Probabilistic Model Checking of Synchronous Programs

Diploma Thesis

University of Kaiserslautern

Department of Computer Science

Embedded Systems Group

Manuel Gesell

June 2008

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Kaiserslautern, den 30.06.2008

Manuel Gesell

Danksagung

An dieser Stelle möchte ich allen danken, die es mir ermöglicht haben diese Arbeit zu verfassen. Dazu zählen vor allem meine Frau Johanna, meine Freunde und meine Kollegen in der Arbeitsgruppe Eingebettete Systeme. Des Weiteren danke ich besonders meinen Korrekturlesern Owen Schenkel, Andreas Morgenstern und Jens Brandt für die geleistete Arbeit. Meinem Betreuer Prof. Dr. Schneider danke ich vorallem für das angenehme Arbeitsklima und die gute Betreuung während der Arbeit.

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays, the application area of embedded systems is widespread. They control household appliances like microwave ovens, radios, the anti-lock braking system in cars, the stability of aircraft and much more. Reliability, fault tolerance and safety properties are very important for some embedded systems. These properties can be checked by verification. Verification in safety critical systems is inalienable. It is important to identify and eliminate errors as early as possible. Synchronous languages have some important abilities to archieve these exigencies. Unfortunately the semantic is harder to read. A main point for this is that statements may executed in parallel that are placed on totally different lines in the code. Nevertheless the advantages of synchronous languages for embedded systems overbalance the disadvantages.

1.2 State of the Art

There exists many synchronous languages like Esterel[18], Lustre[21], VHDL[25], statemate[45] and much more[47]. Each language has advantages and disadvantages, which differs from each other because they are developed for special purposes. In this work the synchronous language Quartz[41] will be used to exemplify the approach of this work, because this language is developed and used by the Embedded Systems group of the University of Kaiserslautern. The advantages of synchronous languages for embedded systems and the requirement of comfortable verification techniques for critical systems is realised by model checking [14]. Model checking is a successful technology to verify requirements and designs for a variety of real-time embedded and safety-critical systems. The essential idea behind model checking is that a model checking tool takes a system requirements or design, a so called model, and a property, a so called specification that the final system is expected to satisfy. The tool then generates a counterexample if possible or confirm that the system satisfy the specification. A counterexample manifests in which way the specification is not fulfilled. With the help from the counterexample it is possible to find the error and correct the model. Model checking itself has many forms of expression. Some early approaches based on automatic abstraction and modular

verification like [32] or Boolean automata [33, 2]. Amla et al. [1] use timing diagrams for model checking. A important innovation for the model checking is the technique of symbolic model checking[13], which is used in combination of binary decision diagrams (BDDs) [49]. Symbolic model checking is based on Boolean formulas that represent the model instead of previously representing the explicit model. BDDs are used to represent these Boolean formulas. This innovation first acceptable to verify industrial projects. Another popular approach of model checking is bounded model checking[9]. This approach depends on SAT-solvers. Using this technique both the model and specification are translated to a propositional logic formula by rolling them out. The model checking problem is converted into a satisfiability problem of propositional logic. A further innovation was model checking with Presburger arithmetic [12, 42], which handles infinity-state-systems. Examples for model checking tools are Beryl [3],DCVALID [16], SMV[43], KRONOUS[26], BLAST[10] and SPIN[44, 24]. All have in common that they do not support verification in a probabilistic approach because the underlying system is based on the temporal logic Linear Time Temporal Logic(LTL) or Computation Tree Logic(CTL). In this work another approach to extend model checking is regarded, the so called probabilistic model checking. For this approach LTL or CTL are not sufficient. Nevertheless, it is possible to extend CTL. An extension of the temporal logic CTL needed for probabilistic model checking is the Probabilistic Computation Tree Logic(PCTL), which is described in [22] and [8]. Continuous Stochastic Logic(CSL) is another extension of CTL, introduced in [4, 5] for this purpose. An overview and the problems of using probabilistic model checking for synchronous languages are described in [15, 39]. Probabilistic model checking is used to exhibit the stochastic behaviour of a system. It is a formal verification technique for the analysis of systems. Probabilistic model checking focuses on proving the correctness of stochastic systems. The behaviour of stochastic systems depends on probabilities. It is possible to couch that a system satisfies a specification in a certain percentage rate. Probabilistic model checking tools are PRISM [28], MRMC[35], E⊢MC^2[19], YMER[50] and VESTA[48]. A comparison of these tools are made in [36]. In this comparison three tools accented. YMER was the fastest tool with good memory usage but it accepts a limited range of supported probabilistic operators, which is a great disadvantage. MRMC was faster and used often used less memory than PRISM for the studied experiments but MRMC does not support such big systems as PRISM. PRISM is for small system slower than YMER and MRMC but the usability and the additional features like plotting results and the competitiveness with these two tools was a main reason for my decision to choose PRISM as the verification tool. Another advantage of PRISM is that it is platform independent hence an inclusion into the Averest framework is more easily realisable. A further argument is the wide range of publications for PRISM and the first experience in practical use [29]. The analysis of a system with probabilistic model checking allows a better statement to be made in the case of fault tolerance and other safety properties. A non-probabilistic model checker for example only attests if the specification is fulfilled or not. A probabilistic model checker additionally makes a statement about how far the satisfaction of the property is. This means that it is better that a specification is fulfilled in 99 percent than for five percent. For some systems there is no discrepancy between 99 percent and 100 percent.

1.3 Contribution

The contribution of this work is to make probabilistic model checking for synchronous programs possible. To achieve this goal an already existing intermediate format will be introduced. It is possible to translate all synchronous programs into this format. The language Quartz is a case in point. Then a compiler from the intermediate format to the PRISM language will be developed. This compiler will be called Opal. The PRISM language itself allows, with help of the tool PRISM, a probabilistic model checking.

1.4 Outline

Chapter 2 will be concerned with synchronous languages. It will describe the syntax and the semantics of the synchronous language Quartz in section 2.1. The basics for the translation of Quartz programs into guarded actions also will be given. Tools that allow Quartz to be used for different applications will be presented in section 2.2. Additionally, the main part of chapter 2 will be the introduction of the Averest interchange format(AIF) in section 2.3. The basic principals of the probability theory will be explained in chapter 3. This basic knowledge will be useful in the following chapters. A presentation of the probabilistic model checking tool PRISM will be made in chapter 4. The presentation includes some example code and the usage of the tool. With the help of the tool PRISM, probabilistic model checking for synchronous programs given in the Averest interchange format will be achived. In chapter 5 the compiler from the AIF to a PRISM language file will be described. Therefore, the first innovation, the Opal Probability File (OPF), will be described in section 5.1. A file in this format will be the parameter of the later introduced compiler Opal. The principals of the new compiler Opal will be given in section 5.3. Additionally an example will be explained at the end of this chapter. Section 5.4 will contain some code snippets of important functions, which are written in Moscow ML[34]. A conclusion will be given in chapter 6.

Chapter 2

Synchronous Languages

Synchronous languages can be used to describe reactive systems, because it is possible to generate software and hardware from the same synchronous program. The decision which part of the program will be synthesized in hardware can be made in the final stages of the development, there is no need to make it earlier. Later on you can easily change the hardware-software partitioning of a system. It is also possible to simulate application-specific hardware. For verification the formal semantics of these programs make it easy to prove the correctness.

All synchronous programs follow the principle of perfect synchrony [6], which can be explained with micro and macro steps as they occur in Esterel or Quartz [41]. Most instructions are executed as micro steps, which occur instantly.

The only way to consume time is by using a macro step. Macro steps include a finitely many micro steps and have the execution time of one step. Every thread will be synchronized after each macro step.

The worst case execution time can be determined easily [30] because every macro step includes only a finite number of micro steps. This is an important feature for real-time and safety-critical systems.

In this work, the synchronous language Quartz will be refered to, it being a modification of Esterel. Quartz will be used to exemplify the approach of this work. Quartz has been developed by the Embedded Systems group at the University of Kaiserslautern. The Averest framework [3] contains tools for compilation, synthesizing, simulation and verification of Quartz programs.

This chapter introduces the basics of the language Quartz. First variables will be described. Then the syntax of statements and modules are explained. At the end the semantic is announced through the control and data flow and guarded actions. A precise definition is given by so called SOS rules, which are defined in [41]. Afterwards, the causality analysis and some basic terms are defined. At the end of the chapter, the Averest framework will be presented. The Averest framework includes tools, which allow to use Quartz for different applications. The main part of the chapter will be the description of the Averest Interchange Format (AIF).

2.1 The Synchronous Language Quartz

This section introduces the basics of the synchronous language Quartz. The section starts with the syntax of variables, statements and modules. Afterwards, the semantics are explained in layman's terms.

2.1.1 Variables

Variables in Quartz can have two modes. The first is the mode memorised, which means that a value is held until the next change. The second is the mode event, which means that an assignment is only valid during the current macro step and if there are no assignments in the current macro step the variable equals the default value of the type. Henceforth mode memorized variables will be known as state variables and mode event variables will be called event variables. State variables work similar as in other programming languages. In the declaration a variable can be defined as an event variable or not. An assignment is a micro step, that way different assignments are able to be done in one single macro step.

Example

State variable:	event variable:
int x;	**event int** x;
bool $a,\ b$;	**event bool** $a,\ b$;
$x = 7$;	$x = 7$;
next$(a) =$ **true**;	**next**$(a) =$ **true**;

The keyword **next** means in this case that the assignment takes place in the following macro step. The example on the left-hand side defines the state variables a, b and x. Their values are only changed by assignments. Hence, **x**=7 holds in the current macro step. The value of **a** equals in the current macro step to $false$, which is the default value of type *bool*. After the current macro step **a** takes the assigned value $true$. The value of variable **b** equals to the default value $false$ until the first assignment. The example on the right-hand side defines three event variables, which values are changed in each macro step to the new assigned value or if an assignment is absence to the default value of the type. Hence, in the current macro step **x**=7 holds and **a=false** holds because there is no assignment in the current macro step. In the next macro step **x** equals the default value of the type **int**, which is zero, and **a=true** holds through the **next** assignment of the previous macro step.

2.1.2 Statements

It was mentioned that the end of a macro step is defined through a **pause** statement. Next, some other basic statements of the language Quartz will be described. A lot of these statements exist but they are only macros of the basic statements, therefore they will be omitted.

```
pause;
```

The control rests until next step in this statement. **pause** is one of the statements that define a macro step.

```
await b;
```

This statement consumes at least one step and defines like **pause** the end of a macro step. After the first step, the predicate **b** is checked. Unless **b** is true the control rests in the statement otherwise the control leaves this statement and enters the next statement. A modification is the statement **await immediate b** where the predicate **b** is checked also in the first step.

```
S1; S2
```

If **S1** is instantaneous the execution of **S2** starts in the same macro step. Otherwise **S2** starts in the step **S1** terminates.

```
S1 || S2
```

The execution of both statements **S1** and **S2** starts immediately and is synchronised at each macro step. The control rests in the statement until both statements terminat.

```
if (b) {S1} else {S2}
```

If **b** is true, the execution of **S1** starts, otherwise the execution of **S2** starts. After the termination of the executed statement, the control leaves the statement.

```
loop {S};
```

The statement **S**, called loop body, reruns after termination. The statement **S** has to consume time, otherwise the macro step contains infinitely many micro steps, which is a contradiction.

```
abort S when (b);
```

The execution of **S** starts immediately without checking **b**. In the case **S** terminates the whole statement terminates. Otherwise **b** is checked in the next macro step. If **b** is true, the statement terminates immediately, otherwise the execution of **S** is continued until **b** is true or **S** terminates in one of the following steps. There are two modifiers, first **weak abort** in the step where **b** is true, the action in the current macro step of **S** is executed and the statement terminates afterwards. The other modifier is **when immediate**, where **b** is also checked in the first step. Both modifierers can occur at the same time.

2.1.3 Module

A Quartz program consists of a list of modules. Modules have names and declared inputs and outputs, where outputs have an Ampersand between type and name. The body includes statements and calls of other modules. All statements till a **pause** or equivalent statement are executed simultaneously. A change of a variable takes immediately place.

Example

```
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);
}
```

The **module** is known under the name of ABRO. The three event variables **a**, **b** and **r** of type **bool** are inputs. The output of type **bool** is the event variable **o**. The program awaits after the first step the occurrence of the events **a** and **b**. The order of the occurrence of **a** and **b** are not important. Only when each event occurs at least once is the output **o** emitted in the same step and the program awaits in the next step, a reset like event **r**. If **r** occurs before **a** and **b** have occurred at least once, the abort statement terminates and the loop reruns the body so that the computation starts from the beginning. The computation also restarts if the control rests in the **await**(**r**) statement and **r** occurs.

2.1.4 Semantics

So far the language Quartz has been introduced. The formal semantics of the language Quartz can be described by logical transition rules, which have to be defined for each statement. Additionally, the application of the rules require an environment $\mathcal{E}$. The transition rules state in which way a statement is executed. With this rules it is possible to check the equivalence of two statements. And the consistence of the performed actions and the environment have to be checked. In [41] a straightforward description has been given. Here it will be omitted.

2.1.5 Control and Data Flow

The control flow describes how control moves in a statement. Here the distinction between entering, moving inside and leaving a statement is made. The only statements where the control can rest are **pause**, **suspend immediate** and **await**.

- Entering statement S means that the control is outside S and enters S in this macro step and does not leave it in this step. In the next step, the control is inside S. Only statements consuming time could be entered. Statements consuming no time are called instantaneous statements.
- Moving inside statement S means that the control is inside S and does not leave it this step. In the next step the control is inside S.
- Leaving statement S means that the control is inside S and leaves it this step. In the next step, the control is outside S.

The data flow describes preconditions for actions and the effect in case of execution the action.

Control Flow Predicates

The following predicates are defined as:

- $Enter(S)$ is true if the control enters statement S in this macro step
- $Inst(S)$ is true if statement S is consuming no time
- $Move(S)$ is true if in this macro step the control is moving inside statement S
- $Term(S)$ is true if control leaves statement S in this macro step

All of these formulas can be recursively defined like it is done in [41].

Properties of Control Flow Predicates

The following implications are valid

- $Inst(S) \rightarrow \neg Enter(S)$
- $Enter(S) \rightarrow X(Inside(S))$
- $Term(S) \rightarrow Inside(S)$
- $Move(S) \rightarrow Inside(S)$
- $Move(S) \rightarrow X(Inside(S))$
- $Move(S) \rightarrow \neg\, Term(S)$

Guarded Actions

A guarded actions for a variable x is a tuple $(\gamma,\ \epsilon)$ where γ is a precondition and ϵ is an expression. The guarded action can only be executed if the precondition γ is fulfilled in this macro step. In each macro step, all guarded actions that the guard is fulfilled is executed. The changes of the executed guarded actions immediately take place. It is possible to translate every synchronous program into guarded actions [41, 7, 31]. The control flow can be easily translated with help of the control flow predicates. The data flow can be translated through the definition of sets for surface and depth for each statement. Afterwards, using a fix-point computation with help of ternary simulation or dual rail encoding the outputs of each program can be computed. This method is called causality analysis.

2.1.6 Causality Analysis

Through the parallel execution of guarded actions and the paradigm of perfect synchrony, there maybe occurs some conflicts like causality cycles. To explain causality cycles some terms are introduced in this section. Afterwards the problem of causality cycles and the technique to resolve these problem are introduced informally. This sections introduces the causality analysis only informal, formal definitions are given in [41, 7, 31].

Determinism, Reactivity and Logical Correctness

The set of all possible synchronous programs, hence every Quartz program, can be classified into deterministic and reactive programs. The conjunction of both classes are the so called logical correct programs.

Determinism

A program is deterministic if each input has at most one possible output. This means that the execution of a program with the same input has always the same output. For a deterministic program it is not possible to get two different outputs for the same input. A program not satisfying this ability is called non-deterministic.

Example

A example for a deterministic program:

```
module P1 (bool i, bool &o){
    if(i) then nothing else emit o;
    ||
    if(o) then nothing else emit o;
}
```

Program P_1 is deterministic because each input `i` has at most one possible output. The output for input **i=false** is **o=true**. The else part of the first if-then-else statement is executed, which emits **o**, because the input variable `i` is false. Parallel, the then part of the second if-then-else statement is executed. Hence, **o=true** holds. The output for input **i=true** is undefined because there exists no possible execution of the program. A contradiction occurs after the execution of the then part from the first if-then-else statement, which makes nothing. Now each assignment of the variable **o** results in a contradiction. In case **o=true**, the then part of the second if-then-else statement is executed and no **emit o** occurs, which is a contradiction to **o=true**. In the other case **o=false**, the else part of the second if-then-else statement is executed and a **emit o** occurs, which is a contradiction to **o=false**.

Reactivity

A program is reactive if every input has at least one possible output. Reactive programs are similar to total functions in the mathematics, there are no inputs that the outputs are undefined. However, one input may have more than one possible output, which makes a reactive program non-deterministic.

Example

A example for a reactive program:

```
module P2 (bool i, bool &o){
    if(o) then emit o else nothing;
}
```

This program is reactive because there are two possible outputs for each input **i**. The two possible outputs are **o** holds and **o** does not hold.

Logical Correctness

Each input of a logical correct program has exactly one possible output. It is the conjunction of deterministic and reactive programs. Only logical correct programs are translated into AIF. The compiler Ruby, which is described in section 2.2.1, returns a failure message if the compilation of a logical in-correct program is tried.

Example

The above mentioned examples, P_1 and P_2, are either reactive or deterministic but not both. Hence, they are not logical correct. The next example is a logical correct program:

```
module P3 (bool i, bool &o){
    if(i) then nothing else emit o;
}
```

This program is logical correct because it is deterministic and reactive. This program has exactly one output for every input. The possible inputs are **i=true**, which has the output **o=false**, and **i=false**, which has the output **o=trueT**. The program P_3 computes the $\neg i$ function.

Causality Cycle

Every acyclic program have a unique behavior, which can be determined with a simple test. A causality cycle is given, if the guard ϕ of a guarded action (ϕ, α) depends on variables modified by α or there exists several guarded actions that depends on each other.

Example

In the next examples a,b and c are Boolean.

```
(a, emit b)
(b, emit c)
(c, emit a)
```

This causality cycle is not deterministic. All Boolean may be either *true* or $false$. However, there are logical correct cyclic programs like:

```
(b, emit a)
(a&¬b, emit b)
```

This program, which has the outputs **a=false** and **b=false**, is logical correct.

Constructive and Self-Satisfying Programs

Logical correct cyclic programs are classified into constructive and self-satisfying programs. Constructive programs allows to constructively compute their actions without case distinctions. Self-satisfying programs need a case distinction to compute their actions but only one case make sense.

The Causality Analysis

The causality analysis is a heuristic to compute the outputs of programs in many cases. Therefore, a good heuristic for checking logical correctness is required. However, it is a heuristic and so some logical correct programs are not accepted. A program is constructive if the causality analysis accepts the program. The heuristic is based on a tree-valued logic and uses fix-point iteration. The here mentioned causality analysis can be improved, approaches therefore are given in [11].

Schizophrenia and Reincarnations Problems

There exists much more problems for synchronous languages like schizophrenia problems, which occurs if a statement is started more than once in a macro step, or the reincarnation problem, which is related to schizophrenia and takes place if a local declaration is left and re-entered within the same macro step. All these problems and their solutions are omitted here but are described in [41].

2.2 Averest Framework

A programming language only makes sense if tool exists, which can be deployed for wide spread use. In this chapter the tools contained in the Averest framework will be described. Averest based on the language Quartz and is developed by the Embedded Systems group of the University of Kaiserslautern. The project homepage with the complete documentation and a download part is [3]. The tools involved are the compiler and simulator Ruby, the model checker Beryl and the code generator Topaz. The communication between these tools is realized through an Averest Interchange Format(AIF), which is created from a Quartz file by the tool Ruby and read from all other tools. The Averest Interchange Format will be the main point of this chapter. Since this format is the mentioned intermediate format, which the compiler Opal is based on. All these tools have their specific ranges of application.

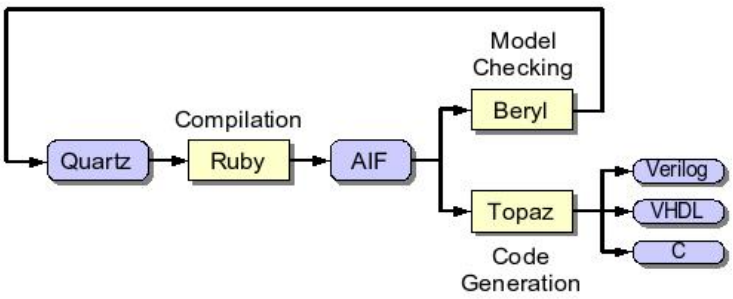

Figure 2.1: Averest Framework [3]

2.2.1 Tools

The compiler and simulator Ruby

A Quartz program can be translated by Ruby to a symbolic description of a transition systems into an AIF file. Furthermore Ruby can simulate a program specified in a Quartz file. For all applications Ruby is needed because it is the only way to generate a AIF file, which is read from all other programs. Ruby uses all techniques and more described in this chapter.

The Model Checker Beryl

The tool Beryl is a symbolic model checker for the verification of finite and infinite state systems. It is able to check μ-calculus formulas in different ways. In addition it is able to calculate at an architecture independent level the worst case execution time of Quartz programs. Probabilistic model checking is not supported by Beryl.

The Code Generator Topaz

For code generation the tool Topaz exists. Quartz programs translated into an AIF file could synthesize to hardware description languages like VHDL or into conventional programming languages like C. For procedural languages, like C, a serialization of the synchronous code is needed and possible.

2.3 The Averest Interchange Format

The Averest Interchange Format(AIF) is a XML file. The compiler Ruby translates each Quartz module into a AIF file, which contains the whole module in form of guarded actions, which will be defined later on, for control and data flow. The AIF file is like a system of equations. Additionally, the file contains the declarations of each variable, the special control flow signals and some definitions are shortcuts for often used terms in the equation system. The last section mentioned is not really necessary but it may make the file shorter and more readable. An exact definition is given in [40].

2.3.1 Declarations

Each variable with exception of the global control signals used in this module must be declared first in this section. There are XML tags to identify the flow of the variable: $< input >$, $< local >$ and $< output >$. The attribute *storage* is essential for the last two tags. This attribute describes whether the variable is an event variable or a state variable. The child nodes of these XML tags defines the type of the variable. In addition, each state of the system has a unique Boolean event variable defined as $< label >$. Each variable or label definition includes the attribute *name*, which has the value of the unique name of it. An example of a declarations section with two variables and one label are:

```
<declarations>
    <input name="variable1"><bool/></input>
    <local name="variable2" storage="memorized">
        <nat size="6u"/>
    </local>
    <label name="ell1"/>
</declarations>
```

The first variable **variable1** is an input variable of type Boolean. **variable2** is a state variable because the attribute **storage** equals **memorized** and the type is a natural number between 0 and 6. The system itself contains only 2 states where *ell*1 holds or not.

2.3.2 Control Signals

As described in section 2.1.5 there exist several control signals. In addition to the declarations, there are a fixed set of description signals each module has. These signals control the module execution. This set includes the signals:

- *_goAlpha* is the condition that the module is activated in this macro step.
- *_goEta* is the condition that the module is entered in this macro step.
- *_abort* is the condition that the module is aborted in this macro step.
- *_suspend* is the condition that the module is suspended in this macro step.
- *_inst* is the condition that the execution of the module is instantaneously.
- *_inside* is the condition that the control flow is inside the module in this macro step.
- *_term* is the condition that the execution of the module is terminated this macro step.

This signals can be used as variables everywhere in the file. In the AIF file the control signals presented as following:

Example

```
<controlSignals>
    <goAlpha name="_goAlpha"/>
    <goEta name="_goEta"/>
    <abort name="_abort"/>
    <suspend name="_suspend"/>
    <inst name="_inst"/>
    <inside name="_inside"/>
    <term name="_term"/>
</controlSignals>
```

2.3.3 Definitions

Definitions are shortcuts for often used terms. The syntax is quite similar to the definitions of variables. The usage is the same as the usage of variables.

Example

```
<definitions>
    <def name="_var1">
        <bvOr>
            <bvVal><var name="a"/></bvVal>
            <bvVal><var name="b"/></bvVal>
         </bvOr>
    </def>
</definitions>
```

2.3.4 Data Flow

All changes of data are described in the section <**dataFlow**> through guarded actions. The guarded actions are arranged into immediate and delayed guarded actions. Immediate guarded actions take place instantaneously, thus in the current macro step. Delayed guarded actions take place in the next macro step. These actions are like a carry for the next macro step. All changes of variables are described in this section.

2.3.5 Control Flow

All changes of the inner state of the system is described in the <**controlFlow**> section through guarded actions. There is a splitting into delayed and immediate guarded actions like described above. Only labels are changed in this section.

2.3.6 Guarded Actions

Let a tuple $(guard, execution\_part)$ be a guarded action for a variable x, where $guard$ is a Boolean and $execution\_part$ is an expression. In the AIF file, the variable name defines which variable is changed through the action. A guard is

a Boolean term over all variables, labels and control signals. The execution part describes the change of the variable and has the same type. Guarded actions in the AIF file are described in XML and have following the syntax:

```
<var name="name"/>
    <immediate>
        <guard>
            guard
        </guard>
        execution_part
    </immediate>
    <delayed>
        <guard>
            guard
        </guard>
        execution_part
    </delayed>
```

The <**immediate**> part describes changes of the variable in the current macro step. All changes described in the <**delayed**> part describes changes in the next macro step, which can be defined through a **next** assignment. A precise definition for the syntax of the guarded actions is given in [40].

Tasks

The AIF supports some other features not used in this work, so they will be omitted here. For example the <**task**> part of the AIF, which defines specifications, the program specified in the AIF has to fulfil. As a further work the PRISM properties, defined in the PRISM property file, may be included in this part. A problem therefore is that this part is taken out of the Quartz file during compilation.

Example AIF File - bvTest.aif

```
<?xml version="1.1"?>

<aif>
<module name="bvTest">
<declarations>
    <input name="a"><bv size="2u"/></input>
    <input name="b"><bv size="2u"/></input>
    <input name="mode"><bool/></input>
    <output name="c" storage="memorized"><bv size="2u"/></output>
    <label name="ell1"/>
</declarations>
<controlSignals>
    <goAlpha name="_goAlpha"/>
    <goEta name="_goEta"/>
    <abort name="_abort"/>
    <suspend name="_suspend"/>
```

```
    <inst name="_inst"/>
    <inside name="_inside"/>
    <term name="_term"/>
</controlSignals>
<definitions>
    <def name="_var1">
        <bvOr>
            <bvVal><var name="a"/></bvVal>
            <bvVal><var name="b"/></bvVal>
         </bvOr>
    </def>
</definitions>
<dataFlow>
    <depth>
    <actions><var name="c"/>
        <immediate>
            <guard>
                <blVal><var name="mode"/></blVal>
            </guard>
            <bvVal><var name="_var1"/></bvVal>
        </immediate>
    </actions>
    <actions><var name="c"/>
        <immediate>
            <guard>
                <blNot>
                <blVal><var name="mode"/></blVal>
                </blNot>
            </guard>
            <bvVal><var name="a"/></bvVal>
        </immediate>
    </actions>
    </depth>
</dataFlow>
<controlFlow>
    <surface>
        <actions><var name="ell1"/>
        <delayed>
            <guard>
                <blVal><var name="_goEta"/></blVal>
            </guard>
            <blTrue/>
        </delayed>
    </actions>
    </surface>
    <depth>
    <actions><var name="ell1"/>
        <delayed>
            <guard>
                <blVal><var name="ell1"/></blVal>
```

```
            </guard>
            <blTrue/>
        </delayed>
    </actions>
    </depth>
</controlFlow>
<tasks>
</tasks>
</module>
</aif>
```

Outline

This chapter introduced the synchronous language Quartz. A programming languages is only as good as its peripheral tools, as a result of this the last chapter presented a framework including a great composition of such tools. So far every thing presented is known and developed by the Embedded Systems group of the University of Kaiserslautern. The next chapter introduces the basic of the probabilistic theory used in the following chapters.

Chapter 3

Probability Theory

The behavior of a stochastic process is unpredictable. An often repeated stochastic process may have different results. But the results exhibit certain statistical patterns, which can be studied and predicted. The law of large numbers and the central limit theorem are two representative mathematical results describing such patterns. With probabilistic model checking you are automatically involved with such problems. A simple introduction will be given in this chapter based on [20, 17]. Some basic definition will be given at the beginning. Then, Markov chains[37] used in the next chapters are defined.

3.1 Definitions

Definition 1 (Sample Set) *All possible results, the so called sample set, of a experiment will be defined as* Ω.

Example

For example all possible results of a coin flip are *head* or *tail*, this implies $\Omega = \{head, tail\}$.

Definition 2 (Event Space) *The event space* Σ *is a* σ*-algebra of subsets of* Ω. *An element* $A \in \Sigma$ *is called an event.*

Definition 3 (Probability Measure) *The probability measure* P *is a function from* Σ *to the real numbers. Let* $A \subseteq \Omega$ *then* $P(A)$ *is the probability that* A *is the result of the experiment. For example* $P(head)$ *of a coin flip experiment equals to* $\frac{1}{2}$.

Definition 4 (Probability Axioms) *There are three probability axioms:*

1. $P(A) \geq 0$ *for all* $A \in \Sigma$.
2. $P(\Omega) = 1$, *the probability of all possible results equals one.*
3. $P(E_1 \cup E_2 \cup \dots) = \sum_{i \in \mathbb{N}} P(E_i)$, *for any countable sequence of pairwise disjoint events* E_i.

Definition 5 (Measure Space) *A function ϕ defined on a σ-algebra Σ over a set X, which takes values between zero and one, satisfies the following rules*

- $\phi(\{\}) = 0$.
- $\phi(E_1 \cup E_2 \cup \dots) = \sum_{i \in \mathbb{N}} \phi(E_i)$, *for any countable sequence of pairwise disjoint events E_i.*

is called a measure.

A measure space is the triple (X, Σ, ϕ).

Definition 6 (Probability Space) *Let the sample set Ω, the σ-algebra Σ of subsets of Ω and the probability measure P be given. The probability space (Ω, Σ, P) is a measure space with a measure P that satisfies the probability axioms.*

Elementary

- $P(\{\}) = 0$, the probability of an impossible result equals zero.
- $\sum_{\forall \omega \in \Omega} P(\omega) = 1$, the sum of all possible results equals to 1
- $0 \leq P(A) \leq 1$, all probabilities are in the interval 0 up to 1
- $P(\bar{A}) = 1 - P(A)$, the sum of probabilities that occur and those that not occur of an result equals 1

Definition 7 (Independence) *Let A and B be events, then A and B are independent if*

$$P(A \cap B) = P(A)P(B)$$

holds.

Definition 8 (Mutually Exclusive) *Let A and B be events, then A and B are mutually exclusive if*

$$P(A \cap B) = 0$$

holds.

Definition 9 (Conditional Probability) *Let $A, B \subset \Omega$ with $P(B) > 0$ then*

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

is called the conditional probability. For $P(B) = 0$ the conditional probability $P(A|B)$ is undefined.

Definition 10 (Lemma) *Let A and B be independent events, then $P(A|B) = P(A)$ and $P(B|A) = P(B)$.*

proof:

$$\begin{aligned} P(A|B) &= \frac{P(A \cap B)}{P(B)} \\ &= \frac{P(A)P(B)}{P(B)} \\ &= P(A) \end{aligned}$$

the other proof is analogous.

Definition 11 (Lemma) *Let A and B be mutually exclusive events, then the conditional probability $P(A|B)$ equals to zero.*

proof:

$$\begin{aligned} P(A|B) &= \frac{P(A \cap B)}{P(B)} \\ &= \frac{0}{P(B)} \\ &= 0 \end{aligned}$$

Definition 12 (Bayes Theorem) *The relation between $P(A|B)$ and $P(B|A)$ is defined as:*

$$P(B|A) = P(A|B)\frac{P(B)}{P(A)}$$

Definition 13 (Random Variable) *A random variable is a function mapping the possible results to the real numbers. The formal definition for a random variable X and a probability space (Ω, Σ, P) is:*

$$X : \Omega \rightarrow [0, 1]$$

3.2 Markov Chain

Definition 14 (Markov chain) *A sequence of random variables X_1, X_2, X_3, ... over a probability space (Ω, Σ, P) that satisfies for all $n \in \mathbb{N}$*

$$P(X_{n+1} = x \mid X_n = x_n, \ldots X_0 = x_0) = P(X_{n+1} = x \mid X_n = x_n), \quad x, x_i \in X$$

is called a (discrete-time) Markov chain. In a Markov chain each state is independent from the future and past states. A Markov chain with continuous index $n \in \mathbb{R}$ is called continuous-time Markov chain.

Outline

The next chapter presents a tool developed by the Computing Laboratory at the University of Oxford. This tool allows probabilistic model checking. Probabilistic model checking for synchronous programs is permit with assistance of this tool.

Chapter 4

PRISM

PRISM[28] is a probabilistic model checker. PRISM is both a command-line and a graphical user interface tool. It is able to handle formal modeling and analysis of systems, which exhibit random or probability behavior. PRISM is used to construct a probabilistic model, and then for evaluating the result of properties. For the purpose of the construction and analysis of a model by PRISM, it must be specified in the PRISM language. The simulation of a model is also supported by PRISM. There is also the possibility to define experiments over certain constants and afterwards plot these results into a graph, which is shown at the end of the chapter, or plot a score table. A profound insight is given in [23]. A range of model analysis techniques is contained in PRISM. These include graph-based algorithms for reach ability called qualitative methods, and quantitative methods for numerical computation of probabilities and expected cost or reward values. The underlying data structure that PRISM is based on are multi-terminal binary decision diagrams (MTBDDs)[38]. In PRISM, three different types of probability models are supported. These will be introduced in the next section. Afterwards, the basics of the PRISM language, which is used to describe a model will be discussed. At the end of the chapter, an example will make the main features of the PRISM language clearer. With a model some properties can be specified, which would be verified, this is the issue of section 4.3 Property specification. An example of a property specification file for PRISM will also be described.

4.1 Probabilistic Models

In PRISM, three types of probability models are supported: Markov decision processes (MDPs), continuous-time Markov chains (CTMCs) and discrete-time Markov chains (DTMCs). Each model has a specific application area. In the PRISM language the used probability model is specified by one of the keywords **dtmc**, **ctmc** or **mdp**. One of these can occur anywhere in the file except inside modules or declarations where the default model is MDP. The user must choose one of the models because the results of an analysis greatly depends on the chosen model.

Markov Decision Process - MDP

A Markov decision process is a non-deterministic model. It is used in situations where outcomes are partly under the control of a decision maker and partly random. In the fact, if there are two or more commands enabled, the choice which command is taken is non-deterministic.

Continuous-Time Markov Chain - CTMC

A process satisfies the Markov property if the probability distribution of the current state is conditionally independent of the path of past states. This means that the process takes no account of values in the past. Only the current state is responsible for the probability distribution. A continuous-time Markov chain is a stochastic process that satisfies the Markov property and means that transitions could occur at every point in time. In fact if there are two or more commands enabled, the choice which command is taken is modeled as a race between transitions.

Discrete-Time Markov Chain - DTMC

A discrete-time Markov chain is a discrete-time stochastic process with the Markov property. Only at discrete points of time does the system follow a transition between states. This may change its state from what it is currently to another state, or it may remain the same. The chosen transition depends on the probability distribution of the current state. In fact, if there are two or more commands enabled, each command is selected with equal probability. This behavior is needed to analyze synchronous programs, because a synchronous program only changes it state at the beginning of a macro step as a result of the perfect synchrony property.

4.2 PRISM Language

This section describes the basics of the PRISM language, which is needed to analyze a model in PRISM. Fundamental components are variables and modules. A model will contain a number of modules, global variables and some declarations. A module itself contains local variables and commands. The commands describe the behavior of a module. The state of the model is determined by the local states of all modules and the global variables.

Commands

A command takes the form:

$$[\ syncLabel\ ]\ guard\ \text{->}\ prob_1\ :\ update_1\ +\ \ldots\ +\ prob_n\ :\ update_n\,;$$

The *syncLabel* could be used to synchronize different modules, which will be described in the Section 4.2.1. The guard *guard* depends on all global and local variables of the model not only on the local variables of the module. If the guard is equal to true in the current state, the transition is enabled. Which transition is selected depends firstly on the chosen model and secondly on the probabilities in the command. The $prob_i$'s are *double* constants. If a command

consists only of one $prob_1$ block, which equals to 1.0, this block can be omitted and only the $update_1$ block is necessary in this case. It is necessary that in DTMCs and MDPs the sum of the $prob_i$ equals one because the $prob_i$'s stand for probabilities. In CTMCs the $prob_i$'s stand for (positive-valued) rates, rather than probabilities. The $update_i$'s contains assignments to local variables and in the case that the *label* is empty, assignments to global variables of the model are also allowed, but only in this case. The assignment in a $update_i$ block are connected by an ampersand. An assignment takes the form:

```
variable_name'=new_value
```

The apostrophe indicates the variable, which is changed in this assignment.

Example

```
[] die=1 -> 0.5:(z'=2)&(count'=count+2) + 0.5:(z'=3)&(count'=count+3);
[] die=6 -> (count'=0);
```

The first command has the guard **die**=1 and if enabled there are two blocks, which will be executed with the same probability. The command in the second line has the guard **die**=6 and if enabled count will be reset to zero with probability 1.0.

Module

A module is specified as:

```
module name definitions commands endmodule
```

Each module has a unique *name*. Declarations of variables are made in the section *definitions* in the form:

```
identifiers : range init value;
```

The *identifiers* is the unique name of the variable. *Range* is a finite interval of integers with upper and lower bound of the form [*lb*..*ub*] or the keyword *bool* for the two Boolean values, other data types are not supported. The **init** keyword with a *value* is optional and defines the initial value of the variable and reduces the number of initial states of the model.

Model

A model contains modules and global variables, which differs from local variables in modules only in the starting keyword **global**. Global variables can only changed by non-synchronized commands thus commands without a label in the square brackets at the beginning of a command. Additionally a module could contain declarations. To define the set of initial states there exists the **init** *predicate*; **endinit** construct. The *predicate* should be a predicate over all global and local variables. Any state that satisfies the predicate is an initial state. The construct **system** ... **endsystem** allows to modify the synchronization between modules described in the section 4.2.1. With the **rewards** *name predicate* : *value*; **endrewards** construct you are able to define costs or rewards for states satisfying

the *predicate*. A path have the costs/rewards of the sum of all states on the path. Optionally, rewards can be given names, to introduce multiple rewards. You can extend the rewards for transitions with the construct **rewards** *name* [*syncLabel*] *predicate*:*value*; **endrewards.** Each Transition, satisfying the predicate and having the same *syncLabel* produces the costs/rewards *value*.

Constants

The PRISM language additionally supports constants. A constant value can be expected anywhere a constant can be used. Constants are implemented thus:

```
const type name = value;
```

There are three types supported by constants **int**, **bool** and **double**. But **double** constants occur only in the $prob_i$ blocks of a command.

4.2.1 Synchronizing Modules

The *syncLabel* in a command can be used to force two or more modules to make a transition simultaneously. If in one module the *guards* of all commands with the same *syncLabel* are evaluated to false, this transition is disabled in all modules.

The construct system ... endsystem

The construct **system** ... **endsystem** is used to change the synchronization between modules. The following operators can be used:

- M_1 || M_2: This operator, which is the default operator, defines the full parallel composition of both modules. All commands with the same *syncLabel* are synchronized.
- M_1 ||| M_2: This operator defines the asynchronous parallel composition. Therefore, no synchronisation is made. Both modules behave fully interleaved.
- M_1 |[**a**,**b**,...]|M_2: This operator defines a restricted parallel composition of both modules. This means both modules are only synchronized by actions of the set {a,b,...}.
- **M** {**a**<−**b**,**c**<−**d**,...}: This operator renames the actions a,c,... into b,d,....
- **M** / {**a**,**b**,...}: All actions in the set {a,b,...} are not visible outside M. These actions are not used for synchronizing.

Example - twodice.pm

To understand the syntax and semantics of a file in PRISM language the following file will be described explicit:

```
1 dtmc
2
3 module twofoursideddice
```

```
4     die1 : [0 .. 4] init 0;
5     die2 : [0 .. 4] init 0;
6     result : [0 .. 8] init 0;
7
8  [] die1=0 -> 1/4:(die1'=1) + 1/4:(die1'=2) + 1/4:(die1'=3) +
       1/4:(die1'=4);
9  [] die2=0 -> 1/4:(die2'=1) + 1/4:(die2'=2) + 1/4:(die2'=3) +
       1/4:(die2'=4);
10 [try] die1!=0 & die2 != 0 -> result'= die1 + die2 & die1'=0 & die2'=0;
11 endmodule
12
13 rewards "trials"
14 (die1!=0&die2!=0) : 1;
15 endrewards
16
17 rewards "sum_of_results"
18 [try] true : die1 + die2;
19 endrewards
```

In this example file the probabilistic model of a discrete-time Markov chain is used (Line 1). The module name is **twofoursideddice**. There are three variables defined **die1** and **die2** are integers of range zero to four with zero as specified initial value. The variable **result** is a integer with range zero to eight where the initial value is zero (Lines 4-8). Three commands are defined. The first two commands assign the variables **die1** or **die2** the values one to four with equal probabilities (Lines 8 and 9). In this case each of this two commands are modeling a throw of a four-sided die. The last command sums up the results of the dice if they both are thrown (Line 10). At the end of the file there are two rewards defined. The reward "**trials**" is produced if both dice are thrown (Lines 13 to 15), it is a state reward. And the transition reward "**sum_of_results**" is produced if the command labeled with **try** is executed (Lines 17 to 19). This command is defined in line 10 and the value of the reward equals the new value of the variable **result**.

4.3 Property Specification

In Order to analyze a probabilistic model specified in the PRISM language, it is necessary to define some properties. The properties have to be given in a language based on the logic's PCTL [22] [8] for DTMCs and MDPs or CSL [4] [5] for CTMCs. Both are probability extensions of the temporal logic CTL. Files are given the extension .pctl for properties of DTMCs and MDPs and extension .csl for properties of CTMCs. This section intoduces the basic operators to specify properties. Afterwards a example file will demonstrate and explain the use of the operators. The basic syntax of a PRISM property *prob* is given by the following grammar:

```
prob := true | false | expr | !prob |
        prob & prob |
        prob | prob |
        prob => prob |
```

```
          P bound [ pathprob ] |
          S bound [ prob ]
bound := >=p | >p | <=p | <p | =? | min =? | max =?
pathprob := X prob | F prob | F time prob |
              G prob | G time prob |
              prob U prob |
              prob U time prob
time := >=t | <=t | [t,t]
```

where *expr* is a PRISM language expression evaluated to a Boolean, p is a PRISM language expression evaluated to a double in the range of [0,1] and t is a PRISM language expression evaluated to a non-negative double or integer.

The P Operator

The operator **P** is a Path operator to determine the probability of a path and is used in the form:

$$\mathbf{P}_{bound} \ [\ pathProb \ ]$$

A *bound* is one of the following modifier

- >=,>,<,<= p - a bound of the probability satisfied in *pathProb*, where p is a probability.
- =? - a query of the actual probability in *pathProb*
- min/max =? - a query of the worst/best case of the probability satisfied in *pathProb*

The last two modifiers are only supported by the probabilistic model MDP and the MDP supports only these two modifiers.

The S Operator

The long-run operator **S** is used to reason the steady-state behavior of a model. Currently, this operator is only supported by CTMC models. In [46] the **S** operator is defined.

The U Operator

This is the until operator, which can be modified with the optional *time*.

$$[\ prob_1 \ \mathbf{U}_{time} \ prob_2 \ ]$$

The property is true if $prop_2$ is true in some state in the path and $prob_1$ is true in all precedent states. The *time* modifier take one of the forms $>= t_1$, $<= t_1$ or $[t_1, t_2]$, where t_i is an integer. *Time* restricts the occurrence of $prob_2$, the property is only true if $prob_2$ occurs in the given interval.

The X Operator

The next operator is true if *prob* is true in the next state.

$$\mathbf{X} \ prob$$

The F Operator

```
F prob
```

F is the Finally operator and equals true if there is a path from the current state to a state where *prob* holds.

The G Operator

```
G prob
```

The general operator is evaluated to true if there is a path starting in the current state where *prob* holds in each state of the path.

The Operators R, C, I

The properties, which relate to the expected values of the mentioned rewards or costs can analyse through the R operator. This operator works similar to the S or P operators.

```
R bound [ rewardprop ]
```

where *bound* is defined as the modifier *bound* of the P operator and *rewardprob* is one of the four different types of reward properties defined in [27], namely:

- "reachability reward": F *prop*

  Associate a reward for each path of a model, which reached a state satisfying PRISM property *prob*. The reward is the sum of the state rewards of the path plus the rewards of each transition between these states.

- "instantaneous reward" : I=t

  where t is defined as above. Associates with a path the reward in the state of that path when exactly t time units have elapsed.

- "steady-state reward" : S

  Associates the reward of a long-run.

The Labels init and deadlock and Other Labels

It is also possible to define labels, which are shortcuts of a *prob*. Labels can be defined in the model or the property files. There are two predefined labels **init**, which is true in the initial states and **deadlock**, which holds in states where deadlocks are found.

Labels are defined using the keyword label, followed by a unique name in double quotes, and then an expression, which evaluates to a Boolean. The definition is illustrated in the following example:

```
label "a holds" = a=true;
label "initial states or a equals 42" = "init" | a=42;
```

Example - twodice.pctl

This example describes a property specification file for the above mentioned twodice.pm file. In this file there are definitions for a constant, a label and four properties.

```
1  const int x;
2
3  label "init_done" = die1!=0&die2!=0;
4
5  P=? [ !"init_done" U die1+die2=x & "init_done" ]
6
7  P=? [ true U result!=0 ]
8
9  R{"trials"}=? [ F result=x ]
10
11 R{"sum_of_results"}=? [ F result=x ]
12
13 "init" => P>=1.0 [ F result > 0]
```

In line

1 the constant x is defined, but there are no value assigned to x.

3 the label "init_done" is defined. All states that fulfil **die1**!=0&**die2**!=0 are marked with this label.

5 the first real property specification appears. Here is a query for the probability defined. The requested probability is the probability that the variable **result** takes the value x.

7 this probability query ask for the probability that the variable **result** will be changed.

9 a reward query is defined. The result is the number of trials till **result** takes the value x.

11 a reward query is defined too. The result is the sum of the **result** variables of all trials till **result** equals x.

13 this property holds if the probability, that the variable **result** is at some point of time grater than 0, is at least 1.0 for all initial states.

This properties can be analyzed by PRISM as well. The output of a experiment over constant x from 1 to 8 for each line are:

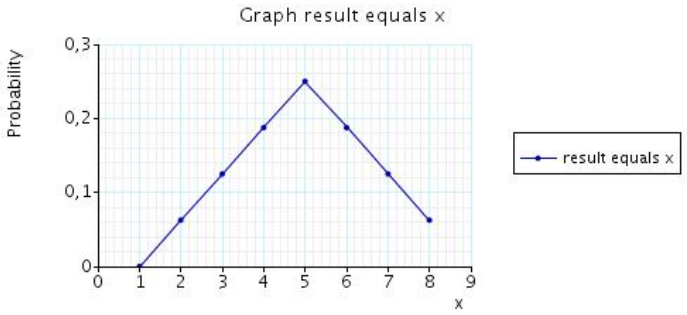

Figure 4.1: Plot generated by PRISM

This plot shows the probability that the variable **result** takes the value x. This result is not surprising as the expected value of a four-sided die is $2,5$ and so the expectation value of two four-sided dice is 5. This explains the maximum at $x = 5$. Also the value $x = 1$ can not reached with two four-sided dice hence the probability $P(1) = 0$.

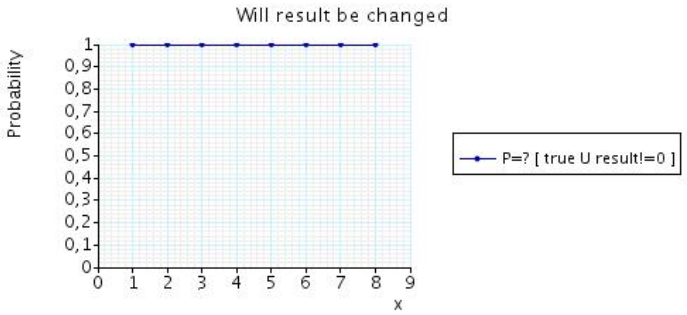

Figure 4.2: Plot generated by PRISM

It is easy to see that the result of this experiment does not depend on the constant x. The result is not unexpected because after the execution of the two commands in Lines 8 and 9 of the twodice.pm file the only possible command, which is enabled is the command in Line 10. But this commands assigns the variable **result** the value from the sum of the variables $die1$ and $die2$. Due to the fact that these values are per definition grater or equal to one the variable **result** is changed with a probability 1.0. In other words, the variable result will always be changed at some point of time.

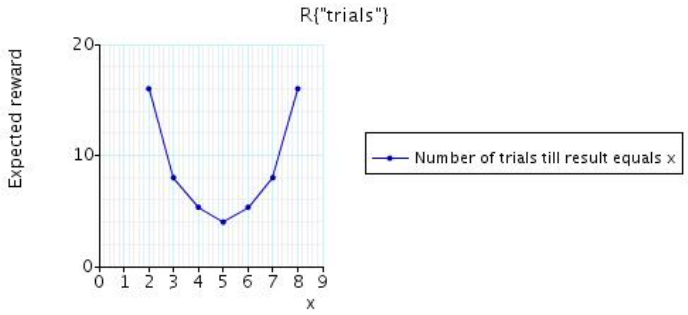

Figure 4.3: Plot generated by PRISM

This plot shows the trials are needed until the variable **result** takes the value x. In this plot it is obvious that the result of this experiment correlates with the expectation of two four-sided dice. The number of trials declines if x comes closer to the expectation.

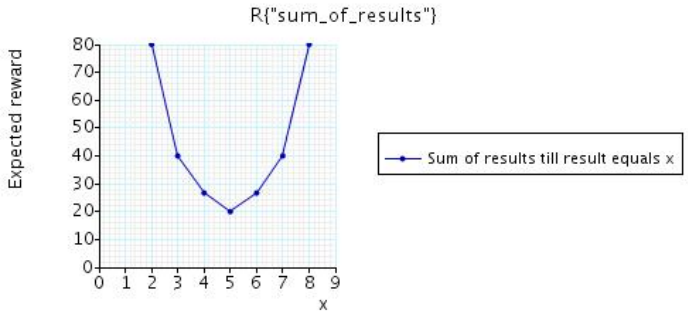

Figure 4.4: Plot generated by PRISM

This plot shows the sum of all results until the variable **result** takes the value x. The number of trials declines, if x come closer to the expectation, this is because the sum over all results declines in a similar manner.

Simulator Screenshot

This is a screenshot of the PRISM simulator, which simulates the above mentioned example. On the left hand-side some simulation options can be found. There, it is possible to choose an arbitrary number of steps to simulate automatically by taking the option *AutoUpdate*. The option *DoUpdate* allows the user to choose an active command for himself. The latter can be used to simulate a certain example. Below this, defined path formulas and labels are listed. Additionally, it is shown if a path formula is satisfied in the current path and if the current state fulfils a label definition. In this screenshot the current state fulfils the **init_done** label. At the top some statistical information about the current probabilistic model, the current path length and the reward of the current path are listed. With the button *NewPath* it is possible to start

a new simulation and define a new initial state. The restarting of the current path is made by taking the *ResetPath* button. To save the current path, the *ExportPath* button exists. The current path is listed in the center. The list contains each variable and their values for each state and the rewards.

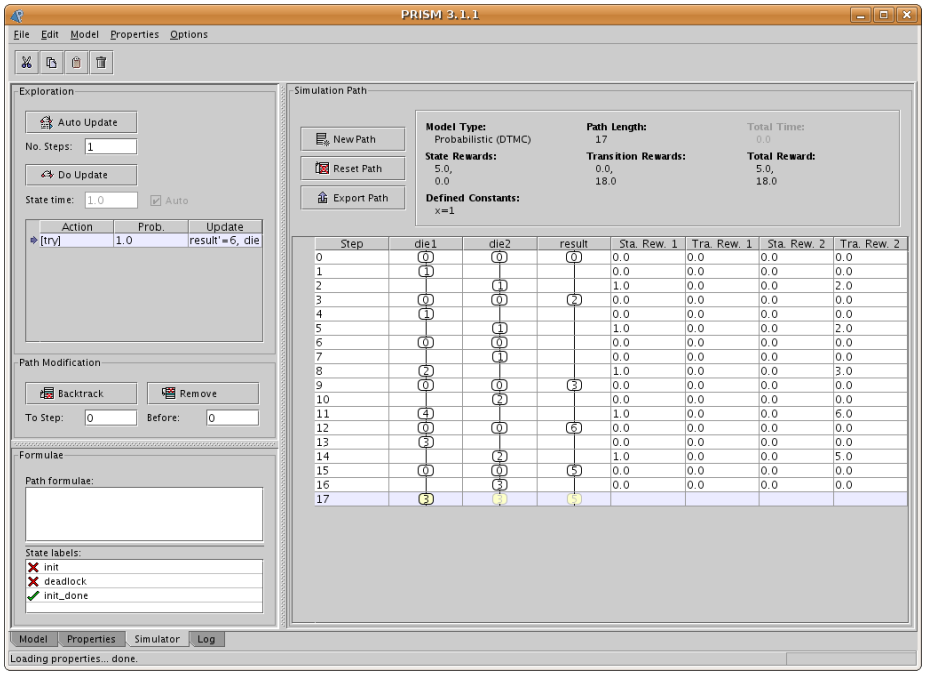

Figure 4.5: PRISM Simulator Screenshot

Outline

Some additional information is needed to permit probabilistic model checking. These information will be saved in an additional file, which is introduced and described in the next chapter. Afterwards, the compiler Opal is presented.

Chapter 5

The Compiler Opal

This chapter is the main part of the thesis and describes the compiler Opal. First, a needed aditionally file, the Opal Probability File, is introduced. Afterwards an overview of the information flow that allow probabilistic model checking for synchronous programs is given. At the end of the chapter some code snippets are presented and described. An Example will show, what the compiler Opal is able to do.

5.1 Opal Probability File

To verify a synchronous programs in a probabilistic way the behaviour of the input variables has to be described. Therefore an additionally file is needed as AIF does not support this. Here, the so called Opal probability file(OPF) will be introduced. This file is also an XML file. This file includes not only probabilities for the input variables but there is the possibility to define some code snippets to simulate errors in the execution or define some rewards and costs to analyze the program more precisely too. The probabilities for the input variables are defined through probability specifications. They will be saved in a list and will be emptied step by step.

5.1.1 Probability Specifications

For each variable it is possible to specify the probability for each value of the range. Therefore probability specifications are introduced. The starting is that there is a variable of a certain type with a domain and a specification list. Now there are several options to define the probabilities for each value in the domain.

- <**Pequal**/>
- <**Interval prob**="*probability*" **begin**="*start*" **end**="*end*">

  spec_list

  </**Interval**>
- <**Pfkt**>*MLfunction*</**Pfkt**>
- <**PList**>

- <**Repeat**/>

<**Pequal**/>

If a <**Pequal**/> specification is read from the specification list all, values in the current domain are equiprobable. An interval defined later in the specification list is excluded from the domain and takes the probability of the attribute **prob** if specified else the probability of one single value. The <**Pequal**/> statement forbade the later use of the <**Pfkt**> statement for this variable except in <**intervals**> because it does not make sense.

<**Interval**>

An interval divides the domain of the variable in up to three part domains, which are computed separately. Firstly all values less than *start*, secondly the interval itself and then the values grater than *end*. All undefined probabilities for values less than *start* equal zero or are equiprobable in a <**Pequal**/> environment. The probabilities for the values between *start* and *end* are defined in *spec_list*. All probabilities greater than *end* are defined in the specification list or if the list is empty the probabilities will equal zero or are equiprobable in a <**Pequal**/> environment.

The delimiter **prob** is optional in a <**Pequal**/> environment. If given the sum of all probabilities in the interval equals the *probability*. Otherwise the sum of all probabilities in the interval equals the probability of one single value of the domain.

Example

```
<Interval prob="1.00" begin="1" end="8">
    <Interval prob="0.5" begin="2" end="3">
        <Pequal/>
    </Interval>
    <Pequal/>
    <Interval begin="5" end="6">
        <Pequal/>
    </Interval>
</Interval>
```

The first interval starts at value one. Hence, all values less than one do not occur and have the probability zero. The second interval splits the first interval into three parts. These are: values between one and less than two, which have the probability zero too, the interval itself with the values two and three. These are equiprobable through the <**Pequal**/> tag with the probability 0.25 each because the probability specified in the attribute **prob** is distributed to all elements in the interval. Now the second <**Pequal**/> mode starts which means that all values between four and eight are equiprobable, except for the third interval, which takes the probability of a single value in the <**Pequal**/> mode. Hence the probability for the values four, seven, eight and the interval is 0.25 each. The values five and six have as a result of this the probability 0.125. The probabilities for values greater than eight are zero because the specification list is empty and no <**Pequal**/> statement occurred outside the first <**interval**>.

<Pfkt>

If a <**Pfkt**> tag is read the ML function defined in $MLfunction$ is executed once for each value of the actual domain except for the values inside further intervals in the specification list. Thereby the parameters of the function call are the regarded value[1] itself, the maximum of unused probability in the domain and the number of calls of the function in this domain. So it is possible to define personalized distribution functions. The <**Pfkt**/> statement forbade the later use of the <**Pequal**/> statement for this variable except in <**intervals**> because it does not make sense.

Example

```
<Pfkt>
"fn (value::maxprob::calls::[]) =>
        let
            val maxProb =
                getOpt(Real.fromString(maxprob), 0.0)
            val maxCalls =
                getOpt(Real.fromString(calls), 1.0)
        in
            maxProb/maxCalls
        end"
</Pfkt>
```

Note that the parameter of the function is a string list because it is compiled and called at run-time.

<PList>

In the case a variable is translated into a list of sub-variables there is a need to split the specification list as well. This is done by the tag <**PList**>. Each specification list surrounded with this tag corresponds to a sub-variable. If the list of <**PList**> tags is shorter than the list of sub-variables the list specified in the last tag is used or the whole specification list if there is no <**PList**> tag.

Example

In the next two examples the input variable **vector** is a bit vector of length three.

```
<input name="vector" >
    <PList><Pequal/></PList>
    <PList>
        <Pfkt>"fn _ => 0.5"</Pfkt>
    </PList>
</input>
```

The variable **vector** is translated into **vector_bv_1**, **vector_bv_2** and **vector_bv_3**, which is described in Section 5.3.1. The first <**PList**> tag includes the specification list for the first sub-variable, namly the list {<**Pequal**/>}. The second and

[1] In case of a Boolean the value is zero for $false$ or one for $true$

last <**PList**> tag is used for the last two sub-variables because the list of <**PList**> tag is shorter than the list of sub-variables.

```
<input name="vector" >
    <Pequal/>
</input>
```

In this example each sub-variable get assigned the list {<**Pequal**/>}. The list {<**Pequal**/>} could be replaced with a arbitrary specification list, which does not include the <**PList**> tag.

<Repeat/>

To repeat a <**PList**> tag without copying the whole tag, the tag <**Repeat**/> exists. This tag is a placeholder and returns the same specification list as the previous <**PList**> tag.

Example

In the next example the input variable **vector** is a bit vector of length four.

```
<input name="vector" >
    <PList><Pequal/></PList>
    <Repeat/>
    <PList>
        <Pfkt>"fn _ => 0.5"</Pfkt>
    </PList>
    <Repeat/>
</input>
```

The variable **vector** is translated into **vector_bv_1**, **vector_bv_2**, **vector_bv_3** and **vector_bv_4**. The first <**PList**> tag includes the specification list for the first sub-variable, namly the list {<**Pequal**/>}. This list is also assigned to the second sub-variable by the <**Repeat**/> tag. The last two sub-variables get assigned the list including the <**Pfkt**> tag with the same reason.

5.1.2 Syntax

First the corresponding AIF file is mentioned by the tag <**Aif file**="*path*"> Afterwards there is the section <**Probabilities**>, which contains for every input variable a section <**input name**="*variable*">, which contains a probability specifications list described above. In the section <**Vars**> declared variables are added to the variables of the AIF file. PRISM language commands described in section <**Commands**> are added at the end of the module described in the AIF file. The last two sections are useful to simulate some execution errors or save some values in new variables. To define some extra rewards, costs or add other PRISM language code outside the module there exists the <**Code**> section. The behaviour of the last three mentioned sections is exactly described in section 5.3.6.

5.1.3 Example OPF File - bvTest.opf

This example includes a OPF file for the example given in 2.3.6

```
<?xml version="1.1"?>

<Aif file= "../examples/bvtest.aif" >
<Probabilities>
    <input name="mode"><Pequal/></input>
    <input name="a">
        <PList><Pequal/></PList>
        <Pfkt>"fn (value::tl) =>
            case getOpt(Int.fromString(value),0) of
                0 => 0.25
                | 1 => 0.75"
        </Pfkt>
    </input>
    <input name="b">
        <PList><Pequal/></PList>
        <Pfkt>"fn (value::tl) =>
            case getOpt(Int.fromString(value),0) of
                0 => 0.0
                | 1 => 1.0"
        </Pfkt>
    </input>
</Probabilities>
<Vars>new_Var: bool; //A new variable</Vars>
<Commands>//section for self-defined Commands</Commands>
<Code>
rewards
true : 2
endrewards
//A self-defined reward.
</Code>
</Aif>
```

Probabilities for the input variables are defined in the section <**Probabilities**>. In the section <**Vars**> a new variable **new_Var** is introduced. A new reward is defined in the <**Code**> section, which rewards each state with the cost of two.

Outline

Now everything needed to allow probabilistic model checking for synchronous programs is explained. The compiler Opal introduced in the following sections transforms AIF and OPF files into PRISM language files. The next section describes the information flow, starting in a Quartz program up to the model checker PRISM. In the subsequently section, the proceeding of Opal from the input files towards the output file, which allows probabilistic model checking with assistance of the tool PRISM, is described. Afterwards, some code snippets are reviewed.

5.2 The Information Flow

To allow probabilistic model checking for synchronous programs the compiler Opal was developed. The information flow from a synchronous program given in Quartz to the probabilistic model checker PRISM is:

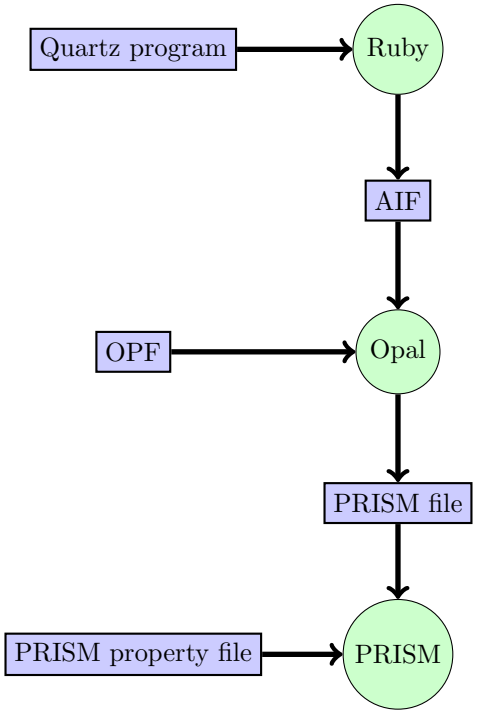

Figure 5.1: Information Flow

The user has to define the Quartz program, a OPF file and the PRISM property file. The AIF file is generated, by Ruby, through compiling the Quartz program. The compiler Opal uses the generated AIF file and the user defined OPF file to generate the PRISM file. The model checker PRISM takes the generated PRISM file and the user defined PRISM property file as inputs. Afterwards, PRISM is able to verify if the model, which is defined in the PRISM file, satisfies the properties, which are defined in the PRISM property file. The latter file is not needed in case of simulation.

5.3 Program Procedure

As previously mentioned, it is possible to translate every synchronous program into a form which corresponds to the AIF. Therefore, it is possible to translate each synchronous programs written in an arbitrary synchronous language into the PRISM language with the techniques described in this section. Opal is a program to translate a AIF file to a PRISM file. This allows all synchronous programs, primarily Quartz programs, to be verified in a probabilistic way.

However, for this some probability information is needed too. This information is read from a so called OPF file described in last chapter.

The program procedure is as follows:

1. Rename variables and the corresponding variables in the actions.
2. Analyze the data dependency and identify non-conflicting immediate actions.
3. Find cyclic dependant immediate actions, modify them and go to step two otherwise continue.
4. Rename the target value of each delayed action to archieve independence between these actions. Hence there is no need to rearrange these actions.
5. Append new actions, which assigns the old targets the values of the new targets of step four.
6. Translation of the actions into PRISM commands
7. Analyse the OPF
8. Compose the whole PRISM file

The renaming of variables is required because Quartz supports some data types, which are not supported by PRISM. For example finite bit vectors are not supported, but a bit vector consists of a finite number of Boolean. Hence bit vectors are split in the Boolean components and each Boolean has the name of the bit vector as name prefix and the construct _bv_i, where i represents the position in the bit vector.

In step two there are a list of sorted actions and a set of unsorted or not visited actions. All actions read out of the AIF are put into the set of unsorted actions. Each action in the set that has no dependencies other than input variables or variables, which have no more actions in the set, is identified and putted at the end of the sorted list. This sort sequence is redone until the set does not change anymore or is empty.

If there are still actions in the set, a cyclic dependency occurs in the set. This is not a real cycle however, because in immediate actions there are no real cycles as this is a contradiction to a causal program (2.1.6). This problem is discussed later on. If there is a cycle, the cycle is removed by modifying some actions in the set and afterwards the analysis starts at step two again.

Delayed actions are easy to handle because the only thing to do is to store the value of the next step in a new variable and after the computation of all delayed actions, the stored value is written back into the correct variable.

After the sorting of the variables, which is a serialization of the guarded actions, the guarded actions are transformed to PRISM commands.

Some commands additionally added, thus the analysis of the OPF file.

At the end the whole PRISM file has to composed.

5.3.1 Rename Variables

Because PRISM only supports finite data types, only AIF files without infinite data types are able to be translated, but there are still data types not supported by PRISM. These are bit vectors, finite natural numbers, arrays and tuple. But these can be translated into supported data types.

An integer with size n is translated into an integer with range $-2^{(n+1)}$ to 2^n as follows:

int[**n**] **number** ⇒ **number**:[$-2^{(n+1)}$ .. 2^n]

A natural number with size n is translated into an integer with range 0 to $2^n - 1$ as follows:

nat[**n**] **number** ⇒ **number**:[0 .. $2^n - 1$]

A bit vector with length n is translated as follows:

bv[**n**] **vector** ⇒ **vector_bv_1**:**bool** ... **vector_bv_n**:**bool**

An array with length n and type α is translated as follows:

α **field**[**n**] ⇒ **field_arr_1**:α ... **field_arr_n**:α

Afterwards the translation is done for the variables of type α

A tuple is translated as follows:

$\alpha_1 * \ldots * \alpha_n$ **t** ⇒ **t_tup_1**:α_1 ... **t_tup_n**:α_n

Afterwards the translation is done for the variables of type α_i

The variable names in each action will also be replaced if necessary.

Example

Looking at following example: **bv**[3] **example**[2] is an array of bit vectors and is translated into:

example_arr_1_bv_1:**bool**
example_arr_1_bv_2:**bool**
example_arr_1_bv_3:**bool**
example_arr_2_bv_1:**bool**
example_arr_2_bv_2:**bool**
example_arr_2_bv_3:**bool**

Clearly the new variables are Boolean and supported by PRISM. The actions still have to change. For each variable renamed by opal a tuple, which contains the old name and a list of the new names will be stored to rename the variables in the actions.

5.3.2 Rename the Variables in Guarded Actions

The renaming of variables in a guarded action is split into two steps. Firstly, each variable except the label variables are replaced with the list of new variable names. Then, for each list item, the action is copied and the first element of each list is inserted. Note that each list must have the same length otherwise

a type inconsistency is occurred, which implies that the AIF file is incorrect and an exception is raised. Afterwards the new actions are integrated into one PRISM command.

Examples

Looking at the following two examples: First example is an addition of two integers and the second is a bit-wise disjunction over two bit vectors *param*1 and *param*2 of length two stored in an other bit vector *result*. The meaning of the *ctrl* variable in both examples will be described later on.

Example One

An abridgment of the AIF file is:

```
<actions>
    <var name="result"/>
        <immediate>
            <guard>
                <blVal><var name="guard"/></blVal>
            </guard>
            <intAdd>
                <intVal><var name="param1"/></bvVal>
                <intVal><var name="param2"/></bvVal>
            </intAdd>
        </immediate>
</actions>
```

And the counterpart in the PRISM language:

```
[](ctrl=8&guard)->result'=(param1 + param2) & (ctrl'=9);
```

Example Two

An abridgment of the AIF file:

```
<actions>
    <var name="result"/>
        <immediate>
            <guard>
                <blVal><var name="guard"/></blVal>
            </guard>
            <bvOr>
                <bvVal><var name="param1"/></bvVal>
                <bvVal><var name="param2"/></bvVal>
            </bvOr>
        </immediate>
</actions>
```

And the counterpart in the PRISM language:

```
[] (ctrl=4&guard) -> result_bv_2'=
            (param1_bv_2 | param2_bv_2)
            & result_bv_1'=(param1_bv_1 | param2_bv_1)
            & (ctrl'=5);
```

Note that the structure in the AIF file of both examples is very similar but through the types of the variables the translation into the PRISM language is quiet different.

5.3.3 Resolve the Data Dependency in Immediate Actions

It is possible that there are actions in a cyclic dependency as described in Section 2.1.6. The actions in this cycle depend on each other but at run-time at least one of the guards of the actions is false. If this is not the case, the program is non-deterministic, which is a contradiction to synchronous languages. Due to the fact that Opal modifies the actions that way so that a case differentiation will take place. The best way to explain this is with an example:

Example

If the Boolean mode is held variable z depends on variable x and x depends on y. If Boolean mode is not held y depends on z. But at compile time there is a data dependency cycle because variable z depends on variable x and x depends on y and y depends on z. Opal resolves this problem this way so that for each action in the cycle the other actions are copied and the guard of the regarded action is negated and append in the other guards. After this, the cyclic dependency for this cycle is resolved and Opal starts the analysis at step two again.

```
<actions>
    <var name="z"/>
        <immediate>
            <guard>
                <blVal><var name="mode"/></blVal>
            </guard>
            <blVal><var name="x"/></blVal>
        </immediate>
</actions>
<actions>
    <var name="x"/>
        <immediate>
            <guard>
                <blVal><var name="mode"/></blVal>
            </guard>
            <blVal><var name="y"/></blVal>
        </immediate>
</actions>
<actions>
    <var name="y"/>
        <immediate>
            <guard>
```

```
                <blNot>
                    <blVal><var name="mode"/></blVal>
                </blNot>
            </guard>
            <blVal><var name="z"/></blVal>
        </immediate>
</actions>
```

This three actions are translated into six actions. Afterwards duplicates are removed like the last action.

```
<actions>
    <var name="z"/>
        <immediate>
            <guard>
                <blVal><var name="mode"/></blVal>
            </guard>
            <blVal><blITE>
                <blVal><var name="mode"/></blVal>
                <blVal><var name="x"/></blVal>
                <blVal><var name="z"/></blVal>
            </blITE></blVal>
        </immediate>
</actions>
<actions>
    <var name="z"/>
        <immediate>
            <guard>
                <blVal><blNot>
                    <var name="mode"/>
                </blNot></blVal>
            </guard>
            <blVal><blITE>
                <blVal><var name="mode"/></blVal>
                <blVal><var name="x"/></blVal>
                <blVal><var name="z"/></blVal>
            </blITE></blVal>
        </immediate>
</actions>
<actions>
    <var name="x"/>
        <immediate>
            <guard>
                <blVal><var name="mode"/></blVal>
            </guard>
            <blVal><blITE>
                <blVal><var name="mode"/></blVal>
                <blVal><var name="y"/></blVal>
                <blVal><var name="x"/></blVal>
            </blITE></blVal>
        </immediate>
```

```
</actions>
<actions>
    <var name="x"/>
        <immediate>
            <guard>
                <blVal><blNot>
                    <var name="mode"/>
                </blNot></blVal>
            </guard>
            <blVal><blITE>
                <blVal><var name="mode"/></blVal>
                <blVal><var name="y"/></blVal>
                <blVal><var name="x"/></blVal>
            </blITE></blVal>
        </immediate>
</actions>
<actions>
    <var name="y"/>
        <immediate>
            <guard>
                <blVal><blNot>
                    <var name="mode"/>
                </blNot></blVal>
            </guard>
            <blVal><blITE>
                <blNot><blVal>
                    <var name="mode"/>
                </blVal></blNot>
                <blVal><var name="z"/></blVal>
                <blVal><var name="y"/></blVal>
            </blITE></blVal>
        </immediate>
</actions>
```

The next action is generated but also deleted because it is the same as the previous.

```
<actions>
    <var name="y"/>
        <immediate>
            <guard>
                <blVal><blNot>
                    <var name="mode"/>
                </blNot></blVal>
            </guard>
            <blVal><blITE>
                <blNot><blVal>
                    <var name="mode"/>
                </blVal></blNot>
                <blVal><var name="z"/></blVal>
                <blVal><var name="y"/></blVal>
```

```
            </blITE></blVal>
        </immediate>
</actions>
```

It is easy to see that the cycle is resolved and the analysis has to start again to sort the remaining actions.

5.3.4 Resolve the Data Dependency in Delayed Actions

To resolve the dependency in delayed actions, new variables are introduced for each variable changed by a delayed action. The new variables have the same name with an additional "next_" prefix. And in each delayed action the target variable is replaced with the new variable. Now there is no need to sort the delayed actions. They are completely inserted in the sorted list. Afterwards new actions are inserted in the sorted list, which assigns the variables the value of the new ones.

Example

The delayed action

```
<actions>
    <var name="ell1"/>
        <delayed>
            <guard>
                <blVal><var name="ell1"/></blVal>
            </guard>
            <blTrue/>
        </delayed>
</actions>
```

is translated into these two actions. The last one is append after all delayed actions to resolve the dependency.

```
<actions>
    <var name="next_ell1"/>
        <immediate>
            <guard>
                <blVal><var name="ell1"/></blVal>
            </guard>
            <blTrue/>
        </immediate>
</actions>
<actions>
    <var name="ell1"/>
        <immediate>
            <guard>
                <blTrue/>
            </guard>
            <blVal><var name="next_ell1"/></blVal>
        </immediate>
</actions>
```

5.3.5 Translation of Actions into Commands

Now there are only sorted actions without unsolved dependencies and with supported types. Now it is possible to translate the guarded actions directly into PRISM commands. Each guarded action generates one command where the *guard* is translated to the PRISM *guard* and the *variablename* together with the *executionpart* results in a PRISM $update_i$ block. Additionally there is a counter called *ctrl*, which takes care that the serialized actions are executed in the right order. At the end of each $update_i$ block, the counter *ctrl* is updated. And the *guard* is extended with a *ctrl* query. For each command there exists a command with negated guard to update the *ctrl* variable. Because this global variable cannot be updated by a synchronized command so this extra command is needed.

Example

For Example the translated commands for the 5 actions are given in the Example above are:

```
[] (ctrl=6&mode) -> x'=(mode?y:x) & (ctrl'=7);
[] (ctrl=6&!(mode)) -> (ctrl'=7);

[] (ctrl=7&mode) -> z'=(mode?x:z) & (ctrl'=8);
[] (ctrl=7&!(mode)) -> (ctrl'=8);

[] (ctrl=8&!(mode)) -> z'=(mode?x:z) & (ctrl'=9);
[] (ctrl=8&mode) -> (ctrl'=9);

[] (ctrl=9&!(mode)) -> y'=(!(mode)?z:y) & (ctrl'=10);
[] (ctrl=9&mode) -> (ctrl'=10);

[] (ctrl=10&!(mode)) -> x'=(mode?y:x) & (ctrl'=11);
[] (ctrl=10&mode) -> (ctrl'=11);
```

Transformation of Guarded Actions

The transformation of guarded actions into PRISM commands can be summarized:

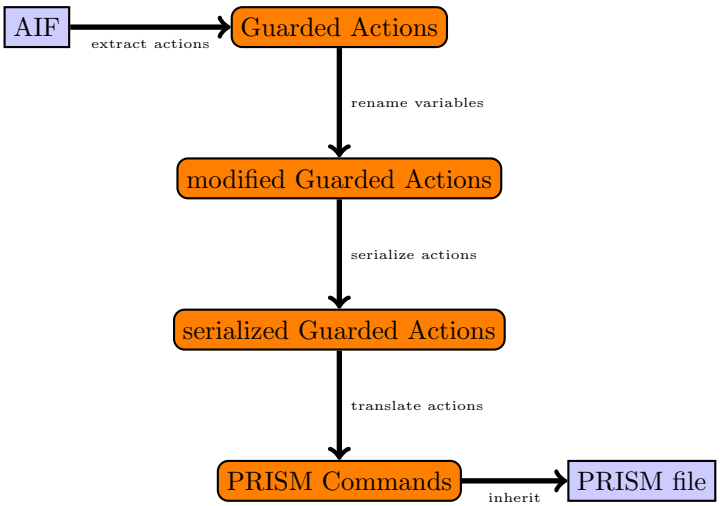

Figure 5.2: Transformation of Guarded Actions

5.3.6 Compose the Whole PRISM File

Opal defines first the probabilistic model *dtmc* in the PRISM file. Then the module with the same name described in the AIF file is generated. Therefore all translated variables are written down. Afterwards, the variables declared in the OPF in the <**Vars**> section are appended. Then the commands, which fulfil the input probabilities defined in the OPF in the <**Probabilities**> section are generated. Afterwards, the sorted and translated actions take the main part of the module. At the end of the module, commands declared in the OPF in the <**Commands**> section are added. The code snippets defined in the OPF in the <**Code**> section are inserted after the module declaration. At the end of the file, some static PRISM code is added to define the control signals and their behaviour.

Example

In this example a simple Quartz program of a function schedule is given. First, the extracted guarded actions are shown. Then, the resulting AIF file is shown. Afterwards, the OPF file, which includes for each input variable a probability specification list and some additional code, is presented. At the end, the compiled PRISM language file, which is based on the AIF and OPF file, is presented.

```
module P(bool mode, nat[10] z, nat[10] &x, nat[10] &y){
    loop{
        pause;
        if(mode)
            then x=y-z
            else y=x+z;
    }
}
```

This Quartz file defines following guarded actions:

```
(mode&ell1 , x=y-z)
(¬mode&ell1 , y=x+z)
(ell1 , emit next ell1)
(_goEta , emit next ell1)
```

These guarded actions are translated into following AIF file:

```
<?xml version="1.1"?>
<aif>
<module name="SimpleSchedule">
<declarations>
    <input name="mode"><bool/></input>
    <input name="z"><nat size="10u"/></input>
    <output name="x" storage="event"><nat size="10u"/></output>
    <output name="y" storage="event"><nat size="10u"/></output>
    <label name="ell1"/>
</declarations>
<controlSignals>
    <goAlpha name="_goAlpha"/>
    <goEta name="_goEta"/>
    <abort name="_abort"/>
    <suspend name="_suspend"/>
    <inst name="_inst"/>
    <inside name="_inside"/>
    <term name="_term"/>
</controlSignals>
<definitions>
    <def name="_var1">
        <natSub>
            <natVal><var name="y"/></natVal>
            <natVal><var name="z"/></natVal>
        </natSub>
    </def>
    <def name="_var2">
        <natAdd>
            <natVal><var name="x"/></natVal>
            <natVal><var name="z"/></natVal>
        </natAdd>
    </def>
</definitions>
<dataFlow>
    <depth>
    <actions><var name="x"/>
        <immediate>
            <guard>
                <blVal><var name="mode"/></blVal>
            </guard>
            <natVal><var name="_var1"/></natVal>
        </immediate>
```

```
    </actions>
    <actions><var name="y"/>
            <immediate>
                <guard>
                    <blNot>
                    <blVal><var name="mode"/></blVal>
                    </blNot>
                </guard>
                <natVal><var name="_var2"/></natVal>
            </immediate>
    </actions>
    </depth>
</dataFlow>
<controlFlow>
    <surface>
    <actions><var name="ell1"/>
        <delayed>
            <guard>
                <blVal><var name="_goEta"/></blVal>
            </guard>
            <blTrue/>
        </delayed>
    </actions>
    </surface>
    <depth>
        <actions><var name="ell1"/>
        <delayed>
            <guard>
                <blVal><var name="ell1"/></blVal>
            </guard>
            <blTrue/>
        </delayed>
    </actions>
    </depth>
</controlFlow>
</module>
</aif>
```

Next a possible, user defined, OPF file is:

```
<?xml version="1.1"?>

<Aif file= "../examples/SimpleSchedule.aif" >
<Probabilities>
<input name= "mode" > <Pequal/> </input>
<input name="z" >
        <Interval prob="1.0" begin="2" end="3"><Pequal/></Interval>
</input>
</Probabilities>
<Vars>Test : bool init false;</Vars>
<Commands>
```

```
[] !Test -> Test'=true;
</Commands>
<Code>
rewards
true : 2;
endrewards
</Code>
</Aif>
```

This two files are translated into following PRISM file:

```
// File automatically created by opal

 //discrete-time Markov chains (probabilistic)
dtmc
module SimpleSchedule
    z : [0 .. 10];
    mode : bool;
    y : [0 .. 10];
    next_y : [0 .. 10];
    x : [0 .. 10];
    next_x : [0 .. 10];
    ell1 : bool;
    next_ell1 : bool;
    //Variables of the ProbFile
    Test : bool init false;
    ctrl : [0 .. END_STEP] init 0;
    ctrl_lock : [0 .. END_STEP] init 0;

[initial] _goAlpha=true -> (ctrl' = 1);
[end_step] ctrl=END_STEP -> (ctrl' = 1) & ctrl_lock' = 0;
    [] ctrl=1 -> 0.5:(z'=2)&(ctrl'=ctrl+1)+0.5:(z'=3)&(ctrl'=ctrl+1);
    [] ctrl=2 -> 0.5:(mode'=true)&(ctrl'=ctrl+1)
            +0.5:(mode'=false)&(ctrl'=ctrl+1);

// immidiate commands
[] (ctrl=3&mode) -> x'=(mode?(y - z):x) & (ctrl'=4);
[] (ctrl=3&!(mode)) -> (ctrl'=4);

[] (ctrl=4&!(mode)) -> y'=(!(mode)?(x + z):y) & (ctrl'=5);
[] (ctrl=4&mode) -> (ctrl'=5);

// delayed commands
[] (ctrl=5&_goEta) -> next_z'=1 & (ctrl'=6);
[] (ctrl=5&!(_goEta)) -> (ctrl'=6);

[] (ctrl=6&ell1) -> next_ell1'=true & (ctrl'=7);
[] (ctrl=6&!(ell1)) -> (ctrl'=7);

[] (ctrl=7&_goEta) -> next_ell1'=true & (ctrl'=8);
[] (ctrl=7&!(_goEta)) -> (ctrl'=8);
```

```
[] (ctrl=8&_goEta) -> z'=next_z & (ctrl'=9);
[] (ctrl=8&!(_goEta)) -> (ctrl'=9);

[] (ctrl=9&_goEta) -> next_z'= & (ctrl'=10);
[] (ctrl=9&!(_goEta)) -> (ctrl'=10);

[] (ctrl=10) -> (ell1' = next_ell1) & (ctrl'=11);

[] (ctrl=11) -> (next_ell1' = false ) & (ctrl'=12);

//Commands added through OPF file
[] !Test -> Test'=true;

endmodule

rewards
true : 2;
endrewards

const int END_STEP = 12;

module controlSignals
_goAlpha : bool init true;
_goEta : bool init false;
_abort : bool init false;
_suspend : bool init false;
_inst : bool init false;
_inside : bool init false;
_term : bool init false;

[initial] _goAlpha=true & _goEta=false -> (_goEta' = true);
[end_step] (_goEta=true) & (ctrl=END_STEP) -> (_goEta' = false) &
    (_goAlpha'=false) & (_inside' = true);
[end_step] (_goEta=false) & (ctrl=END_STEP) -> (_inside' = true);
endmodule
```

Outline

Now the idea Opal based on is presented. For a better understanding of the proceeding the important components of the program code, which is written in Moscow ML, are discussed.

5.4 Code Review

5.4.1 Starting Opal

To start Opal the command

```
./opal OPF file
```

is used. Then Opal reads the OPF file and the AIF file specified there.

5.4.2 Reading Files

The compiler Opal read data from two intput files. The already existing AIF file, which is generated by Ruby and the new for Opal developed OPF file, which the user has to write.

AIF File

To read the AIF file the already existing AIF parser of the Embedded Systems group is used. This parser returns a tree, which represents the data of the AIF file.

OPF File

The OPF file is a XML file like the AIF file an the syntax is described in section 5.1.2. For this file a Lex/Yacc parser was defined. The used functions **mkId**, **mkFKT**, **mkpath**, **mkNatlit**, **mkIntlit** and **mkReallit** only format their parameter. The **Token** definition in the lexer file ConfigParser.lex:

```
rule Token = parse
    (* --- whitespaces --- *)
      "<?xml version=\"1.1\"?>" { Token lexbuf } (* skip prolog *)
    | [' ' '\t'  '\012'] { Token lexbuf } (* skip whitespace *)
    | ['\n' '\r'] { Token lexbuf } (* skip newlines *)
    (* --- parsing special sections --- *)
    | "<!--" {  commentStart := getLexemeStart lexbuf;
                commentDepth := 1;
                SkipComment lexbuf; Token lexbuf }
    | "<Pfkt>" { FKT(mkFKT (Funktion lexbuf)) }
    | "<Commands>" { COMMANDS(Commands lexbuf) }
    | "<Code>" { CODE(Code lexbuf) }
    | "<Vars>" { VARIABLES(Vars lexbuf) }

    (* --- attributes --- *)
    | "name=" { attNAME }
    | "size=" { attSIZE }
    | "begin=" { attBEGIN }
    | "end=" { attEND }
    | "file=" { attFILE }
    | "prob=" { attPROB }

    (* --- declarations --- *)
    | "<Probabilities>" { PSTART }
    | "</Probabilities>" { PEND }
    | "<input" { beINPUT }
    | "</input" { enINPUT }
```

```
    (* --- identifiers --- *)
    | "<Aif"                  { beAif }
    | "</Aif"                 { enAif }
    | "<Pequal"               { bePE }
    | "<Interval"             { beI }
    | "</Interval"            { enI }

    (* --- general tokens --- *)
    | ">"                     { xx }
    | "/>"                    { ex }

    | "\""['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']*"\""
                              { ID (mkId (getLexeme lexbuf)) }
      | "\""['0'-'9']+"u\""
                              { NATLIT (mkNatlit (getLexeme lexbuf)) }
      | "\""['+' '-' '~']?['0'-'9']+"\""
                              { INTLIT (mkIntlit (getLexeme lexbuf)) }
    | "\""['+' '-' '~']?['0'-'9']+'.'['0'-'9']+"\""
                              { REALLIT (mkReallit (getLexeme lexbuf)) }
    | "\""['a'-'z' 'A'-'Z' '0'-'9' '_' '\\' '/' '+' '*' '.']*"\""
                              { PATH (mkpath (getLexeme lexbuf)) }

    | eof                     { EOF }
    | _              { lexerError lexbuf "Illegal symbol in input" }

(* skip comments *)
and SkipComment = parse
        "-->"                 { commentDepth := !commentDepth - 1;
                                if !commentDepth = 0 then ()
                                else SkipComment lexbuf }
      | "<!--"                { commentDepth := !commentDepth + 1;
                                SkipComment lexbuf }
      | (eof | '\^Z')         { commentNotClosed lexbuf }
      | _                     { SkipComment lexbuf }

(* parsing a ML function *)
and Funktion = parse
        "</Pfkt>"             { "" }
      | (eof | '\^Z')         { commentNotClosed lexbuf }
      | _                     { (getLexeme lexbuf)^(Funktion lexbuf) }

(* parsing PRISM commands to add them at the end of a module *)
and Commands = parse
        "</Commands>"         { "" }
      | (eof | '\^Z')         { commentNotClosed lexbuf }
      | _                     { (getLexeme lexbuf)^(Code lexbuf) }

(* parsing PRISM code to add it behind the module declaration *)
and Code = parse
```

```
        "</Code>"                { "" }
      | (eof | '\^Z')            { commentNotClosed lexbuf }
      | _                        { (getLexeme lexbuf)^(Code lexbuf) }

(* parsing variables to be added to the variables declared in the AIF
    file*)
and Vars = parse
        ";"                      { (Vars lexbuf) }
      | "</Vars>"                { [] }
      | (eof | '\^Z')            { commentNotClosed lexbuf }
      | [^';' '<']*          { ((getLexeme lexbuf)^";")::(Vars lexbuf) }
;
```

The grammar defined in the ConfigParser.grm file based on the above file is:

```
LIT:
    NATLIT                     { $1 }
  | INTLIT                     { $1 }
PROBLIST:
                               { [] }
  | ITEM PROBLIST              { $1::$2 }
ITEM:
    bePE ex                    { (config.P_equal) }
  | FKT                        { (config.P_fkt($1)) }
  |     beI attPROB REALLIT attBEGIN LIT attEND LIT xx PROBLIST enI xx
                               { (config.ProbInterval($3, $5, $7, $9)) }
  |     beI attBEGIN LIT attEND LIT xx PROBLIST enI xx
                               { (config.Interval($3, $5, $7)) }
VAR:
    beINPUT attNAME ID ex      { (config.NoProb($3)) }
  | beINPUT attNAME ID xx PROBLIST enINPUT xx
                               { (config.Var($3, $5)) }
VARLIST:
                               { [] }
  | VAR VARLIST                { $1::$2 }


NEW_VARS:
                               { [] }
  |   VARIABLES                { $1 }
PRISMCODE:
                               { "" }
  |   CODE             { ("//Code added through the OPF file\n"^$1) }
PRISMCOMMANDS:
                               { "" }
  |   COMMANDS         { ("//Commands added through OPF file\n"^$1) }
PROBFILE:
    beAif attFILE PATH xx
        PSTART VARLIST PEND NEW_VARS PRISMCOMMANDS PRISMCODE
    enAif xx EOF
                               { config.ProbList($3,$6,$8,$9) }
```

```
;
```

5.4.3 Important Functions in Opal

To describe the whole code did not make sense. The important functions of Opal are described in this section.

Translating Variables

The code for translating local and output variables from AIF data to PRISM language is:

```
fun mkVariables ((inp,storage,typedec)::tl) =
    (case typedec of
        AIF.typeBool =>
            (VARS:=(LOCAL(inp,[(inp,storage,"bool")])))
            ::(!VARS);(inp,storage,"bool")
            ::(mkVariables tl))
        | AIF.typeBv(size) =>
            (VARS:=(LOCAL(inp,(mkBv inp size storage))))
            ::(!VARS);(mkBv inp size storage)
           @(mkVariables tl))
        | AIF.typeNat(size) =>
            if(size > 0)
                then    (VARS:=(LOCAL(inp,  [(
                            inp,
                            storage,
                            "[0 .. "^(Int.toString(size))^"]"
                                        )]))
                        ::(!VARS); (
                            inp,
                            storage,
                            "[0 .. "^(Int.toString(size))^"]"
                                   )
                        ::(mkVariables tl))
                else    raise Fail "No infinity types supported."
        | AIF.typeInt(size) =>
            let val size_str = Int.toString(size)
            in
            if(size > 0)
                then    (VARS:=(LOCAL(inp,  [(
                            inp,
                            storage,
                            "[-"^size_str^".."^size_str^"]"
                                        )]))
                        ::(!VARS);  (
                            inp,
                            storage,
                            "[-"^size_str^".."^size_str^"]"
                                   )
```

```
                        ::(mkVariables tl))
                else    raise Fail "No infinity types supported."
            end
        | AIF.typeArray(size, typedec) =>
                (VARS:=(LOCAL(inp,(mkArray inp size typedec storage)))
                ::(!VARS);
                (mkArray inp size typedec storage)@(mkVariables tl))
        | AIF.typeTuple(types) =>
                (VARS:=(LOCAL(inp,(mkTupel inp types storage)))
                ::(!VARS);
                (mkTupel inp types storage)@(mkVariables tl)))

 |  mkVariables _ = nil

and mkBv inp 0 _ = nil
 |  mkBv inp size storage =
                (inp^"_bv_"^(Int.toString size), storage, "bool")
                ::(mkBv inp (size-1) storage)

and mkArray name 0 _ _ = nil
 |  mkArray name size typedec storage=
                (mkVariables ([(
                    name^"_arr_"^(Int.toString size),
                    storage,
                    typedec  )]))
                @(mkArray name (size-1) typedec storage)

and mkTupel name [] _ = nil
 |  mkTupel name (typedec::tl) storage =
                (mkVariables ([(
                    name^"_tup_"^(Int.toString
                        (List.length(typedec::tl))),
                    storage,
                    typedec  )]))
                @(mkTupel name tl storage)
```

If a infinity data type is read a exception is raised, because PRISM does not support them. The code for input variables are analog but the field storage is omitted.

Translating Guarded Actions

Guarded actions are translated into PRISM commands to generate a PRISM file. The next code listings will exemplify the procedure to translate the already sorted guarded actions. The translation is made in two steps. There are actions from a unsupported types hence they have to translated into supported data types similar to the translation of variables. Opal resolve this conflict by splitting unsupported types into basic types, which can be translated easily.

Step One

The first step translates the actions into a list. This is shown for Boolean and bit vectors.

```
fun printBlExpr AIF.blTrue = ["true"]
 | printBlExpr AIF.blFalse = ["false"]
 | printBlExpr (AIF.blVal(LocE)) = printLocExpr(LocE)
 | printBlExpr (AIF.bvPos(BvE, pos)) = [List.nth(printBvExpr(BvE),pos)]
 | printBlExpr (AIF.bvEQ(BvE1, BvE2)) =
   [bvtobool
       (writeBV
           (printBvExpr(BvE1))
           (printBvExpr(BvE2))
           " (" " = " ") "
        )
    ]
 | printBlExpr (AIF.bvNE(BvE1, BvE2)) =
   [bvtobool
       (writeBV
           (printBvExpr(BvE1))
           (printBvExpr(BvE2))
           " (" " != " ") "
        )
    ]
 | printBlExpr (AIF.natEQ(NatE1, NatE2)) =
   ["("^(hd(printNatExpr(NatE1)))^" = "^(hd(printNatExpr(NatE2)))^")"]
 | printBlExpr (AIF.natNE(NatE1, NatE2)) =
   ["("^(hd(printNatExpr(NatE1)))^" != "^(hd(printNatExpr(NatE2)))^")"]
 | printBlExpr (AIF.natLT(NatE1, NatE2)) =
   ["("^(hd(printNatExpr(NatE1)))^" < "^(hd(printNatExpr(NatE2)))^")"]
 | printBlExpr (AIF.natLE(NatE1, NatE2)) =
   ["("^(hd(printNatExpr(NatE1)))^" <= "^(hd(printNatExpr(NatE2)))^")"]
 | printBlExpr (AIF.intEQ(IntE1, IntE2)) =
   ["("^(hd(printIntExpr(IntE1)))^" = "^(hd(printIntExpr(IntE2)))^")"]
 | printBlExpr (AIF.intNE(IntE1, IntE2)) =
   ["("^(hd(printIntExpr(IntE1)))^" != "^(hd(printIntExpr(IntE2)))^")"]
 | printBlExpr (AIF.intLT(IntE1, IntE2)) =
   ["("^(hd(printIntExpr(IntE1)))^" < "^(hd(printIntExpr(IntE2)))^")"]
 | printBlExpr (AIF.intLE(IntE1, IntE2)) =
   ["("^(hd(printIntExpr(IntE1)))^" <= "^(hd(printIntExpr(IntE2)))^")"]
 | printBlExpr (AIF.blNot(BlE)) =
   ["!("^(hd(printBlExpr(BlE)))^")"]
 | printBlExpr (AIF.blAnd(BlE1, BlE2)) =
   ["("^(hd(printBlExpr(BlE1)))^" & "^(hd(printBlExpr(BlE2)))^")"]
 | printBlExpr (AIF.blOr(BlE1, BlE2)) =
   ["("^(hd(printBlExpr(BlE1)))^" | "^(hd(printBlExpr(BlE2)))^")"]
 | printBlExpr (AIF.blImp(BlE1, BlE2)) =
   ["("^(hd(printBlExpr(BlE1)))^" -> "^(hd(printBlExpr(BlE2)))^")"]
 | printBlExpr (AIF.blXor(BlE1, BlE2)) =
   ["("^(hd(printBlExpr(BlE1)))^" xor "^(hd(printBlExpr(BlE2)))^")"]
```

```
  | printBlExpr (AIF.blXnor(BlE1, BlE2)) =
    ["("^(hd(printBlExpr(BlE1)))^" xnor "^(hd(printBlExpr(BlE2)))^")"]
  | printBlExpr (AIF.blIte(BlEB, BlE1, BlE2)) =
    ["(("^(hd(printBlExpr(BlEB)))^")? "^
        (hd(printBlExpr(BlE1)))^" : "^
        (hd(printBlExpr(BlE2)))^")"]
```

Boolean translated into a list containing one element. This is analog for translation of integer and natural numbers. Bit vectors translated in a different way.

```
fun printBvExpr (AIF.bvConst(boolelem::tl)) =
    (Bool.toString(boolelem))::(printBvExpr (AIF.bvConst(tl)))
  | printBvExpr (AIF.bvConst([])) = []
  | printBvExpr (AIF.bvVal(LocE)) = printLocExpr(LocE)
  | printBvExpr (AIF.blRepl(BlE, 0)) = []
  | printBvExpr (AIF.blRepl(BlE, pos)) =
    (hd(printBlExpr(BlE)))::(printBvExpr (AIF.blRepl(BlE, pos-1)))
  | printBvExpr (AIF.bvNot(BvE)) = mkNOT (printBvExpr(BvE))
  | printBvExpr (AIF.bvAnd(BvE1, BvE2)) =
    writeBV (printBvExpr(BvE1)) (printBvExpr(BvE2)) "(" " & " ")"
  | printBvExpr (AIF.bvOr(BvE1, BvE2)) =
    writeBV (printBvExpr(BvE1)) (printBvExpr(BvE2)) "(" " | " ")"
  | printBvExpr (AIF.bvImp(BvE1, BvE2)) =
    writeBV (printBvExpr(BvE1)) (printBvExpr(BvE2)) "(" " -> " ")"
  | printBvExpr (AIF.bvXor(BvE1, BvE2)) =
    writeBV (printBvExpr(BvE1)) (printBvExpr(BvE2)) "(" " xor " ")"
  | printBvExpr (AIF.bvXnor(BvE1, BvE2)) =
    writeBV (printBvExpr(BvE1)) (printBvExpr(BvE2)) "(" " xnor " ")"
  | printBvExpr (AIF.bvIte(BvE, BvE1, BvE2)) =
    writeBVITE  (printBvExpr(BvE))
                (printBvExpr(BvE1))
                (printBvExpr(BvE2))
  | printBvExpr (AIF.bvConcat(BvE1, BvE2)) =
    ((printBvExpr(BvE1)))@((printBvExpr(BvE2)))
  | printBvExpr (AIF.bvReverse(BvE)) =
    List.rev((printBvExpr(BvE)))
  | printBvExpr (AIF.bvSegment(BvE, seg_start, seg_end)) =
    List.take(
        List.drop((printBvExpr(BvE)),seg_start),(seg_end-seg_start)
        )
and mkNOT (item::tl) = "!("^item^")"::(mkNOT tl)
  | mkNOT [] = []
and writeBV (bv1::tl1) (bv2::tl2) pre inf suf =
    (pre^bv1^inf^bv2^suf)::(writeBV tl1 tl2 pre inf suf)
  | writeBV _ _ _ _ _ = []
and writeBVITE (bvb::tlb) (bv1::tl1) (bv2::tl2) =
    ("("^bvb^"?"^bv1^":"^bv2^")")::writeBVITE tlb tl1 tl2
  | writeBVITE _ _ _ = []
```

Each bit vector is disjointed into Boolean. Each list item represents a Boolean of the bit vector. the same procedure is done for tuple and arrays.

Step Two

In step two the target variable of the guarded action is translated into a list like described above. Then the following functions is applied with the result of step one and the translated variable list as parameters.

```
fun insertParam (item::tl1) (param::tl2) =
    (param^"'='^item)::(insertParam tl1 tl2)
 | insertParam _ _ = nil
```

The types of **item** and **param** are the same. Afterwards each list item have to be packed into one $update_i$ block described in section 4.2. Step one translates the guard of the guarded action, which is used to generates the *guard*, as well.

Determining the Probabilities

The probabilities, which are described in the OPF file are calculated by many recursive functions. Here two cases of such a function will be described. The used **printProbList** function composes the given parameters to a string.

```
 | getProbString   (Var(name, ((config.P_equal)::tl)::tl2))
           (var_id::tlv)
           ("bool"::tlt)
           (maxItems::tli)
           ((levelItems,max_prob)::tll)
           (Prob::tlp)
           mode =
   let
       val prob = Real.toString(max_prob/levelItems)
       val item = printProbList
                  var_id
                  ("true"::["false"])
                  (prob::[prob])
   in
       item::
       (getProbString (Var(name, (tl2)))
              tlv tlt tli tll tlp mode)
   end
```

The parameters of the function are the probability specification list described in the OPF file, the list of translated variable names, the list of types, the maximum number of all values in the range of the variable, which can be different for tuple. The variable **max_prob** is the probability, which have to be distributed in the *Pequal* mode. The variable **levelItems** contains the number of items, which the *Pequal* mode have to assign. The already assigned probability is saved in the variable **Prob**. The last variable **mode** is true if the function call is in *Pequal* mode.

```
 | getProbString   (Var(name, (config.P_fkt(fkt)::tl)::tl2))
           (var_id::tlv)
           (range_str::tlt)
           (maxItems::tli)
```

```
            ((levelItems,max_prob)::tll)
            (maxProb::tlp)
            false =

    let
        val begin = extractIntervalBegin range_str
            val endI = extranctIntervalEnd range_str
            val value_list = (mkValueList begin (endI))
        val _ = compile fkt
        val prob_list = (map
            (fn param => (Real.toString)
                (call   (param::
                    (Real.toString(maxProb))::
                    [(Int.toString(List.length(value_list)))]
                )))
            value_list)
        val item_prefix = printProbList var_id value_list prob_list
    in
        item_prefix::
        (getProbString  (Var(name, (tl2)))
                tlv tlt tli tll tlp false)
    end
```

In this function all parameters for the call of the ML function are computed. Then the call of this function for each value is done. All the not mentioned functions have a simple name, which corresponds to the operation of the function. All other cases not mentioned here have programmed in a similar way.

Chapter 6

Summary

In this work one way to allow probabilistic model checking for synchronous languages is presented. A single synchronous language was exemplary used. It was explained in which format an arbitrary synchronous language is needed. The required methods to transform the language in this format is mentioned. To understand the principles of probabilistic model checking basic definitions were made. The usage of the tool PRISM was also introduced such that simple examples are understandable. The, for the usage of Opal, needed OPF format was introduced. At the end the main part of this work the compiler Opal was established. The usage of Opal was introduced and explained with the help of an example.

6.1 Further Work

This work is a first approach to allow probabilistic model checking for synchronous programs but there is still space for improvement. The need for improvement will maybe appear after a consequent usage of Opal. It also makes sense to include often used functions described as ML code in the OPF file as macros. A major improvement for probabilistic model checking, PRISM and Opal will be the handling of infinite data types but there are no solutions yet. If improvments are made, Opal have to be extended to allow the usage of infinite data types. A practical improvement will be the exclusion of the serialisation of the AIF because this approach is used in the Averest framework too. Then Opal will have to be changed to allow a serialized AIF maybe called sAIF as input. In this case the serialization can be optimized independent of Opal and other tools. The transformation of guarded actions, which is described in section 5.2, has to change into:

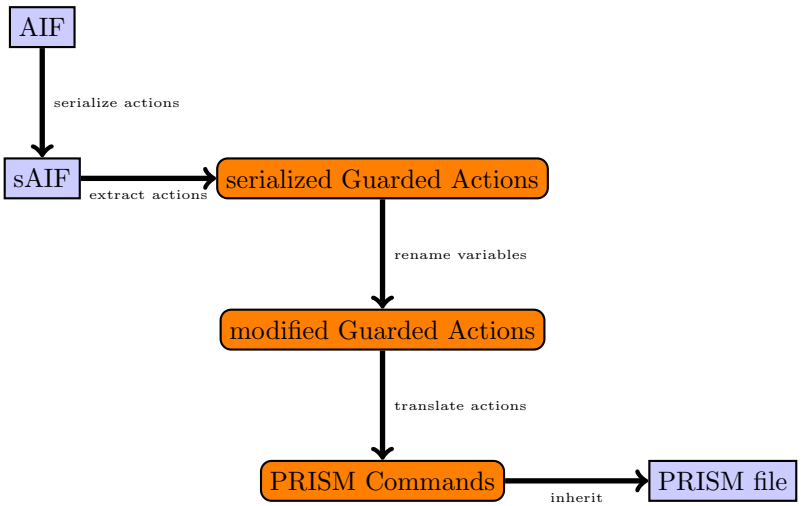

Figure 6.1: Transformation of serialized Guarded Actions

Bibliography

[1] Nina Amla, E. Allen Emerson, Robert P. Kurshan, and Kedar S. Namjoshi. Model checking synchronous timing diagrams. In Formal Methods in Computer-Aided Design, pages 283–298, 2000.

[2] Properties Budde And. A generator of boolean acceptors for safety. citeseer.ist.psu.edu/514920.html.

[3] Averest website. http://www.Averest.org/.

[4] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In R. Alur and T. Henzinger, editors, Proc. 8th International Conference on Computer Aided Verification (CAV'96), volume 1102 of LNCS, pages 269–276. Springer, 1996.

[5] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time markov chains. In Proc. 10th International Conference on Concurrency Theory (CONCUR'99), volume 1664 of Lecture Notes in Computer Science, pages 146–161, 1999.

[6] Albert Benveniste, Paul Le Guernic, Yves Sorel, and Michel Sorine. A denotational theory of synchronous reactive systems. Inf. Comput., 99(2):192–230, 1992.

[7] G. Berry. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org/, July 1999.

[8] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. Thiagarajan, editor, Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science, volume 1026 of LNCS, pages 499–513. Springer, 1995.

[9] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In M. Zelkowitz, editor, Advances in Computers, volume 58, pages 118–149. Academic Press, 2003.

[10] Blast website. http://mtc.epfl.ch/software-tools/blast/.

[11] F. Boussinot. SugarCubes implementation of causality. Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis Cedex (France), September 1998.

[12] Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. ACM Trans. Program. Lang. Syst., 21(4):747–789, 1999.

[13] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. Information and Computation, 98(2):142–170, June 1992.

[14] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 343–354, New York, NY, USA, 1992. ACM.

[15] C. Courcoubetis and S. Tripakis. Probabilistic model checking: formalisms and algorithms for discrete and real-time systems, 2000.

[16] Dcvalid website. http://www.tcs.tifr.res.in/∼pandya/dcvalid.html.

[17] Udo Kamps Erhard Cramer. Grundlagen der Wahrscheinlichkeitsrechnung und Statistik. Springer-Verlag, 2007.

[18] Esterel website. http://www.esterel-technologies.com/.

[19] etmc^2 website. http://www7.informatik.uni-erlangen.de/etmcc/.

[20] M. Habib. Probabilistic methods for algorithmic discrete mathematics. Springer-Verlag, 1998.

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

[22] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. Formal Aspects of Computing, 6(5):512–535, 1994.

[23] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), volume 3920 of LNCS, pages 441–444. Springer, 2006.

[24] G.J. Holzmann. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, 2004.

[25] IEEE Computer Society. IEEE Standard VHDL Language Reference Manual. New York, USA, 2000. IEEE Std. 1076-2000.

[26] Kronous website. http://www-verimag.imag.fr/TEMPORISE/kronos/.

[27] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07), volume 4486 of LNCS (Tutorial Volume), pages 220–270. Springer, 2007.

[28] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 2.0: A tool for probabilistic model checking.

[29] Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic model checking in practice: case studies with prism. SIGMETRICS Perform. Eval. Rev., 32(4):16–21, 2005.

[30] G. Logothetis, K. Schneider, and C. Metzler. Runtime analysis of synchronous programs for low-level real-time verification. In Symposium on Integrated Circuits and System Design (SBCCI), SÃ£o Paulo, Brazil, 2003. IEEE Computer Society.

[31] S. Malik. Analysis of cyclic combinational circuits. In International Conference on Computer Aided Design (ICCAD), pages 618–625, Santa Clara, CA, USA, 1993. IEEE Computer Society.

[32] A. Merceron. Checking synchronous programs using automatic abstraction, 1996.

[33] A. Merceron, M. Morley, and A. Poigné. Checking synchronous programs via boolean automata. citeseer.ist.psu.edu/article/merceron95checking.html.

[34] Moscow ml website. http://www.itu.dk/people/sestoft/mosml.html.

[35] Mrmc website. http://wwwhome.cs.utwente.nl/∼zapreevis/mrmc/.

[36] H.A. Oldenkamp. Probabilistic model checking - a comparison of tools. http://wwwhome.cs.utwente.nl/∼oldenkampha/.

[37] W. Jahn P. Langrock. Einfuehrung in die Theorie der Markovschen Ketten und ihre Anwendungen. Teubner, 1979.

[38] D. Parker. Implementation of symbolic model checking for probabilistic system, 2001.

[39] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems, P. Panangaden and F. van Breugel (eds.), volume 23 of CRM Monograph Series. American Mathematical Society, 2004.

[40] K. Schneider. Averest interchange format. http://www.averest.org/documentation/aif.pdf, 2006.

[41] K. Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, 2008.

[42] T. Schuele. Verification of Infinite State Systems Using Presburger Arithmetic. PhD thesis, University of Kaiserslautern, Germany, 2007.

[43] Smv website. http://www.cs.cmu.edu/∼modelcheck/smv.html.

[44] Spin website. http://spinroot.com/spin/whatispin.html.

[45] Statemate website. http://modeling.telelogic.com/products/statemate/index.cfm/.

[46] William J. Stewart. Introduction to the Numerical Solution of Markov Chains. Princeton University Press, 1994.

[47] Synalp website. http://www.inrialpes.fr/synalp/.

[48] Vesta website. http://osl.cs.uiuc.edu/∼ksen/vesta2/.

[49] I. Wegener. BDDs - design, analysis, complexity, and applications. Discrete Applied Mathematics, 138(1-2):229–251, 2004.

[50] Ymer website. http://www.cs.cmu.edu/∼lorens/.