ON THE FORMAL SEMANTICS OF BLUESPEC SYSTEM VERILOG

June 2013
Oddur Óskar Kjartansson
Master of Science in Computer Science
ON THE FORMAL SEMANTICS OF BLUESPEC SYSTEM VERILOG

Oddur Óskar Kjartansson
Master of Science
Computer Science
June 2013
School of Computer Science
Reykjavík University

M.Sc. RESEARCH THESIS
On the Formal Semantics of Bluespec System Verilog

by

Oddur Óskar Kjartansson

Research thesis submitted to the School of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

June 2013

Research Thesis Committee:

Anna Ingólfsdóttir, Supervisor
Professor, Reykjavik University

Hans Hüttel
Associate Professor, Aalborg University

Marjan Sirjani
Associate Professor, Reykjavik University
The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this research thesis entitled On the Formal Semantics of Bluespec System Verilog submitted by Oddur Óskar Kjartansson in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

__________________________________________
Date

__________________________________________
Anna Ingólfsdóttir, Supervisor
Professor, Reykjavik University

__________________________________________
Hans Hüttel
Associate Professor, Aalborg University

__________________________________________
Marjan Sirjani
Associate Professor, Reykjavik University
The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this research thesis entitled **On the Formal Semantics of Bluespec System Verilog** and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the research thesis, and except as herein before provided, neither the research thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author’s prior written permission.

______________________________
Date

______________________________
Oddur Óskar Kjartansson
Master of Science
On the Formal Semantics of Bluespec System Verilog

Oddur Óskar Kjartansson

June 2013

Abstract

Bluspec System Verilog (BSV) is a high-level hardware description language. The language is used for hardware modelling and synthesis of hardware. In BSV hardware is viewed as a collection of statefull elements and the behaviour of the system is described using guarded atomic actions called rules, which contain statements that can alter the state in clock-cycles.

In this project we introduce a language which is a sub-set of BSV, called $\mu$BSV. This language reflects the basic behaviour of BSV. We provide formal semantics for $\mu$BSV using natural semantics.
Um formlega merkingarfræði Bluespec System Verilog

Oddur Óskar Kjartansson
Júní 2013

Útdráttur

Bluspec System Verilog (BSV) er skilgreiningamál ætlað fyrir hönnun á smárásum. Það er notað við hönnun og smíði á smárásum. Í BSV lítum við á rásirnar sem safn af einingum sem geyma upplýsingar og hegðun kerfanna er skilgreind með einföldum aðgerðum sem kallaðar eru reglur. Reglurnar geta haft áhrif á upplýsingarnar sem rásirnar geyma í hverjum klukkuslætti.

Í þessu verkefni kynnum við mál sem er hlut mengi málsins BSV, sem við köllum $\mu$BSV. Því er ætlað að endurspeglar grunnhegðun BSV. Við skilgreinum formlega merkingarfræði fyrir $\mu$BSV.
Acknowledgements

I want to thank my supervisor, Anna Ingólfsdóttir, her help went far beyond what can be expected of a faculty member advising on a project. The faculty and staff of the Computer Science department at Reykjavík University, who are truly an amazing group of people. My family, they have been incredibly understanding supportive. My friend Björn Jónsson, who is always willing to lend a helping hand, or mind, be that discussing the work or world domination. Hjalti Magnússon, my on call mathematician, who is always willing to answer my silly questions.

Lastly I would like to thank ICE-TCS for support on this project.
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Introduction .................................................. 1

1.2 Problem Statement ................................. 3

1.3 Overview of the Thesis ..................... 3

## Part I Background

2 Background .................................................. 5

2.1 Program Execution ......................................... 7

2.1.1 CPU execution of programs .................... 7

2.1.2 Hardware Execution of Programs ................. 8

2.2 Hardware Design ........................................ 9

2.3 Hardware Description Languages ............... 10

2.3.1 Bluespec System Verilog ....................... 11

2.3.2 Informal Description of the Language BSV .... 12

2.3.3 Language Constructs of BSV ..................... 12

2.4 Informal Description of BSV Program Execution 13

2.4.1 Scheduler ............................................. 14

2.4.2 Describing the Modules ......................... 14

2.5 Examples of BSV Programs ....................... 15

2.5.1 Hello World .......................................... 15

2.5.2 Examples Reflecting Module Hierarchy .......... 16

2.6 Formal Semantics ..................................... 21

2.6.1 Different Types of Semantics .................... 22
List of Figures

2.1 A simple transistor diagram. ................................. 10
2.2 Module definition hierarchy reflection example 2.3 ............ 20
2.3 Module instanciation tree representing example 2.3 ........... 20
2.4 Rules and which variables they read from ...................... 20
2.5 Scheduler for the program in example 2.3 ...................... 21
3.1 Relations in module definitions. ............................... 34
3.2 Big step operational semantics for BSV ....................... 39
3.3 The execution of the first clock cycle in example 3.2.7 ....... 41
3.4 The execution of the second clock cycle in example 3.2.7 .... 41
3.5 The execution of the second clock cycle in example 3.2.8 .... 43
A.1 Example of a X-labelled trees, where $X = \{t_0, t_1, t_2, t_3\}$ .... 54
B.1 NAND gate and an NOR gate. ................................. 58
List of Tables

2.1 A very simple program in BSV .............................................. 15
2.2 Example of a module hierarchy ............................................. 17
2.3 Concrete syntax example reflecting module hierarchy ................ 19
2.4 Natural semantics rules the simplified language ....................... 24
B.1 Truth table for NAND gate ................................................ 57
C.1 Algorithm implimented in BSV ........................................... 59
List of Algorithms

1. FullList ......................................................... 62
2. SingleMod ......................................................... 62
3. ModuList ......................................................... 62
4. ModuleMethods .................................................. 63
5. SingModMethods ................................................. 63
6. MethodList ....................................................... 63
Nomenclature

\( \text{AExp}(X, N) \)  The set of arithmetic expressions, page 33
\( \text{AMethNames} \)  The set of action method names , page 32
\( \text{AMethods}(X, Y, N, L) \)  The set of action methods , page 41
\( \text{ameths}_M \)  The action methods of \( M \), page 41
\( \mathbb{B} \)  The set of boolean , page 32
\( \text{BExp}(X, N) \)  The set of arithmetic expressions, page 33
\( \text{MethValVars} \)  The set of method value variables , page 32
\( \text{ModDef} \)  The set of module definitions , page 33
\( \text{ModNames} \)  The set of module names , page 32
\( \text{mods}_M \)  The submodules of \( M \), page 41
\( \text{ModVars} \)  The set of module variables , page 32
\( \mu \text{BSV} \)  The subset of BSV described , page 31
\( \mathbb{N} \)  The set of numerals , page 32
\( \text{RuleNames} \)  The set of rule names , page 32
\( \text{Rules}(X, L, N) \)  The set of rules, page 41
\( \text{rules}_M \)  The rules of \( M \), page 41
\( \text{Stm}(X, Z, L, N) \)  The set of arithmetic expressions, page 33
\( \text{sto}_M \)  The store of module \( M \), page 41
\( \text{ValVar} \)  The set of value variables , page 32
\( \text{VMethNames} \)  The set of value method value names , page 32
**VMethods**($Y$) The set of value methods, page 41

**vmeths**$_M$ The value methods of $M$, page 41

**DAMeth** Action method declaration, page 33

**DModDef** Module declaration, page 33

**DModVar** Module variable declaration, page 33

**DRule** Rule declaration, page 33

**DValVar** Value variable declaration, page 33

**DVMethod** Value method declaration, page 33

**BSV** Bluespec System Verilog, page 15

**CPU** Central Processing Unit, page 14

**DAG** Directed Acyclic Graph, page 17

**FPGA** Field Programmable Gate Arrays, page 15

**HDL** Hardware Description Language, page 14

**IC** Integrated Circuit, page 13

**RTL** Register Transfer Level, page 15
Chapter 1

Introduction

1.1 Introduction

When we deal with programs which are executed by a single processor, we think of instructions which are executed one after another by the processor. Most general purpose processors use this model, and the programming languages we use are essentially converted into a sequence of instructions which the processor performs. In other words, the syntax of the programming language is converted into a sequence of instructions for the processor, using a compiler. This can be viewed as a high-level abstraction of the informal semantics of programming languages.

When dealing with hardware, we have a different view of how actions are performed. Hardware is a collection of transistor elements, which are connected so that they can perform logical actions, based on the input to the hardware. The input to the hardware is voltage, either high of low indicating either a 0 or a 1 bit. The logic is performed using semiconducting devices. The hardware is driven by a clock, which is dictates when actions have been completed and the next action can be performed. The hardware can be combined to form memory, and the output of the hardware is a combination of bits in the memory and the input. The hardware can therefore be viewed as a combination of state-full elements, called registers, and combinational logical devices which describe how the states are updated. The updates of the registers can be designed to be performed simultaneously within a clock cycle. Hardware is often designed to perform highly parallel operations.

The language Bluespec System Verilog (BSV) is a high-level hardware description language. BSV has two types of compiles, one which turns code into programs which can
be simulated on a computer, and another which produces hardware. The aims of the language is to abstract away from the hardware such that the programmer does not need to think about how the underlying hardware is generated. This is based on the same idea as for other high-level programming languages, where the programmer does not work with instructions executed on the processor, rather just the intended meaning of the program. Higher-level abstraction facilitates modular designs which can easily be plugged together to form complex systems. However, some of the underlying features of hardware constructs are present in the language such as modules and clock cycles.

Why do we want to describe the formal semantics of a programming language? The main reason is that we want to be able to reason about how the program works. We want our system to behave correctly. What correctly means is up for debate. In general we want the systems we use to conform with what they were designed to do. A similarity in the physical world would be bridge design, we want the bridge to withstand some given load, earthquakes, and not to have unwanted behaviour such as uncontrolled oscillation. Engineers have sophisticated tools to aid in design and to verify that the designs meet their requirements.

In a similar way we want systems to deliver the results they are designed to do, without unwanted behaviour.

Building software systems is in general a complicated task. The problems which we encounter are among others related to concurrency in systems. In concurrent systems multiple processes are running simultaneously and what we are faced with are issues such as deadlocking and resource starvation. These problems have proven to be hard to solve and become increasingly harder as the systems grow in size. We want to be able to make informed decisions, and reason about the behaviour of the system.

As the devices we use, the hardware, becomes increasingly mobile, other interesting questions arise. If we assume that we have two candidate designs, one which consumes $10\,mW$ of electricity and another designs that consumes $20\,mW$, but otherwise have the same functionality - how can we validate that both designs are equivalent?

Formal semantics describe, using mathematical models, how a program behaves when it is executed. We can use mathematical proof systems to prove various properties of programming language, based on the models. A formal semantic model of a language is free of ambiguities, and therefore is useful when implementing the compiler for the language.

There are different types systems used to describe formal semantics, they can be divided in to three main categories: Axiomatic semantics where the meaning of a program is de-
derived from a set of axioms, denotational semantics, where each phrase of the language are described using denotations and operational semantics where the execution of the language is described. Operational semantics can be divided further into structural operational semantics and natural semantics. In structural operational semantics, or small step semantics, we are interested in how each construct of the language affects the state of the program. In natural semantics, or big step semantics, we are interested in the overall affect of the program to the state of the program. In this thesis we are using natural semantics, based on clock cycles, to describe the behaviour of the language BSV.

Additionally to the benefits we have mentioned, it is also possible to use the formal semantics as input to a model checker. Using a model checker, we can verify that the software we are designing does not have certain unwanted properties.

With current technologies, building hardware components is becoming similar to building software. The designs can be converted directly into hardware descriptions that can then be converted into hardware. What we want to be able to do, before we create the actual hardware, is to be able to reason formally about how the hardware described in a hardware description language will behave.

1.2 Problem Statement

Formal semantics for the language BSV describing the runtime behaviour is needed. With formal semantics we can reason formally about the behaviour of the system which the language describes. Using formal methods it is possible to build tools which can check if the systems have certain properties. We can furthermore verify, given a formal specification, that the system we are designing is in accordance to the specification. The problem we are working on is to give a formal semantic for the hardware description language BSV. The formal description can then be used to model and verify the hardware created in BSV.

1.3 Overview of the Thesis

The thesis is organized as follows. In chapter 2 we give background to the work conducted in the project, we start by giving a short overview on program execution on a personal computer and using hardware for execution. We introduce formal semantics and describe the notation used. We give an informal description of the language Bluspec Sys-
tem Verilog and end the chapter by giving some concrete examples of code written in BSV. In chapter 3 we define the language $\mu$BSV which is a subset of BSV. We start by defining the abstract syntax of the language and follow on to define the formal semantics of the language using natural semantics. In chapter 4 we conclude our work, discuss related and future work. Some further examples, useful notation and a simple description on implementing logic with transistors can be found in the appendix.
Part I

Background
Chapter 2

Background

In this chapter we will provide a background on the topics of the thesis. In order to gain insight into the problems we are addressing, we need to give a brief introduction to program execution, hardware design, the language Bluespec System Verilog (BSV) and formal semantics.

2.1 Program Execution

In order to contrast the difference between how we evaluate programs in a programming language which is run on a general purpose processing unit or a Central Processor Unit (CPU), we will give a high level description of such mechanisms.

2.1.1 CPU execution of programs

CPUs are the hardware components responsible for performing the work in general purpose computing devices, such as a personal computer. The CPU is responsible for the calculations performed on computer

The CPU only understands instructions on binary form called machine code. The instructions perform actions on the smallest level of granularity, such as reading from a register, take the value from one of the registers and perform arithmetic actions. If we were to program on that level, by define each instruction the processor should perform, programming would be complicated and the programmer would very quickly lose overview of his work.
Higher-level programming languages were invented in order to abstract away from these tiny instructions, or essentially abstract away from the underlying hardware. Higher-level programming languages rely on either compilers or interpreters which take the syntax written in the language and converts machine code.

The trend has been to create a higher level of abstraction in programming languages. That means that the user, the programmer, does not need to think about what goes on when he accesses a file on the hard drive or how he gets data from a web server on the other side of the planet. The programmer relies on smaller components or modules which are responsible for such tasks. This idea of modularity, where an isolated function has been defined and the module has a defined interface, makes programming a much easier task. The programmer references the module, and has simple methods to interact with the modules. The ideas of higher-level programming and modularity creates a framework for a programmer, where he does not need to know the details on how the underlying hardware works.

2.1.2 Hardware Execution of Programs

What we mean by hardware execution of a program is how the hardware complied from the higher-level description behaves. The program is converted into hardware, and we are describing how the logic of the hardware performs. We still use the notion of a program, as we feel that the concept encapsulates the meaning adequately.

The execution of programs in hardware obeys different laws than those described for execution on a CPU. We view the hardware on the register transfer level (RTL). On the RTL, we have state-full elements called registers and combinational logic devices which perform single bit binary instructions such as AND, OR and NOT. These devices are comprised of transistors which perform the desired logic, based on the values of the registers.

In RTL we are working on the bit level, and are interested in how the bits flow between elements in the circuit [6]. We have a clock which discretizes the actions performed by the device. The clock is responsible for advancing the time in the system. It is beneficial to perform as many bit operations in each clock cycle as possible. The limitations we have are essentially those that we can only store a single value in each register at a time, that is only one combinational element can set the value of a register at the time. In our notion of time in these circuits, we assume that the actions are instantaneous, we are therefore not interested in the time it takes for the circuit to reach a steady state.
The combinational circuits dictate how the values of the registers are updated. In each clock cycle, the state is changed based on how the previous state, the input, and how the combinational elements are connected.

The execution model which we are describing for hardware is considerably different than the model traditional CPU model. The aim is to create circuits which perform specialized tasks using highly parallel calculations, and to describe how the execution of those circuits will be performed.

2.2 Hardware Design

In this project, the hardware design we are focusing on is what generally is called integrated circuit (IC) design. In other words, we are not talking about the supporting hardware such as capacitors, inductors, resistors etc., which lay on the boards outside of the IC.

ICs can be found almost everywhere. Everything which contains a computer contains a IC in some form. ICs can be anything from simple circuits which are used for Binary Coded Decimal displays up to complex general purpose processors.

The first IC emerged in the late 1940s. Initially ICs were designed “by hand”. As ICs became more complicated it became evident that some other method was needed to design them. In order to tackle this problem hardware description languages (HDL) were introduced. The first HDL was the ISPS Computer Description Language [4].

Hardware description languages are programming languages, where instead of compiling the code into binary instructions where a central processing unit (CPU) handles the instructions, the code is converted into hardware.

ICs contain a large collection of transistors. Transistors are semiconductor devices which contain three ports. Essentially one of the ports controls how the flow of current between two other gates. Transistors come in various forms, where the physical parameters vary.

Figure 2.1 shows us a simple diagram of a transistor. The port marked as $A$ controls the flow of current from the voltage source $V_{CC}$ through the transistor to the ground. So if we supply high enough voltage to the port $A$, usually referred to seeing the port to high, current will flow. On the other hand, if we set the voltage of $A$ below a certain threshold no current will flow.
2.3 Hardware Description Languages

In electronics, a hardware description language (HDL) is used in the process of designing and implementing hardware. It describes how circuits operate and aids in the design process by means of testing and verifying various aspects of the hardware using simulations.

There are many hardware description languages available. The languages differ both in approaches and the level of abstraction they use. The trend has been to create higher-level programming languages for this purpose. The following levels can be used to describe the varying complexity of hardware design.

1. **Transistor level**: Synthesis of the circuit. This is usually done using software which converts the Register Transfer Level (RTL) descriptions directly into hardware (tapeout).

2. **Register transfer level**: (RTL) This is usually created using software which converts higher level descriptions such as Verilog to RTL circuits. Simulation and verification is usually done at this level of abstraction.

3. **Higher level description languages**: This is the highest level of abstraction. Hardware is described syntactically and “compilers” are then used to convert the design into RTL or any other intermediate format before the design is converted into hardware.

When a hardware description, be that from a higher level language or a lower level language, is realized, the process is called *tape-out*. Tape-out is the final state of the design.
process of hardware and is the form which is sent to a factory. The factory then converts the tape-out to ICs.

2.3.1 Bluespec System Verilog

*Bluespec System verilog (BSV) is a high-level hardware description language used for architectural exploration, hardware modelling and synthesis of semiconductor chips.*

In general, BSV is indented to be able to create highly parallel hardware. High parallelism is a huge benefit in hardware, since it can produce faster hardware and/or reduce the power consumption.

The main benefits of BSV are the following:

1. It is easy to reuse code.
2. The code can be transformed directly into circuits.
3. It is claimed to have verifiable properties.

BSV is described in [10] as follows:

In BSV, one views hardware as a collection of stateful elements (e.g., registers, memories) and describes its behavior using rules, or Guarded Atomic Actions which modify these elements. All legal behaviors of a Bluespec program can be explained in terms of rules being applied in some sequence. Scheduling is the process of selecting which rules to execute in parallel while maintaining this semantic invariant. The scheduling decision can have a large impact on critical design properties such as pipeline concurrency and clock frequency. What constitutes a good schedule often depends upon the application and requires the designer’s input.

The aim of BSV is to create a higher-level approach to hardware design, while still being able to maintain a high level of flexibility in the design. The language is described to be a high-level synthesizable specification language, which means that specifications can be directly converted, by some means, into hardware.

The semantics we define excludes packages, has only one type of variable declaration and interfaces are assumed. That means, that in stead of explicitly defining interfaces, we assume that all methods can be used by other modules. The language contains other constructs, such as for looping which can be used to repeatedly define instances of modules, as these actions can be viewed as macros for building larger systems, and don’t affect
how modules are executed, they were omitted. The constructs we include are sufficient to describe clock cycles.

As these construct are not converted to hardware, they are not part of the semantics.

2.3.2 Informal Description of the Language BSV

The language BSV is an object oriented high level hardware description language. The objects, called modules, are the central constructs of the language and will be compiled into hardware components.

Modules are defined using module definitions that correspond to class description in object oriented languages like, for instance, Java, and give an abstract description or a blueprint for the modules. The concrete modules are then obtained as instances of this general description.

Module definitions do not support inheritance or other similar abstractions mechanism, although instances can be initialized with parameters when they are created. Modules form a hierarchy by aggregation as a module description can involve instances of other module descriptions. This reflects the modular structure of the hardware which the modules are representing. This dependency between modules is not allowed to be circular and can therefore be described using DAGs (directed acyclic graphs). A special module represents the starting point of the program. This module is called top module and is denoted by Top.

As a consequence of the module description hierarchy, the module instances, or simply the modules, are organized in a tree structure that is obtained by unfolding the underlying module description DAG where different instances of the same module description are represented as different nodes.

Figure 2.2 shows how a module definition forms a DAG, and a corresponding module instance is unfolded into a tree.

2.3.3 Language Constructs of BSV

Following is a short description of language constructs in BSV, these constructs are explained as they are used in the examples of BSV in this chapter.

**Package:** A collections of modules, they can be viewed in as similar to name spaces or libraries in languages such C++. 

Modules: The building blocks of BSV. Modules can be viewed as classes, they are though closer to type classes in Haskell.

Variable declaration: Variables declared in the module.

Module variable declaration: Instance of other modules used as part of a module, similar to variables.

Rules: Logic implemented in the module.

Methods: Are used to communicate with other modules. There are three types of methods;

  Action methods: Can change the values of registers in a module.
  Value methods: Gets value from registers in a module.
  ActionValue methods: Is a combination of action- and value methods.

Variable declaration and initialization: Two types of initialization are defined in BSV,

  Pure: (=) Where value on the right hand side is bound to the variable on left hand sight.

  Side-effect initialization: (<-) The right hand side has side effects and also returns a value. The return value is bound to the variable on the left hand side.

Interfaces: Define the blueprint to how modules can communicate with each other. The interfaces describe which methods need to be implemented so that the module can use a given interface for communication.

2.4 Informal Description of BSV Program Execution

The execution order of rules and which rules should fire simultaneously is defined in BSV by the Scheduler. The rules runs in single clock cycle but several rules can run in parallel in the same cycle. No two rules can attempt to update the same register of a module in the same clock cycle.

The BSV compiler is capable of detecting attempts to run more than one rule simultaneously, and issues errors if it is attempted. Hardware which violates these rules can not be created.
2.4.1 Scheduler

The scheduler is responsible for defining in which order the rules defined in all the modules of the program are run. That is, when the hardware is compiled, all the rules defined within each module are checked and a schedule of which rules can fire at the same clock cycle is given. The scheduler can be viewed as a switchboard operator which decides at each clock cycle which rules can fire in the next clock cycle.

The scheduler plays a key role in improving the parallel performance of the hardware, two different schedulers can result in different performance of the hardware, even though they are solving the same problem.

We have a limited access to controlling how the scheduler works, but we can insert commands into the programs to tell the compiler that certain rules are more urgent than others.

2.4.2 Describing the Modules

As explained above, a module is translated into hardware components that consist of a hierarchy of sub-components that all have access to their local states or registers. The registers are hardware components which contain the values of variables. We refer to registers containing values as if a single register could contain a single value. This is a simplification as a register can only contain a single bit of information, but they can be combined to store abstract values such as an integer.

These components can communicate with others up and down the hierarchy, where a parent module can get and push data to the child module. If a module needs information further down the hierarchy, that can be done by defining a function in the child module which then gets the data from its child and the data is returned to the grandparent module.

A module consists of a module name, value-variables, module instances, method definition and a collection of rules. The value variables represent the local registers, or the state, of that module and will therefore be updated when the module is executed. The module instances inside a module description represent the sub-components of the hardware. The methods take care of communication between modules, up and down the hierarchy. In BSV modules can interact across the module hierarchy by means of special constructs called wires. As this construction is under revision by the BSV designers team at the
moment and will most likely be changed soon, we decided not to handle it in this thesis.

The last part of the module is the collection of rules that correspond to the logical functions of the component and therefore to the executable part of the program. Each rule consists of two main parts: The guard and the body. The guard consists of a Boolean expression that has to evaluate to true in the locals state for the rule to be executed. The body consists of an imperative statement that aims at updating the local state.

### 2.5 Examples of BSV Programs

#### 2.5.1 Hello World

In order to understand how the language behaves, we give some code examples. We will start by giving simple examples and gradually build up to more complex examples.

The first example could be considered the "Hello World" of BSV. The program above will only display "Hello World"

The syntactic constructs of the example given in table 2.1

<table>
<thead>
<tr>
<th>Package Tb:</th>
<th>Defines the outermost construct of the program. All programs in BSV are parts of packages. TB is the name of the module.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(* synthesize *):</td>
<td>Tells the complier that the following module should be synthesizable to a circuit</td>
</tr>
<tr>
<td>module mkTb (Empty):</td>
<td>Defines the module named mkTb. (Empty) defines that the program is not implementing any interfaces.</td>
</tr>
</tbody>
</table>
rule helloWorld: Defines the rule helloWorld which is executed in each clock cycle

$display ("Hello World"): The $display construct is not part of BSV, but is used for debugging purposes in the simulator.

$finish(0): This is also not part of BSV. But since the execution of hardware is not intended to stop, we need a construct which tells the simulator to stop the running of a program.

2.5.2 Examples Reflecting Module Hierarchy

This example reflects module hierarchy in BSV.

Example 2.5.1 To illustrate how interfaces are used we give a code example. The example can be found in table 2.2.
Table 2.2: Example of a module hierarchy

The example shows how modules form a hierarchy:

We have two modules, mkTb and mkM1.

Module mkTb creates an instance of mkM1 called m1.

Module mkM1 implements the interface M1_ifc

Module mkTb has two rules, readvalue and countit.

Module mkM1 has one rule, display.

Both modules have local variables

When the program runs, all the rules can fire in the same clock cycle, since no two rules are trying to update the value of the same variables. The output of the program would be

Value of m1.x: 10
M1: \( x = 10 \)

Repeated until the counter reaches 5. Note that there is no guarantee that the order of the output would be as above, since we have no guarantee about the order of the execution. It is only guaranteed that the rules will fire if the can fire.

Example 2.5.2 This example to explain how to represent the module definitions and module instances. The example is adapted from [3].
package Tb;

(* synthesize *)
module mkTb (Empty);
    Empty m3 <- mkM3;
endmodule

// ------------------------------------------
(* synthesize *)
module mkM3 (Empty);
    Reg#(int) x <- mkReg(10);
    Reg#(int) count <- mkReg(0);
    M1_ifc m1b <- mkM1();
    M2_ifc m2 <- mkM2();
    rule ruleM3;
        $display("M1: x = %0d", x);
        if (count > 2)
            $finish (0);
    endrule
    rule counter;
        count <= count + 1;
    endrule
endmodule

// ------------------------------------------
interface M2_ifc;
    method int read_local_x ();
    method int read_sub_x ();
endinterface

(* synthesize *)
module mkM2 (M2_ifc);
    M1_ifc m1a <- mkM1();
    Reg#(int) x <- mkReg(10);
    rule ruleM2;
        $display("M2: m1a.x = %0d, x=%0d", m1a.read_local_x, x);
    endrule
    method int read_local_x ();
        return x;
    endmethod
    method int read_sub_x ();
        return m1a.read_local_x;
    endmethod
endmodule: mkM2

// ------------------------------------------
interface M1_ifc;
    method int read_local_x ();
endinterface

(* synthesize *)
module mkM1 (M1_ifc);
    Reg#(int) x <- mkReg(0);
    rule ruleM1;
        $display("M1: x = %0d", x);
    endrule
    method int read_local_x ();
        return x;
    endmethod
endmodule: mkM1
endpackage: Tb

Table 2.3: Concrete syntax example reflecting module hierarchy
Figure 2.2 gives an representation on how the module definitions can be represented as directed graph.

Figure 2.3 shows how the module instances are connected.

We now take a look at how the rules behave in example 2.3. Each rule affects the context of the instantiation of the module. That is, we need to check which instance of variables the rules affect.

Figure 2.4: Rules and which variables they read from
In the given example, no rules are in conflict since no rule is updating any of the state for each module. There are therefore no conflicts in running the rules. Figure 2.5 gives an example of how the scheduler for the program in example 2.3 might be constructed.

### 2.6 Formal Semantics

Formal semantics provides a mathematical description of the meaning of programming languages that can be used to reason about software systems in a formal way [9].

Programs are written using syntax and the meaning of the syntax is called the semantics. The semantics describe how the programs will behave. The compiler implementation, for a language, defines the program behaviour. The compiler is therefore one way do describe the semantics. Compilers are often built based on human languages descriptions of the language. These descriptions can be ambiguous or imprecise, which can lead to misunderstanding of how the program will behave.

Using formal semantics, the syntax is converted in to abstract syntax, an abstract representation of the program. A conversion is created from the abstract syntax to the semantic model. Using the model we can prove various properties of the language or programs written in the language. We can, for example, show that some language constructs are equivalent, or that two constructs in different languages describe the same behaviour. It
is therefore an appealing idea to define programming languages, using formal semantics and then base the compilers we build on it.

### 2.6.1 Different Types of Semantics

In the literature programming languages are described formally using different methods. However, the three types of semantics are the most common, are:

**Operational semantics**, which is intended to explain how a program is executed, using inference rules.

**Denotational semantics**, which is intended to explain the effect of executing programs, using mathematical functions.

**Axiomatic semantics**, which can be used to reason about partial correctness of a program, using predicate logic.

In this thesis, we focus on *Operational semantics* which can be divided into structural operational semantics (SOS) [11] and natural semantics [9]. SOS, or small-step semantics, focuses on how each statement in a program affects the state of the program. In SOS, the state of the program changes with each step of the computation, and it is easier to describe various parallel operations using SOS. In this thesis, we use natural semantics, or big-step semantics, that focuses on the overall effect of executing a complete clock cycle on the state.

### 2.6.2 Defining the Semantics of a Language

Operational semantics is a syntax driven approach, in the sense that we start with the syntax and define the transition rules for each of the constructs/structures in the language.

The language is divided into sets, abstract syntax categories. Each construct in the language such as statements and variables fall into one of the categories, and the categories are described using recursive definitions such as context-free grammar. We assume that the program we have has a parsed version of the language called an *abstract syntax tree*. The abstract syntax tree reflects the parts of the language we are interested in exploring. The semantic rules are given by means of inference rules of the form [1]

\[
\frac{\text{premise}}{\text{conclusion}}
\]
This could look as follows
\[
\frac{\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s''}{\langle S_1; S_2, s \rangle \rightarrow s''}
\]
where \(S_1\) and \(S_2\) are statements and \(s\) denotes the state of a program.

As an example of a semantics for a simple language, we define a language containing arithmetic expressions and variables. We define the semantics in the same manner as we will do in the following chapter.

The language has the following abstract syntactic categories [7]

\[\begin{align*}
    n & \in \mathbb{N} : \text{The set of numerals} \\
    a & \in \text{AExpr} : \text{The set of arithmetic expressions} \\
    x & \in \text{Var} : \text{The set of variables} \\
    S & \in \text{Stm} : \text{The set of statements}
\end{align*}\]

We define the state as
\[
\text{State} = \text{Var} \rightarrow \mathbb{N}
\] (2.1)

That is, we map each variable to a natural number. The constructs in the language are defined as

\[
\begin{align*}
    a & ::= x \mid n \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
    S & ::= x := a \mid S_1; S_2 \mid \varepsilon
\end{align*}
\]

In the language we can add, subtract and multiply numbers, assuming normal operational precedence.

We call \((S, s)\) a configuration, and the arrow "\(\rightarrow\)" is a from all configurations to a state defined as

\[
\rightarrow \subseteq (\text{Stm} \times \text{State}) \times \text{State}
\]

Table gives the natural semantics for the language. Looking at the rule \([\text{arith} - \text{add}]\), we can see that the expression \(a_1 + a_2\) can be evaluated given that we can evaluate \(a_1\) and \(a_2\), the same applies for the other rules. And lookup of a variable in the state \(s\) is defined as

\[
\mathcal{A}[x]_s = sx
\]
As can be seen from the rules we are giving instructions on how to evaluate each and every syntactical element of the language using inference rules.

The statements can give values to variables. The values of the variables are stored in a store, denoted by \( s \). We say that a configuration of the program we are running consists of statements and the state, denoted by the pair \( \langle S, s \rangle \). We refer to \( \langle S_1, s \rangle \) as a configuration and "→" as a transition relation, which together form a transition system.

The natural semantics rules can be found in table 2.4. In the rule [seq] we see how we end up in the overall state \( s'' \) if we run statement \( S_1 \) in state \( s \) ending up in state \( s' \), and similarly how we end up in state \( s'' \), which is the overall affect of running the rules in sequence. The rule assign shows how we update the state, by updating the values of the variable \( x \) by evaluating the arithmetic expression \( a \) in state \( s \). Some detail is left out, as we are give insight into what follows.

**Example 2.6.1** To illustrate the language described above, we give a simple example. Assuming that the state \( s \) contains the variables \( x \) and \( y \) and they are initially set to 0. We use the notation \( s[x \mapsto 10] \) to denote how the state \( s \) is updated.

The program

\[
\begin{align*}
\text{x := 10;} & \quad \text{y := x + 10}
\end{align*}
\]

Is sequential composition of two statements. Using the rules defined in 2.4 we get
\[
\frac{[x \mapsto 10]x \rightarrow 10, \ 10 \rightarrow 10}{x + 10 \rightarrow 20}
\]
\[
\langle y := x + 10, s[x \mapsto 10]\rangle \rightarrow s[x \mapsto 10, y \mapsto 20]
\]
\[
\langle x := 10, s \rangle \rightarrow s[x \mapsto 10], \ \langle y := x + 10, s[x \mapsto 10]\rangle \rightarrow s[x \mapsto 10, y \mapsto 20]
\]
\[
\langle x := 10; y := x + 10, s \rangle \rightarrow s[x \mapsto 10, y \mapsto 20]
\]
\[(2.2)\]

which gives us a state \( s \) where \( [x \mapsto 10, y \mapsto 20] \)

**Suggested Further Reading**

H. R. Nielson & F. Nielson: *Semantics with Applications: An Appetizer* [9].

H. Hüttel: *Transitions and Trees* [7].
Part II

Abstract Syntax and Semantics for \( \mu \text{BSV} \)
Chapter 3

Abstract Syntax and Semantics for $\mu$BSV

3.1 Abstract Syntax for $\mu$BSV

In this section we describe an abstract syntax for a suitable subset of the language BSV we call $\mu$BSV which includes enough of features to demonstrate its functionality, these features can be found chapter 2.

Syntactic categories of $\mu$BSV

The syntactic categories consist of names, module declarations, expressions and statements. The names consist of the following categories: ModNames represents the unique names given to each module definition; $\mathbb{N}$ represents the set of natural numbers; $\mathbb{B}$ represents the set of Boolean values; ValVar represent value variables which the values are assigned to, i.e. the registers of the module; ModVars, module instances variable which module instances are assigned to within a module; AMethNames and VMethNames represent the names given to action and value methods; RuleNames represent the names of rules defined within each module; MethValVars represent the variables that occur as formal parameters in the methods and should be disjoint to the set ValVar. The names can be summarized as follows.

Names:

- $n \in \mathbb{N}$: Numerals
- $b \in \mathbb{B}$: Boolean
Next we give a description of the declarations in the language. The system is given as a list of single module declarations ModDef. An executable program, on the other hand, is given as an instance of one of the module declarations in the list.

Each module in ModDef consist of five elements that each is defined recursively as a list consisting of the following categories: value variable declarations ValVarDec; modules instantiated and declared within other modules ModVarDec; the declared action methods or procedures AMethDec, which can return values; the declared value methods or functions VMethDec; the declared rules RuleDec. Each module is assigned to a unique module name from ModNames. All these recursively defined lists of declarations are defined simultaneously using parameters

\[ Par \in \{\text{ModDef, ValVar, ModVar, AMeth, VMeth, Rule}\} \]

where each of these categories is defined separately. The names can be summarized as follows.

**Declared Categories:**

\[ D_{\text{ModDef}} \in \text{ModDef}: \text{Module definitions}, \]
\[ D_{\text{ValVar}} \in \text{ValVarDec}: \text{Value variable declarations}, \]
\[ D_{\text{ModVar}} \in \text{ModVarDec}: \text{Module variable declarations}, \]
\[ D_{\text{AMeth}} \in \text{AMethDec}: \text{Action method declarations}, \]
\[ D_{\text{VMeth}} \in \text{VMethDec}: \text{Value method declarations}, \]
\[ D_{\text{Rule}} \in \text{RuleDec}: \text{Rule declarations}. \]
\[ D_{\text{ParList}}, Par \in \{\text{ValVar, ModVar, AMeth, VMeth, Rule}\}. \]
Syntax for Declarations:

\[ D_{ModDef} ::= module M is D_{ValVarList} D_{ModVarList} D_{AMethList} D_{VMethList} D_{RuleList} \]
\[ D_{ValVar} ::= valvar x ::= n, \]
\[ D_{ModVar} ::= modvar m : M, \]
\[ D_{AMeth} ::= method am with y is S, S \in Stm(X, X \cup \{y\}, L, N), \]
\[ D_{VMeth} ::= method vm is x, x \in X \]
\[ D_{Rule} ::= rule r is b :: S, b \in BExp(X, N), S \in Stm(X, Z, L, N), \]
\[ D_{ParList} ::= \varepsilon | D_{Par}; D_{ParList} for Par \in \{ValVar, ModVar, AMeth, VMeth, Rule\} \].

The definitions of the methods and rules depend on the categories of statements \( Stm(X, Z, L, N) \) and arithmetic and Boolean expressions \( AExp(X, N) \) and \( BExp(X, N) \). These categories are parametrized with the sets of variables and other items that are allowed to be used in the module declarations. For instance for a module that occurs in a list of modules to be well defined, the set \( L \) in \( Stm(X, X \cup \{y\}, L, N) \) in both types of methods has to be restricted to the set of pairs consisting of a module name, defined in the child modules of \( M \), and an action method name defined inside the module description description. The parameters \( X \) and \( X \cup \{y\} \) indicate that variables can only be assigned to variables from \( X \) but the expressions that occur on the right hand side of an assignment can both include variables from \( X \) and \( y \). Then \( y \) is bound by the formal parameter of the method. Later we will define formally when a module definition is well-formed and then these parameters will be explained better. Otherwise the categories are standard and will not be described in more detail.

Categories of Expressions and Statements:

Arithmetic expressions:

\[ a \in AExp(X, N), X \subseteq ValVar, N \subseteq ModNames \times VMethNames \]

Boolean expressions:

\[ b \in BExp(X, N), X \subseteq ValVar, N \subseteq ModNames \times VMethNames \]

Statements:

\[ S \in Stm(X, Z, N, L), X, Z \subseteq ValVar, N \subseteq ModNames \times VMethNames, L \subseteq ModNames \times AMethNames \]
Syntax for Expressions and Statements:

\[\text{AExp}(X, N) : \]
\[a ::= n \in \mathbb{N} \mid x \in X \mid a + a \mid a - a \mid a \cdot a \mid M.\text{vm}, (M, \text{vm}) \in N,\]

\[\text{BExp}(X) : \]
\[b ::= \text{tt} \mid \text{ff} \mid a = a \mid a \leq a \mid b \lor b \mid \neg b, a \in \text{AExp}(X, N),\]

\[\text{Stm}(X, Z, N, L) : \]
\[S ::= x := a, x \in X, a \in \text{AExp}(Z, N) \mid M.\text{am}(a), (M, \text{am}) \in L, a \in \text{AExp}(Z, N) \mid \varepsilon,\]

**Example 3.1.1** We now give an example of a simple program defined using the syntax above.

\[
\begin{align*}
\text{module } Top \text{ is} \\
\text{valvar } x := 0 \\
\text{rule } first \text{ is } x < 10 := x := x + 1
\end{align*}
\]

This program consists of a single module, Top. The module has one register \( x \), that gets the initial value 0, and one conditional rule.

The syntax in the example is constructed as follows.

\[
\begin{align*}
D_{\text{ModDef}} ::= & \text{module } Top \text{ is } D_{\text{ValVarList}} D_{\text{ModVarList}} D_{\text{MethList}} D_{\text{RuleList}} \\
D_{\text{ValVarList}} ::= & \text{valvar } x := 10 \\
D_{\text{ModVarList}} ::= & \varepsilon \\
D_{\text{MethList}} ::= & \varepsilon \\
D_{\text{RuleList}} ::= & \text{rule } first \text{ is } x < 10 := x := x + 1
\end{align*}
\]
Example 3.1.2 We now give a more complicated example where we have multiple modules.

module $M_1$ is
valvar $x := 0$
method $am$ with $y$ is $x := y$
module $Top$ is
valvar $z := 0$
modvar $m_1 : M_1$
rule $first$ is $(z < 5) :: (m_1.am(z); z := z + 1)$

This program contains two modules $Top$ and $M_1$. We have one instance of each module, $top$ and $m_1$ respectively. The instance $top$ contains one rule where a method is called within $m_1$. The method $me$ has one parameter, and the method sets the value of a local register $x$ to the value passed to it.

$D_{\text{ModDef}} ::= \text{module } Top \text{ is } D_{\text{ValVarTop}} D_{\text{ModVarTop}} D_{\text{RuleTop}}$

$D_{\text{ValVarTop}} ::= \text{valvar } x := 0$

$D_{\text{ModVarTop}} ::= \text{modvar } m_1 : M_1$

$D_{\text{RuleTop}} ::= \text{rule } r \text{ is } (z < 5) :: (m_1.am(z); z := z + 1)$

$D_{\text{ValVarM1}} ::= \text{valvar } x := 0$

$D_{\text{MethM1}} ::= \text{method } am \text{ with } y \text{ is } x := y$

In the examples we omit showing the $; \varepsilon$ that should occur at the end of each construct according to the syntax. In order to understand how the program works, it is easier to see the program structure visually. Figure 3.1 shows how two module definition are connected. The instantiation of module $M_1$ the $Top$ module references the definition of for module $M_1$.

If the program would be laid out in hardware, two modules would be created. A instance of the $Top$ module, $top$, and instance of the module $M_1$, $m_1$, would have a connection, using wires, to the $Top$ module trough the method $me$. 
3.2 Natural Semantics for $\mu$BSV

In this section we will define a formal model for the language $\mu$BSV in terms of Natural semantics. For this purpose we model a module with the name $M$ as a 5-tuple of the form

$$\langle \text{sto}_M, \text{mods}_M, \text{ameths}_M, \text{vmeths}_M, \text{rules}_M \rangle.$$ 

Here $\text{sto}_M$ represents the store or the registers of the module $M$, $\text{mods}_M$ its set of submodules, $\text{ameths}_M$ and $\text{vmeths}_M$ its action and value methods respectively and $\text{rules}_M$ the set of rules defined in the module. We define recursively a partial function $\mathcal{M}$ that assigns a module $\mathcal{M}_M$, of the form described above, to its module name $M \in \text{ModNames}$. All the components of the module will be defined simultaneously and depend on the definition of the set of action methods $\text{AMethods}(X, Y, N, L)$ and value methods $\text{VMethods}(X, Y, N, L)$, and the set of rules $\text{Rules}(X, N, L)$ depend on the syntactic categories $\text{Stm}(X, Y, N, L)$ which again depends on $\text{AExp}(X, N)$, $\text{BExp}(X, N)$. The parameters allows us to restrict the definitions to subsets that make sense for the current module according to its position in the module hierarchy. For example, a rule only allowed to call methods that are defined in its sub modules, i.e. in its children in the DAG that describes the module. We start by formally define the models for methods and rules.
**Definition 3.2.1 (Methods and Rules)** We define the sets of action methods, AMethods, value methods, VMethods, and rules, Rules, with parameters as follows:

\[
\text{AMethods}(X, Y, N, L) = \{ \langle y, S \rangle \mid S \in \text{Stm}(X, X \cup \{y\}, L, N), y \in Y \}
\]
where \( X \subseteq \text{ValVar}, \ Y \subseteq \text{MethValVars}, \)
\( L \subseteq \text{ModNames} \times \text{AMethNames} \)
and \( N \subseteq \text{ModNames} \times \text{VMethNames} \).

\[
\text{VMethods}(X) = \{ x \mid x \in X \}
\]
where \( x \subseteq \text{ValVar} \)

\[
\text{Rules}(X, L, N) = \{ \langle b, S \rangle \mid b \in \text{BExp}(X, X, L, N), S \in \text{Stm}(X, X, L, N) \},
\]
where \( X \subseteq \text{ValVar} \), \( L \subseteq \text{ModNames} \times \text{AMethNames} \)
and \( N \subseteq \text{ModNames} \times \text{VMethNames} \).

The parameters we used for restrictions, define which ActionMethods, ValueMethods and Rules which are applicable in the module. The parameter \( X \) represents the value variables, \( Y \) represents the method value variables, \( N \) is the set of pairs of modules names and value methods and \( L \) is the set of pairs of module and action methods which can be used.

**Definition 3.2.2 (Modules)**
We define the set of modules \( M \) recursively as follows:

**Base case:** If

\[
M \in \text{ModNames},
\]
\( Y_M \subseteq \text{MethValVars}, \)
\( \text{sto}_M \in \text{ValVar} \rightarrow \text{fin} \mathbb{N}, \)
\( \text{mods}_M = \emptyset, \)
\( \text{ameths}_M \in \text{AMethNames} \rightarrow \text{fin} \text{AMethods}(\text{dom}(\text{sto}_M), Y_M, \emptyset, \emptyset), \)
\( \text{vmeths}_M \in \text{VMethNames} \rightarrow \text{fin} \text{VMethods}(\text{dom}(\text{sto}_M)), \)
\( \text{rules}_M \in \text{RuleNames} \rightarrow \text{fin} \text{Rules}(\text{dom}(\text{sto}_M), \emptyset, \emptyset) \)

then

\[
\mathcal{M} = \{ M \rightarrow \langle \text{sto}_M, \text{mods}_M, \text{ameths}_M, \text{vmeths}_M, \text{rules}_M \rangle \} \in \mathcal{M},
\]
\( \text{Names}(\mathcal{M}) = \{ M \}. \)
Inductive case: If
\[ M_1, M_2 \in M, \text{ and} \]
\[ \text{Names}(M_1) \cap \text{Names}(M_2) = \emptyset \]
then
\[ M_1 \cup M_2 \in M, \text{ and} \]
\[ \text{Names}(M_1 \cup M_2) = \text{Names}(M_1) \cup \text{Names}(M_2). \]

If
\[ M \in M, \]
\[ M \in \text{ModNames} \setminus \text{Names}(M), \]
\[ Y_M \subseteq \text{MethValVars}, \]
\[ \text{sto}_M \in \text{ValVar} \rightarrow_{f_{\text{in}}} \mathbb{N}, \]
\[ \text{mods}_M \in \text{ModVars} \rightarrow_{f_{\text{in}}} \text{Names}(M), \]
\[ \text{ameths}_M \in \text{AMethNames} \rightarrow_{f_{\text{in}}} \text{AMethods}(\text{dom}(\text{sto}_M)), Y_M, \text{AMethodsPairs}(\text{range}(\text{mods}_M)), \]
\[ \text{vmeths}_M \in \text{VMethNames} \rightarrow_{f_{\text{in}}} \text{VMethods}(\text{dom}(\text{sto}_M)), \]
\[ \text{rules}_M \in \text{RuleNames} \rightarrow_{f_{\text{in}}} \text{Rules}(\text{dom}(\text{sto}_M), \text{dom}(\text{sto}_M), \text{AMethodsPairs}(\text{range}(\text{mods}_M)), \text{VMethodsPairs}(\text{range}(\text{mods}_M))), \]
then
\[ \bullet \ M \cup \{ M \rightarrow \langle \text{sto}_M, \text{mods}_M, \text{ameths}_M, \text{vmeths}_M, \text{rules}_M \rangle \} \in M, \text{ and} \]
\[ \bullet \ \text{Names}(M \cup \{ M \rightarrow \langle \text{sto}_M, \text{mods}_M, \text{ameths}_M, \text{vmeths}_M, \text{rules}_M \rangle \}) = \text{Names}(M) \cup \{ M \}. \]

In the definition above, for \( A \subseteq \text{ModNames} \), we use the notation
\[ \text{AMethodsPairs}(A) = \{ \langle M', am \rangle \mid am \in \text{dom}(\text{ameths}_{M'}), M' \in A \}. \]

and similarly for \( \text{VMethodsPairs} \).

Above we have defined a semantic model for a module description. From this we will derive a model for an instance \( m \) of a module \( M \). We could do this by simply adding the name \( n \) to \( M \) in the model \( M \). But instead of doing so we define the model \( T_{M,m} \) as a
set of elements of the form $m_1m_2\ldots m_n : M_n$ where $M_n$ is a module name that occurs in $\mathcal{M}$ and $m_1 = m$ and $m_2\ldots m_n$ is the sequence of module instance names that occur on the path from $M$ to $(m_n, M_n)$ in $\mathcal{M}$. The corresponding set of strings over module variables $m_1m_2\ldots m_n$ is clearly prefix closed and therefore represents a tree (hence the notation $\mathcal{T}$). From $\mathcal{T}$ and $\mathcal{M}$ we can derive similar representation of a tree with the sets $\text{sto}$, $\text{ameths}$, $\text{vmeths}$, $\text{rules}$ as leaves instead of the module names. We start by defining $\mathcal{T}$ for a given $\mathcal{M}$, $M$ and $m$.

**Definition 3.2.3** Assume that $\mathcal{M}$ is a module hierarchy, $M \in \text{ModNames}$ is the top module of $\mathcal{M}$ and that $m \in \text{ModVars}$. We define the model $\mathcal{T}_{M,m}$ for the instance $m$ of $M$ as follows:

- $(m, M) \in \mathcal{T}_{M,M,m}$,
- If $(\sigma, M) \in \mathcal{T}_{M,M,m}$ and $(M, \langle \text{sto}, \{(m_1, M_1), \ldots, (m_k, M_k)\}, \text{ameths}, \text{vmeths}, \text{rules}\rangle) \in \mathcal{M}$ then $\{(m_1, M_1), \ldots, (m_k, M_k)\} \subseteq \mathcal{T}_{M,M,m}$.

We define the tree of stores $\mathcal{S}_{M,m}$, $\mathcal{A}_{M,M,m}$, and rules $\mathcal{R}_{M,m}$ respectively as follows:

$(\sigma m', M') \in \mathcal{T}_{M,M,m}$ iff

- $(\sigma m', \text{sto}_{M'}) \in \mathcal{S}_{M,M,m}$, and
- $(\sigma m', \text{rules}_{M'}) \in \mathcal{R}_{M,M,m}$.

- $(\sigma m', \text{ameths}_{M'}) \in \mathcal{A}_{M,M,m}$.

### 3.2.1 The Semantics for Module Declaration

We now define the abstract syntax of a syntactic module by means of a translation. We assume that the syntax is well formed.

We use the notation $[\cdot]$ to define a function which converts from the abstract syntax to the semantics model. The function is defined recursively

**Definition 3.2.4** Given a module definition $D_{\text{ModDef}}$, defined recursively as

$$D_{\text{ModDef}} ::= \text{module } M \text{ is } D_{\text{ValVarList}}\ D_{\text{ModVarList}}\ D_{\text{AMethList}}\ D_{\text{VMethList}}\ D_{\text{RuleList}};\ D_{\text{ModDef}} \mid \epsilon$$
\[[D_{\text{ModDef}}]:\]
\[
\begin{align*}
\epsilon = \emptyset \\
\text{module } M \text{ is } D_{\text{ValVarList}} \ D_{\text{ModVarList}} \ D_{\text{AMethList}} \ D_{\text{VMethList}} \ D_{\text{RuleList}}; D_{\text{ModDef}} \end{align*}
\]
\[
= \{ M \rightarrow \langle [D_{\text{ValVarList}}], [D_{\text{ModVarList}}], [D_{\text{AMethList}}], [D_{\text{VMethList}}], [D_{\text{RuleList}}] \rangle \} \\
\cup [D_{\text{ModDef}}]
\]

Where

\[[D_{\text{ValVar}}]:\]
\[
\begin{align*}
\epsilon = \emptyset \\
\text{valvar } x := n \ D_{\text{ValVarList}} = \{ (x, n) \} \cup [D_{\text{ValVarList}}]
\end{align*}
\]

\[[D_{\text{ModVar}}]:\]
\[
\begin{align*}
\epsilon = \emptyset \\
\text{modvar } m : M \ D_{\text{ModVarList}} = \{ m : M \} \cup [D_{\text{ModVarList}}]
\end{align*}
\]

\[[D_{\text{AMeth}}]:\]
\[
\begin{align*}
\epsilon = \emptyset \\
\text{method } am \text{ with } y \text{ is } S \ D_{\text{AMethList}} = \{ am \rightarrow \langle y, S \rangle \} \cup [D_{\text{AMethList}}]
\end{align*}
\]

\[[D_{\text{VMeth}}]:\]
\[
\begin{align*}
\epsilon = \emptyset \\
\text{method } vm \text{ is } x \ D_{\text{VMethList}} = \{ vm \rightarrow x \} \cup [D_{\text{VMethList}}]
\end{align*}
\]

\[[D_{\text{Rule}}]:\]
\[
\begin{align*}
\epsilon = \emptyset \\
\text{rule } r \text{ is } b :: S \ D_{\text{RuleList}} = \{ r \rightarrow \langle b, S \rangle \}
\end{align*}
\]

**Proposition 3.2.5** Well formed translation

Given that the syntax is well formed a module formed by the translation will be well-formed.

### 3.2.2 Semantic Rules

In this subsection we will describe the natural semantic rules for $\mu\text{BSV}$. To simplify the notation we simply write $S, T$, etc instead of $S_{M,(m,M)}, T_{M,(m,M)}$, etc.

**Definition 3.2.6** We say that a set of rules $R$ is consistent, denoted $\text{cons}(R)$, if no two rules in the set attempt to write to the same register (in the same clock cycle).
Modules:

\[
S \vdash (R, \emptyset) \rightarrow U'
\]
\[
\mathcal{M}, (m, M) \vdash S \rightarrow S[U']
\]
\[
R \subseteq \mathbb{R}_{M,M,m,cons(R)}
\]

Rules:

\[
S \vdash ([\sigma_i, r_i], U_{i-1}) \rightarrow U_i, 1 \leq i \leq L
\]
\[
S \vdash ([\sigma_1, r_1], \ldots, [\sigma_L, r_L], U_0) \rightarrow U_L
\]
\[
S(\sigma) \vdash b \rightarrow \text{tt}, S, \sigma \vdash (stm, U) \rightarrow U'
\]
\[
S(\sigma) \vdash ([\sigma, r], U) \rightarrow U'
\]
\[
\mathbb{R}(\sigma)(r) = (b, stm)
\]

Statements:

\[
S(\sigma) \vdash e \rightarrow v
\]
\[
S, \sigma \vdash (x := e, U) \rightarrow U[(\sigma, (x, v))]
\]
\[
S(\sigma) \vdash a \rightarrow v; S([(\sigma m, (y, v))], \sigma m \vdash (S, U) \rightarrow U'
\]
\[
S, \sigma \vdash (m. am(a), U) \rightarrow U'
\]
\[
\mathcal{A}(\sigma m)(am) = (y, S)
\]

Expressions:

\[
S(\sigma) \vdash x \rightarrow S(\sigma)(x)
\]
\[
S(\sigma) \vdash e_1 \rightarrow v_1; S(\sigma) \vdash e_2 \rightarrow v_2
\]
\[
S(\sigma) \vdash e_1 \ op \ e_2 \rightarrow (v_1 \ op \ v_2)
\]

Figure 3.2: Big step operational semantics for BSV

We give an informal description on how the rules described in 3.2 behave. The semantic rule [Clock – Cycle] defines everything which takes place in each clock cycle. We can update the state \( S \) with the update, \( U' \), based on a set of rules, given that the rules are consistent. Using [Rule – Set], we defined how, given a set of rules we collect them and run them resulting in a update on the state \( U_L \). The rule can only fire, give that the guard, \( b \), evaluates to true. [Rule] defines how each statement gives rise to an update \( U \) of the state. [update] defines how the update is performe. The evaluation of method calls are defined using [meth – call]. The rule [variable] defines how we look up values in the store \( S \). The evaluation of arithmetic expressions is then defined using [operator].
### 3.2.3 Examples

#### Example 3.2.7
This is an example of a µBSV program which contains one module with one variable and one rule.

```plaintext
module M1 is
    valvar x := 0
    rule first is x < 10 :: x := x + 1

We translate the syntax to a module:

\[
\begin{align*}
    \text{sto}_{M1} &= \{x \mapsto 0\} \\
    \text{mods}_{M1} &= \emptyset \\
    \text{ameths}_{M1} &= \emptyset \\
    \text{vmeths}_{M1} &= \emptyset \\
    \text{rules}_{M1} &= \{\text{first} \mapsto (x < 10, x := x + 1)\}
\end{align*}
\]

Which gives us the module:

\[
\{M1 \mapsto (\text{sto}_{M1}, \text{mods}_{M1}, \text{ameths}_{M1}, \text{vmeths}_{M1}, \text{rules}_{M1})\} \in \mathcal{M}
\]

Using semantic rules where the initial value of \(S(m1)(x) = 0\), we get, and top module is \(m1 : M1\). Figure 3.3 shows us how we apply the semantic rules for the first clock cycle. The final value of the store is \(S(m1)(x) = 1\).

The execution in the second clock cycle are shown in figure 3.4, after the execution, the store has been updated to \(S(m1)(x) = 2\).
Example 3.2.8 Here we have two modules, two rules and one action method call.

module $M_1$ is

valvar $x := 5$

modvar $m_2 : M_2$

rule $r_1$ is $(x < 10) :: (m_2.am(x))$

rule $r_2$ is $(x < 10) :: (x := x + 1)$

module $M_2$ is

valvar $z := 0$

method $am$ with $y$ is $z := y$
The modules are defined as follows:

\[
\begin{align*}
sto_{M_1} &= \{x \mapsto 5\} \\
mods_{M_1} &= \{m2 \mapsto M2\} \\
ameths_{M_1} &= \emptyset \\
vmeths_{M_1} &= \emptyset \\
rules_{M_1} &= \{r1 \mapsto (x < 10, m2.am(x)), r2 \mapsto (x < 10, x := x + 1)\}
\end{align*}
\]

Which gives us

\[
\{M_1 \rightarrow \langle sto_{M_1}, mods_{M_1}, ameths_{M_1}, vmeths_{M_1}, rules_{M_1} \rangle \} \in M
\]

And for the module \(M_2\)

\[
\begin{align*}
sto_{M_2} &= \{z \mapsto 0\} \\
mods_{M_2} &= \emptyset \\
ameths_{M_2} &= \{am \mapsto (y, z := y)\} \\
vmeths_{M_2} &= \emptyset \\
rules_{M_2} &= \emptyset
\end{align*}
\]

Which gives us

\[
\{M_2 \rightarrow \langle sto_{M_2}, mods_{M_2}, ameths_{M_2}, vmeths_{M_2}, rules_{M_2} \rangle \} \in M
\]

Figure 3.5 shows how one clock cycle is executed in example 3.2.8.

In the example \(R = \{[m1, r1], [m1, r2]\}\).
\[
\frac{S(m1) \vdash x \rightarrow y}{S((m1m2, (y, 5))], (m1m2) \vdash y \rightarrow 5}
\]
\[
\frac{S((m1m2, (y, 5))], m1m2 \vdash (z := y, \emptyset) \rightarrow (m1m2, (z, 5))}{S, m1 \vdash (m2.am(x), \emptyset) \rightarrow (m1m2, (z, 5))}
\]
\[
\frac{S(m1) \vdash x \rightarrow 5; S((m1m2, (y, 5)))], m1m2 \vdash (z := y, \emptyset) \rightarrow (m1m2, (z, 5))}{S, m1 \vdash (m2.am(x), \emptyset) \rightarrow (m1m2, (z, 5))}
\]
\[
\frac{S(m1) \vdash x < 10 \rightarrow \text{tt}, S, m1 \vdash (m2.am(x), \emptyset) \rightarrow (m1m2, (z, 5))}{S \vdash ([m1, r1]) \rightarrow (m1m2, (z, 5)) \quad \text{R}(m1)(r1) = \langle x < 10, m2.am(x) \rangle}
\]
\[
\frac{S \vdash ([m1, r1], \emptyset) \rightarrow (m1m2, (z, 5)), S \vdash ([m1, r2], (m1m2, (z, 5))) \rightarrow (m1m2, (z, 5)) \cup (m1, (x, 6))}{S \vdash \{(\{m1, r1\}, [m1, r2]), \emptyset) \rightarrow \{(m1m2, (z, 5)), (m1, (x, 6))\}}
\]
\[
\frac{S \vdash (R, \emptyset) \rightarrow \{(m1m2, (z, 5)), (m1, (x, 6))\}}{M, (m1, M1) \vdash S \rightarrow S\{(m1m2, (z, 5)), (m1, (x, 6))\}}
\]

Figure 3.5: The execution of the second clock cycle in example 3.2.8
Part III

Conclusions
Chapter 4

Conclusions

The work which was done in this project was towards building formal semantics for the hardware description language BSV. The execution of programs using hardware differs from what we are used to when working with sequential execution of programs on a CPU. In the hardware execution, we are usually working with parallel operations, which are performed on a bit level. Hardware description languages define how these operations are performed and how the hardware interacts with other components.

Formal semantics define the behaviour of a programming language, using mathematical models. Formal semantics can be used to reason about the behaviour of a program and prove various properties of the language. Furthermore it can be used to reason about behavioural equivalence of programs.

The most notable results we provide is the formal semantics for the language $\mu$BSV, which is a sub-set of BSV. The language $\mu$BSV is intended to reflect important behavioural aspects of BSV. As BSV has hierarchical structure similar to object-oriented languages, the objects in BSV are called modules. The hierarchical module behaviour was described. In BSV statements are contained inside guarded atomic actions, called rules. All statements inside rules are executed simultaneously. A set of rules can be executed in the same clock-cycle, provided that they are not updating the same registers. The semantic reflects this parallel execution of rules. The formal semantics we defined is based on the description of the language give in [9].

We provide formal semantics for the language $\mu$BSV using natural semantics.
4.1 Related Work

To our knowledge, work towards defining the formal semantics of hardware description languages is scarce. In [2] Arvind et al. define formal semantics which encapsulates the scheduling aspects of BSV, some important aspects of the language are omitted such as the module behaviour. In [14] Singh and Shulka translated a subset of BSV into PROMELA which is the specification languages of the SPIN model checker. In [13] Richard and Lester use the PSV theorem prover to verify various properties of programs in BSV. In [15] Smit et al. describe the mathematical approach to hardware design by describing the hardware using functional programming methods. In [8] Mueller et al. provide simulation based semantics for the language SystemC where the semantics is defined using distributed Abstract State Machines (ASMs). To our knowledge, limited work has been done towards defining formal semantics of languages which have object-oriented behaviour. In [7] Hüttel defines formal semantics for a concurrent object-oriented language called Cola. Other approaches where simulation based semantics is defined for object-oriented languages have been made. In [12] Ravazi and Sirjani define compositional semantics of System-level designs in SystemC. The descriptions are based on the coordination language Reo where SystemC designs are mapped to Reo circuits.

4.2 Future work

The following items describe the future work we think would be interesting to investigate:

- Most notably, verifying that the semantics given for μBSV reflects BSV adequately, specifically in regards to the timing aspects of the language.

- Describing formally a larger sub-set of BSV, including the type-system and interfaces.

- Converting the language, based on the formal semantics to a graphical model checker such as UPPAAL [5]. This would create a platform where the designer could easily verify properties of the programs in an automated fashion.

- Using a model checker, such as UPPAAL, various properties of programs could be verified using logical formulas. Using a model checker one could test for properties such as if the program dead-locks or if some unwanted states can be reached.
- Use the formal semantics to reason about behavioural equivalence of programs. Behavioural equivalence could be defined using simulation preorder or bisimulation. The behavioural equivalence can be used to reason about aspects of programs such as:

  – Is the program behaving in accordance to the specification.

  – Given two programs, where one is more efficient, are those programs equivalent. Efficiency in this context could be either in regards to power consumption of the hardware or how fast the hardware performs.
Bibliography


Appendix A

Formalism used in defining the Semantics

A.1 Strings and Trees

(We start with some general definitions (independent of the syntax).)

Definition A.1.1 Let $C$ be any (finite) set.

1. A set $B \subseteq C^*$ is prefix closed if whenever $c \in C$ and $s \cdot c \in B$ then $s \in B$.

2. If $B \subseteq C^*$ and $s \in C^*$, let $B \cdot s = \{s' s \mid s' \in B\}$.

3. If $f : C^* \mapsto (V ar \mapsto N)$ then

$$ (f_{s,(x,n)}(s))(x) = n, (f_{s,(x,n)}(t))(y) = (f(t))(y) \text{ if } t \neq s \text{ or } y \neq x. $$

Definition A.1.2 ($C$-labelled trees)

- The set of $C$-labelled trees is defined as the smallest set that satisfies the following:
  - $(c, \emptyset) \in Trees(C)$ for all $c \in C$,
  - $T_1, \ldots, T_k \in Trees(C)$ and $c \in C$ implies $(c, \{T_1, \ldots, T_k\})$.
- $\text{root}((c, X)) = c$.

Example A.1.3 Let $X = \{t_0, t_1, t_2, t_3\}$. We give an example of how we construct a $X$-labelling for the tree given in figure A.1

$$ T = (t_0, \{(t_1, \{(t_3, \emptyset)\}), (t_2, \emptyset)\}) $$
Definition A.1.4 Assume that $T$ is a $C$ labelled tree.

- Define $\mathcal{T}(T) \subseteq C^*$ as follows:
  - $\mathcal{T}((c, \emptyset)) = \{c\}$,
  - $\mathcal{T}((c, \{T_1, \ldots, T_k\})) = \{c\} \cdot (\mathcal{T}(T_1) \cup \ldots \cup \mathcal{T}(T_k))$.

Example A.1.5 Using the tree in figure A.1 we give an example on how we can generate the prefix order for the X-labelled tree using the definition given above

$$\mathcal{T}((t_0, \{(t_1, \{(t_3, \emptyset\})\}, (t_2, \emptyset))) = \{t_0\} \cdot (\mathcal{T}((t_1, \{(t_3, \emptyset\})) \cup \mathcal{T}((t_2, \emptyset)))$$

$$= \{t_0\} \cdot (t_1 \cdot (T((t_3, \emptyset)) \cup t_2))$$

$$= \{t_0\} \cdot ((t_1 \cdot \{t_3\}) \cup \{t_2\})$$

from which we see that the following strings are in the set

$$\mathcal{T}(T) = \{t_0 \cdot t_1 \cdot t_3, t_0 \cdot t_2\}$$

Definition A.1.6 We define the prefix closure of a set, $S$ as

$$\text{Pref}(S) = \{b \mid \exists c. bc \in S\}$$

Example A.1.7 The prefix closure, $\text{Pref}(\mathcal{T}(T))$, from the example above would be:

$$\text{Pref}(\mathcal{T}(T)) = \{t_0, t_0 \cdot t_1, t_0 \cdot t_1 \cdot t_3, t_0 \cdot t_2\}$$

Lemma A.1.8 If $\text{Tree}$ is a $C$-labelled tree then $\mathcal{T}(\text{Tree}) \subseteq C^*$ is prefix closed.
A.2 Mappings

1. If $F, G : A \rightarrow B$, then $G[F]$ is defined by
   \[
   \forall a \in A. G[F](a) = F(a) \text{ if } a \in \text{dom}(F), G(a) \text{ otherwise.}
   \]

2. We let $[x_1 \mapsto a_1, \ldots, x_k \mapsto a_k]$ denote the mapping that maps $x_i$ into $a_i$ for $0 \leq i \leq k$. 
Appendix B

Implementing logic with transistors

We want the hardware to perform logic and being able to store and retrieve data. Every action we want the hardware to perform is performed by transistors. The transistors perform logical operations and they store the data we are working with. Every processor performs its actions by using transistors.

Logical devices are built up from smaller building blocks which can then be combined to perform ever more complicated tasks, but the principles remain the same. We will given an example of how we can do logical operations using simple models of transistors. Building from the simple model we gave in figure 2.1, we give examples of how we can create simple elements such as devices which perform \textit{and operation} and \textit{or operation}.

In figure B.1 we can see that we can combine transistors to create logical operations. These operations can then be combined even further to create more complicated logical devices.

In B.1 we have a simple model of a \textit{NAND} and a \textit{NOR} logic gates. The perform the boolean actions of \textit{NOT AND} and \textit{NOT OR}. B.1 Gives us a truth table for the \textit{NAND} gate. The supply voltage is set to \textit{HIGH}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
A & B & \textit{V}_{\text{Out}} \\
\hline
LOW & LOW & HIGH \\
HIGH & LOW & HIGH \\
LOW & HIGH & HIGH \\
HIGH & HIGH & LOW \\
\hline
\end{tabular}
\caption{Truth table for NAND gate}
\end{table}
Registers are transistor elements consisting of several transistors connected together in such a way that a single bit of information can be stored.

An detailed description on how we can can convert from RTL to logical gates can be found in [6].
Appendix C

BSV Code Examples

In C.1 an example is given on how to calculate Collatz series of numbers.

```haskell
package Tb;
(* synthesize *)
module mkTb (Empty);
    Reg#(int) x <- mkReg (11);
    Reg#(int) count <- mkReg (0);

    // Rule used to display the number
    rule displayNum;
        $display("X %d", x);
        count <= count + 1;
    endrule

    // The case for even numbers
    rule evenX if (x % 2 == 0);
        x <= x / 2;
    endrule

    // The case for odd numbers
    rule oddX if (x % 2 == 1);
        x <= x*3 + 1;
    endrule

    // If the number is equal to 1, we end
    rule finish if (x == 1);
        $display("Count: %0d", count);
        $finish(0);
    endrule

    // To be safe
    rule lessthanzero if (x < 0);
        $display("less than zero");
    endrule

endmodule
endpackage: Tb
```

Table C.1: Algorithm implemented in BSV
Appendix D

Data structures for BSV

D.1 Data structures for BSV

We now start defining the functions that build the data structures that represent the syntax of a BSV program. We start describing an algorithm which takes a list of module declarations, $D_{\text{ModDefList}}$, and builds a corresponding adjacency list that describes the dependency of modules in the program. More precisely the algorithm $\text{FullList}$ creates a set, $\text{FullList}(D_{\text{ModDefList}})$, of pairs of the form $M_1 \rightarrow M_2$, where $M_1$ contains $M_2$ in its list of sub modules, i.e. $M_1$ parent module of $M_2$. As described in Section 2.3.2, if the program is syntactically well-formed, this adjacency list represents a directed acyclic graph (DAG).

D.1.1 Module hierarchy structure

The algorithm $\text{FullList}$ calls the algorithm $\text{SingleMod}$, that creates the adjacencies generated by a single module definition recursively. That algorithm calls the algorithm $\text{ModuList}$ that recursively goes through list of module instances declared inside the single module $M$; it adds $M \rightarrow M'$ into the adjacency list if there is a module declaration of the form $m' : M'$ inside $M$. 
Algorithm 1: FullList

**Input:** The module definition list $D_{ModDefList}$.

**Output:** The adjacency list $\text{FullList}(D_{ModDefList})$.

\[
\text{FullList}(\varepsilon) = \emptyset ; \\
\text{FullList}(D_{ModDef}; D_{ModDefList}) = \\
\text{FullList}(D_{ModDefList}) \cup \text{SingleMod}(D_{ModDef})
\]

The algorithm SingleMod takes as input a single module definition and calls the algorithm ModuList passing it the module variable definition.

Algorithm 2: SingleMod

**Input:** A single module definition $D_{ModDef}$.

**Output:** The adjacency list $\text{SingleMod}(D_{ModDef})$ creates.

\[
\text{SingleMod}(\text{module } M \text{ is } D_{ValVarList} D_{ModVarList} D_{MethodList} D_{RuleList}) = \\
\text{ModuList}(M, D_{ModVarList})
\]

### D.1.2 Module methods structure

The algorithm ModuList takes as input a module variable definition and processes each definition at a time and returns an set which represents which methods are defined in the module.

Algorithm 3: ModuList

**Input:** A parent module $M$ and the list of module variable declaration $D_{ModVarList}$ for $M$.

**Output:** The adjacency set $\text{ModList}(M, D_{ModVarList})$.

\[
\text{ModuList}(M, \varepsilon) = \emptyset ; \\
\text{ModuList}(M, \text{(modvar } m_1: M_1) D_{ModVarList}) = \\
\{M \rightarrow M_1\} \cup \text{ModuList}(M, D_{ModVarList})
\]

Next we define the algorithm ModuleMethods that takes a module definition list as an input and outputs a function that assigns to each module name $M$ the set of methods defined inside it.

It uses the algorithm SingModMethods to build the structure.

The algorithm SingModMethods takes as input a single module method definition and uses the algorithm MethodList to build the structure.
Algorithm 4: ModuleMethods

Input: The module definition list $D_{ModDefList}$.
Output: A function $\text{ModuleMethods}(D_{ModDefList})$ that assigns to each module name the set of methods defined within its body.

\[
\text{ModuleMethods}(\varepsilon) = \emptyset;
\text{ModuleMethods}(D_{ModDef}; D_{ModDefList}) = \text{ModuleMethods}(D_{ModDefList}) \cup \text{SingModMethods}(D_{ModDef})
\]

Algorithm 5: SingModMethods

Input: A single module definition $D_{ModDef}$.
Output: The entry it creates in the function MethodList.

\[
\text{SingModMethods}(\text{module } M \text{ is } D_{ValVarList} D_{ModVarList} D_{MethodList} D_{RuleList}) = \{M \rightarrow \text{MethodList}(D_{MethodList})\}
\]

The algorithm MethodList takes as input the method definition for a module and returns a list which represents which methods each modules contain. It further more contains information on which formal parameter is used in the method.

Algorithm 6: MethodList

Input: A list of method declaration $D_{MethodList}$
Output: The corresponding list of methods, $\text{MethodList}(D_{ModVarList})$

\[
\text{MethodList}(\varepsilon) = \emptyset;
\text{MethodList}((\text{method } me \text{ with } y \text{ is } S)D_{MethodList}) = \text{MethodList}(D_{MethodList}) \cup \{(me, y)\}
\]

The example below shows how the algorithms work.

Example D.1.1 We continue with our running example introduced in Example 3.1.2 and get:

\[
\text{module } M1 \text{ is }
\text{valvar } x := 0
\text{method } am \text{ with } y \text{ is } x := y
\text{module } Top \text{ is }
\text{valvar } z := 0
\text{modvar } m1 : M1
\text{rule } first \text{ is } (z < 5) :: (m1.am(z); z := z + 1)
\]
From the example, only looking at the module definition and the module variable definitions, we get

\[ D_{\text{ModDef}} = \text{module } M_1; \text{ module } Top \text{ is modvar } m_1 : M_1; , \]

Using the algorithm FullList on the module definition we get

\[
\begin{align*}
\text{FullList}(\text{module } M_1; \text{ module } Top \text{ is modvar } m_1 : M_1) \\
= \text{FullList}(\text{module } M_1) \cup \text{SingMod}(\text{module } Top \text{ is modvar } m_1 : M_1) \\
= \text{FullList}(\text{module } M_1) \cup \text{ModuList}(M, \text{modvar } m_1 : M_1) \\
= \text{FullList}(\text{module } M_1) \cup \{Top \rightarrow M_1\} \cup \text{ModuList}(M, \varepsilon) \\
= \text{FullList}(\text{module } M_1) \cup \{Top \rightarrow M_1\} \cup \emptyset \\
= \text{FullList}(\text{module } M_1) \cup \{Top \rightarrow M_1\}
\end{align*}
\]

We now use the algorithm ModuleMethods on the example above, we only view the part of the syntax which is relevant for the example

\[
\begin{align*}
\text{ModuleMethods}(\text{module } M_1; \text{ module } Top \text{ is method } me \text{ with } y) \\
= \text{ModuleMethods}(\text{module } M_1) \cup \text{SingModMethods}(Top \text{ is method } am \text{ with } y) \\
= \text{ModuleMethods}(\text{module } M_1) \cup \{Top \rightarrow \text{MethodList(method } am \text{ with } y)\} \\
= \text{ModuleMethods}(\text{module } M_1) \cup \{Top \rightarrow \{(am, y)\}\} \\
= \emptyset \cup \{Top \rightarrow \{(am, y)\}\} \\
= \{Top \rightarrow \{(am, y)\}\}
\end{align*}
\]

We can use these definitions to define functions Children and Methods that take a module name as an argument and return its children modules and its methods respectively.

**Function** Children

**Input:** A module name \( M \) and a module definition list \( D_{\text{ModDefList}} \).

**Output:** The set of children modules, of \( M \), in \( D_{\text{ModDefList}} \).

\[ \text{Children}(M, D_{\text{ModDefList}}) = \{M' | M \rightarrow M' \in \text{FullList}(D_{\text{ModDefList}})\} \]
**Function** Methods

**Input:** A module name $M$ and a module definition list $D_{ModDefList}$.

**Output:** The set of methods defined, in $M$, in $D_{ModDefList}$.

\[
\text{Methods}(M, D_{ModDefList}) = X \iff M \rightarrow X \in \text{ModuleMethods}(D_{ModDefList})
\]

Now we are prepared for defining when a model list declaration is well formed.

**Definition D.1.2** We say that a model list declaration $D$ is well formed if

- FullList($D$) is a DAG, and
  
  $D = \varepsilon$ or
  
  $D = \text{module } M \text{ is } D_{ValVarList} D_{ModVarList} D_{MethList} D_{RuleList}; D_{ModDef}$

- Where all the methods which are used in some module are defined within the DAG.

- We only define module variables described in a child module

- That all the variables which are referenced exist in the module.