around $2^{40}$, which is too huge. What is worse is that the DGAs will be copied to many of the states again and again, since once the program is flatterned, the related location variable of a particular DGA might exist in many state encodings. This will greatly enlarge the size of the AIFSystem structure.
5 Summary

5.1 Conclusion

In this thesis, a set of optimization techniques is developed for the optimization of an intermediate format of synchronous programs – the Extended Finite State Machine. The foundation of the optimization is dependency analysis of the control flow and data flow of the EFSM. Then, an algorithm of EFSM generation is introduced. In this algorithm, the predicates in guards are treated as boolean variables first. Then an efficient case-distinction procedure is performed for the predicates to generate all possible combinations of boolean assignments of them.

The optimization techniques bring down the abstraction level to predicate logic by using an SMT solver. In particular, constant propagation and invariant inference are developed. These methods restore the predicates in guards to what they used to be, and collect whatever useful information over the EFSM to check the satisfiability of the guards. Once the satisfiability is known, some unexecutable guarded actions can be safely removed. Based on these techniques, a fixpoint computation is done for the EFSM reduction. This procedure is the core of the dead code elimination presented in this thesis.

Another useful optimization technique introduced in the thesis is the passive code elimination. This method tries to identify local variables in EFSM states that are irrelevant for the outputs. The guarded actions that compute these unrequired variables can be safely removed. A conservative approximation is established for the computation.

Finally, an array optimization technique is introduced – array normalization. It uses an SMT solver to detect those semantically equivalent array accessions within each EFSM state, then rewrites them to a syntactically equivalent normal form. Based on this method, many optimizations generally lead to better results. Passive code elimination is shown as an example for illustration.

Tests have been done for the evaluation of the optimization techniques. A preliminary result has shown the potentiality of the optimization techniques. It is shown that by using SMT based techniques, the optimization methods are able to eliminate the codes that can only be identified by checking the semantics of the program. Although
the practical test only samples a tiny amount of programs, more than half of the tested programs shown positive results. Results of tests also shown that dead code elimination and passive code elimination combined together can usually get the most amount of reduction of the size of EFSMs.

5.2 Future Work

There are more things that can be done based on this thesis.

In particular, an important assumption of this thesis is the causality correctness of the Quartz programs. To check the causality correctness of a program, the SMT-based method can also be utilized. An ideal solution might be that, we can establish an assertion system of the program, and by checking satisfiability of the assertion system, we know that whether the program is causality correct.

Another problem is the size of an EFSM. FSM based techniques usually have the problem of state explosion. In particular, a modular way for the EFSM generation is desired for dealing with larger projects. It might be necessary to change the EFSM model to a more hierarchical model, which is able to lead to smaller size of codes. Meanwhile, a further test of C-code generation as well as a thorough comparison between the optimization methods in this thesis and the optimization provided by GCC should be performed for better evaluation of the optimization.
A Supported Types of Quartz Expressions

Supported boolean types:

- `BoolVar of QName`  
- `BoolConst of bool`  
- `BoolNeg of BoolExpr`  
- `BoolConj of BoolExpr * BoolExpr`  
- `BoolDisj of BoolExpr * BoolExpr`  
- `BoolImpl of BoolExpr * BoolExpr`  
- `BoolEqu of BoolExpr * BoolExpr`  
- `NatEqu of NatExpr * NatExpr`  
- `NatLes of NatExpr * NatExpr`  
- `NatLeq of NatExpr * NatExpr`  
- `IntEqu of IntExpr * IntExpr`  
- `IntLes of IntExpr * IntExpr`  
- `IntLeq of IntExpr * IntExpr`  
- `RealEqu of RealExpr * RealExpr`  
- `RealLes of RealExpr * RealExpr`  
- `RealLeq of RealExpr * RealExpr`

Supported integer types:

- `IntVar of QName * bool list option`  
- `IntConst of bool list`  
- `IntAdd of IntExpr * IntExpr`  
- `IntSub of IntExpr * IntExpr`  
- `IntMul of IntExpr * IntExpr`  
- `IntDiv of IntExpr * IntExpr`  
- `IntMod of IntExpr * IntExpr`

Supported natural number types:

- `NatVar of QName * bool list option`  
- `NatConst of bool list`  
- `NatSat of bool list * NatExpr`  
- `NatAdd of NatExpr * NatExpr`
Supported types of *Quartz* expressions:

- **NatSub** of *NatExpr * NatExpr
- **NatMul** of *NatExpr * NatExpr
- **NatDiv** of *NatExpr * NatExpr
- **NatMod** of *NatExpr * NatExpr

Supported real number types:

- **RealVar** of *QName*
- **RealConst** of float
- **RealE** //2.71828
- **RealPI** // 3.14159
- **RealAdd** of *RealExpr * RealExpr
- **RealSub** of *RealExpr * RealExpr
- **RealMul** of *RealExpr * RealExpr
- **RealDiv** of *RealExpr * RealExpr

Supported bit vector types:

- **BtvVar** of *QName * int option
- **BtvConst** of bool list
- **BtvOfNat** of *NatExpr*
- **BtvOfInt** of *IntExpr*
- **BtvNeg** of *BtvExpr*
- **BtvConj** of *BtvExpr * BtvExpr*
- **BtvDisj** of *BtvExpr * BtvExpr*
- **BtvImpl** of *BtvExpr * BtvExpr*
- **BtvEqu** of *BtvExpr * BtvExpr*
- **BtvAppend** of *BtvExpr * BtvExpr*
- **BtvReverse** of *BtvExpr*
- **BtvReplicate** of *BoolExpr * int*

Supported array types:

- **ArrayVar** of *QName * (int list * QType)
Bibliography


[38] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.


