Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Lorant Vrinceanu

Random-Based Model Generation for the Evaluation of Logic Synthesis Performance

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Lorant Vrinceanu

Random-Based Model Generation for the Evaluation of Logic Synthesis Performance

Bachelor Thesis based on the examination and study
regulations for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr.-Ing. Lutz Leutelt
Second examiner: Prof. Dr. Heike Neumann

Day of delivery: 2. November 2020

Lorant Vrinceanu

Title of the Bachelor Thesis

Random-Based Model Generation for the Evaluation of Logic Synthesis Performance

Keywords

HDL, VHDL, Model, FPGA, Synthesis

Abstract

This thesis proposes a method and implements a proof of concept that allows the generation of models of digital systems which can be used to evaluate the performance of FPGA synthesis algorithms. The models are based on a network of interconnected finite state machines with a data path (FSMDs) that are randomly generated. In order to allow the exploration of the space of possible systems in a structured manner, the generation process is controlled by a set of parameters which specify the specific subset to be explored. The generated models are then converted into a VHDL description and synthesized.

Lorant Vrinceanu

Thema der Bachelorarbeit

Zufallsbasierte Modellgenerierung für die Bewertung der Logiksyntheseleistung

Stichworte

Hardwarebeschreibungssprache, VHDL, Modell, FPGA, Synthese

Kurzzusammenfassung

Diese Arbeit schlägt eine Methode vor und implementiert einen Proof of Concept, der die Bewertung der Leistungsfähigkeit von Algorithmen zur FPGA-Synthese anhand automatisch generierter Modelle von digitalen Systemen ermöglicht. Die Modelle basieren auf einem Netzwerk von miteinander verbundenen endlichen Zustandsmaschinen mit Datenpfad (FSMDs), die nach einem Zufallsmodell generiert werden. Um die Untersuchung des Parameterraums möglicher Systeme auf strukturierte Weise zu ermöglichen, wird der Generierungsprozess durch eine Reihe von Parametern gesteuert, die eine Untersuchung einer spezifischen Teilmenge ermöglichen. Die generierten Modelle werden dann in eine VHDL-Beschreibung umgewandelt und synthetisiert.

Contents

List of Tables 6

List of Figures 7

1. Introduction 8
1.1. Outline 9

2. Fundamentals 10
2.1. Theoretical background 10
2.1.1. FPGAs 10
2.1.2. FPGA development flow 10
2.1.3. Design entry 11
2.1.4. FPGA synthesis 19
2.2. Related work 20

3. Requirements 21

4. Concept 23
4.1. Model generation 24
4.1.1. System model 24
4.1.2. ASMD model 27
4.1.3. State model 29

5. Implementation 33
5.1. Architecture and run flow 33
5.2. Parameters 34
5.3. Model generation 34
5.4. VHDL generation 36
5.5. Synthesis 37
5.6. Data storage 38

6. Conclusion 39
6.1. Discussion 39

References 41

Appendix A. Setup guide 43

Appendix B. Class diagrams 44

Appendix C. DVD contents 55

Nomenclature 57

List of Tables

2.1. Levels of abstraction considered in digital system development [6]. 12

4.1. System generation parameters. 27
4.2. ASMD generation parameters. 29
4.3. State generation parameters. 32

List of Figures

1.1. The synthesis process is analyzed as a black box by varying the input in a structured manner and comparing the results of the synthesis. 9

2.1. Typical FPGA development flow [6]. The dashed lines represent the process in case a verification step fails. 11
2.2. Block diagram of an FSM [6]. 14
2.3. Block diagram of an FSMD [6]. 15
2.4. ASMD block diagram with its elements indicated [6]. 18
2.5. ASMD block diagram with the formal definitions of an FSMD marked. 19

4.1. Overall concept of the environment to be developed as a flowchart. 23
4.2. Example of a system with its elements marked: 1 - inputs, 2 - outputs, 3 - ASMDs, 4 - interconnections, 5 - I/O bundles. 25
4.3. Block diagram of an ASMD model. 28
4.4. Structure of an ASMD state. 30
4.5. Binary expression tree of the expression $(r_1 - r_2) * 3 + 2$. 31

B.1. Class diagram of the concept model. 45
B.2. Class diagram of the operations model. 46
B.3. Class diagram of the generators. 47
B.4. Class diagram of the I/O model. 48
B.5. Class diagram of the parameter types. 49
B.6. Class diagram of the run parameters. 50
B.7. Class diagram of the system parameters. 51
B.8. Class diagram of the single run system. 52
B.9. Class diagram of the synthesis runner. 52
B.10. Class diagram of the VHDL generator. 53
B.11. Class diagram of helper classes. 54

1. Introduction

Digital hardware has revolutionized the world in the past decades. Over time, its capabilities have grown exponentially and it is today part of nearly every electronics, communications or control system we use—from a cell phone to a power plant.

Field programmable gate arrays (FPGAs) now contain the equivalent of millions of logic gates and tens of thousands of flip-flops [1]. At this scale, using traditional methods of logic design like drawing schematics becomes non-feasible. Nowadays, complex digital systems are designed by describing their structure and desired behavior using a hardware description language (HDL), like VHDL or Verilog. Computer-aided design software is used to simulate the design in order to verify the intended function of the system, and synthesize the design—a process that maps the high-level description onto actual hardware. Due to a high complexity, the synthesis process is split into several steps, where each step creates a description of the design at a lower level of abstraction by using various transformations and optimization algorithms along the way.

The purpose of this thesis is to create an environment that would allow to evaluate the performance of optimization techniques employed by synthesis tools. This is achieved by generating random models of digital systems. The models are structurally based on a network of interconnected finite state machines with a data path (FSMD). This structure was chosen due to the fact that the FSMD is a universal model that can describe any hardware design.

After the models of multiple systems are generated, they are converted into a VHDL description that can be synthesized. By varying parameters that impact specific areas of the model generation and comparing the results and performance metrics of the synthesis process across the generated systems, conclusions can be drawn regarding the operation of synthesis tools.

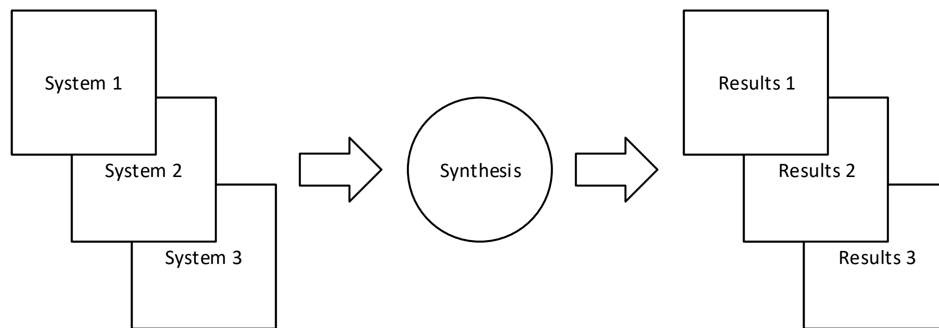


Figure 1.1.: The synthesis process is analyzed as a black box by varying the input in a structured manner and comparing the results of the synthesis.

1.1. Outline

The thesis structure in terms of chapters is described below:

Chapter 2 Fundamentals starts with an introduction of the theoretical background and ends with a review of related work.

Chapter 3 Requirements presents an overview of the requirements for the development of the model generation.

Chapter 4 Concept describes the proposed architecture of the model generation environment.

Chapter 5 Implementation transforms the conceptual description of the project into an implementation and describes the implementation details.

Chapter 6 Conclusion discusses the results of the thesis.

2. Fundamentals

2.1. Theoretical background

2.1.1. FPGAs

FPGAs are reconfigurable hardware chips that are used to implement digital logic functions. The first FPGAs were introduced by Xilinx in 1984 and have since increased in capacity by a factor of more than 10000 and in speed by a factor of 100 [2]. They are most efficient at tasks requiring parallelism and high throughput, but can be tailored to almost any application [3].

The basic structure of an FPGA is composed of the following elements:

- Logic blocks, called Configurable Logic Blocks (CLB) in case of Xilinx devices. These blocks implement the user logic using lookup tables, flip-flops and multiplexers [4]
- Programmable interconnects, which route signals and connect elements to one another
- Input/Output (I/O) pads
- Hard blocks, like embedded memories, digital signal processing (DSP) blocks and multipliers [1], [5].

2.1.2. FPGA development flow

Computer-aided design software is used to aid the designer in all the steps of FPGA-based hardware design. A typical development flow of a design targeting an FPGA is presented in Figure 2.1 and the steps are described below.

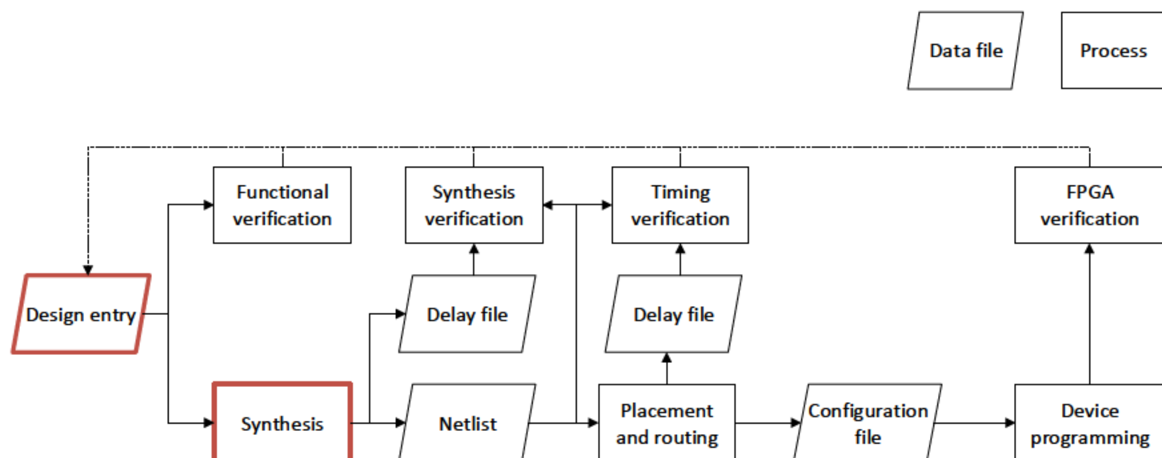


Figure 2.1.: Typical FPGA development flow [6]. The dashed lines represent the process in case a verification step fails.

1. **Design entry** The flow starts with a design file, which is a description of the circuit as a block diagram or HDL code.

2. **Synthesis** At the synthesis stage of the design flow, the provided circuit is mapped to a set of gates that perform the same desired functionality. The logic gates are then mapped into the logic cells of the chosen technology. The results of the synthesis is stored as a netlist.

3. **Placement and routing** At this stage, the data stored in the netlist is translated into specific locations on the target device and the routing between the logic elements is defined. The output of the place and route stage is a bitstream.

4. **Device programming** The bitstream generated on the previous step is used to program the device.

5. **Verification** At various stages of the flow, the design is checked in simulation whether it meets the specification and the performance and timing goals defined [6].

This thesis focuses on the design entry and synthesis steps of the development flow, therefore a more detailed overview of these steps are described in the following sections.

2.1.3. Design entry

At the design entry step, the system representation is created—a description which specifies the intent of the design. Designers use HDLs like Verilog and VHDL for describing the design.

These HDLs allow the specification of the design at different abstraction levels and from various views, which are presented in Table 2.1.

Table 2.1.: Levels of abstraction considered in digital system development [6].

Abstraction level	Behavioral view	Structural view	Physical view
Transistor	Differential equations	Transistor, resistor, capacitor	Transistor layout
Gate	Boolean equations	Gate, flip-flop	Cell layout
Register transfer (RT)	RT operation	Adder, register, multiplexer	Module floor plan
Processor	Algorithm	Processor, memory, I/O interface	IP floor plan

Views

The same system can be examined from different views, depending on the task at hand. There are three views used in digital design [6]:

- The behavioral view treats the system as a black box and focuses on describing the input-output characteristics of the circuit.
- The structural view describes the implementation of the circuit by specifying the components used and the connections between them.
- The physical view is the most detailed perspective which extends the structural view with physical details like the sizes and location of the components.

The synthesis task becomes easier when the amount of structural and physical details increase, but digital designers prefer describing the system from a behavioral view—this process being simpler and less time-consuming [7].

Levels of abstraction

As digital systems become more complicated, being able to describe a system in different levels of abstraction helps to manage complexity. An abstraction is a simplified representation of the system that focuses on critical information and removes unnecessary details.

In digital system development, there are four levels of abstraction considered [6]:

- At the transistor level, the circuit is treated as an analog system, with analog components like transistors and resistors being used for the structural description and their respective input-output characteristics are used in the behavioral view.
- At the gate level, the circuit is composed of digital building blocks like logic gates and can thus be described by boolean equations.
- At the register transfer (RT) level, modules constructed from gates are used in the structural description. These modules can be functional units, storage components and data routing units. Designers use extended finite state machines (FSMs) for the behavioral description of systems designed at this level.
- The processor-level abstraction uses intellectual properties (IPs) like processors and memory modules as structural building blocks and the behavior is typically described by an algorithm in a programming language.

The focus of this thesis is on generating circuits which are designed at and above the RT level, thus an overview of the models of computation used for the behavioral description at this level is presented below.

Finite State Machines

In the field of digital design, finite state machines (FSMs) are used to describe sequential circuits with a complex behaviour. The FSM is a model used to describe a system that transits through a finite number of states based on the state of its inputs and the state it is currently in. An FSM can be in a single state at a time. An FSM can have two types of outputs: Mealy and Moore. If the output is function of the state only, the output is known as a Moore output. However, if the output is a function of the state and the input signals, the output is known as a Mealy output [6].

The block diagram of an FSM is presented in Figure 2.2. The FSM in the diagram is a synchronous FSM, where the state transition is synchronized to a clock signal. The FSM has a state register which acts as a memory element and stores the state of the FSM. The next-state logic and the output logic implement the next-state function and the output function, respectively.

The behavior of an FSM can be described by state diagrams or an algorithmic state machine (ASM), which are out of the scope of this thesis, however an extended type of ASM, called an ASM with a data path (ASMD) will be described in the following section.

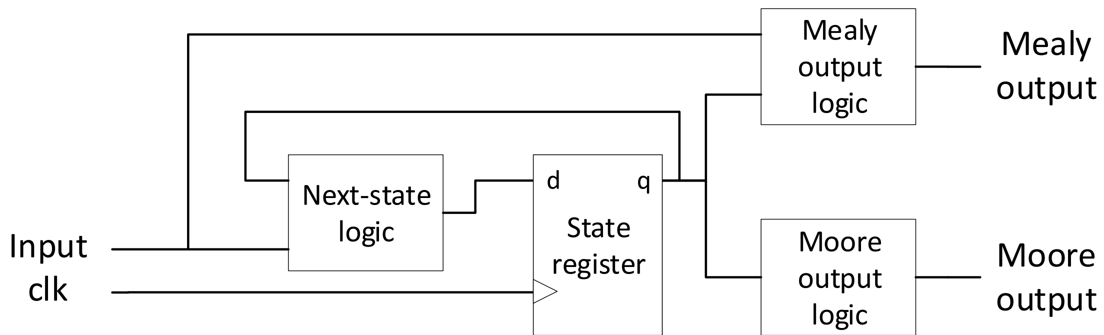


Figure 2.2.: Block diagram of an FSM [6].

Formally, an FSM is defined as a quintuple [7]

$$< S, I, O, f : S \times I \mapsto S, h : S \times I \mapsto O >$$

where

- S is a set of states,
- I is a set of input values,
- O is a set of output values,
- f is the next-state function, and
- h is the output function.

The FSM model can be applied for systems that have up to a few hundred states. Beyond this threshold, FSMs become too complex for human designers. This is due to the fact that FSMs do not allow the usage of variables, as a 16-bit integer variable would already represent 2^{16} or 65536 states [7].

FSMs with a data path and the RT methodology

To tailor the FSM model to more complicated systems, a set of variables and operations on them are introduced, leading to the concept of an FSM with a data path (FSMD). An FSMD is a universal model that can be used to represent all hardware designs [7] and is the cornerstone of the register transfer (RT) methodology.

The RT methodology allows the realization of an algorithm in hardware by providing hardware constructs that resemble the sequential execution method. The algorithm is transformed into a succession of register transfer operations that describe how the data is transformed and transferred between registers.

The characteristics of the RT methodology are [6]:

- The imitation of variables used in an algorithm by using registers which store intermediate data.
- The usage of a data path that realizes all the required register operations by using registers (source and destination) and combinational logic which is responsible for manipulating the data.
- The usage of a control path which specifies the order of the register operations and is realized by an FSM.

The structure of a FSMD is depicted in the block diagram in Figure 2.2.

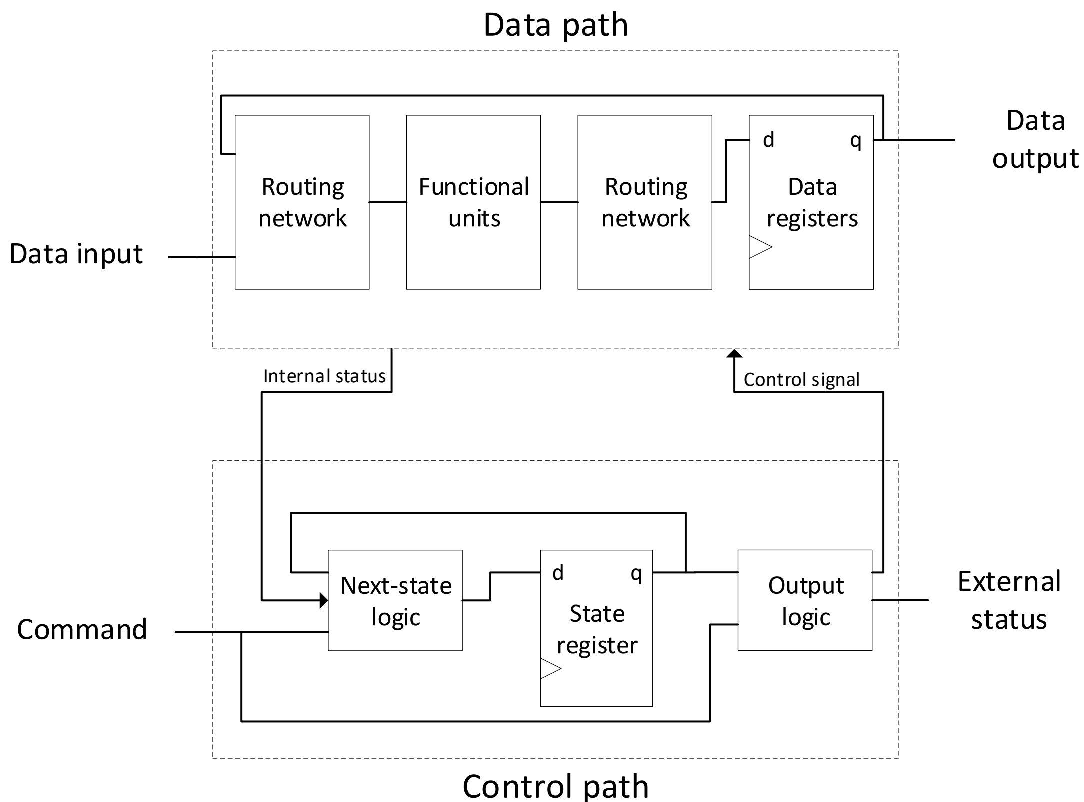


Figure 2.3.: Block diagram of an FSMD [6].

The operation at the base of the RT methodology is the register transfer operation that has the following notation [6]:

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, ..., r_{srcn})$$

The RT operation is comprised of the followings steps:

1. At the rising edge of the clock and after the clock-to-q delay of the source registers, the source registers contain new valid data.
2. The output of the $f(\cdot)$ function is computed by a combinational circuit and provided as an input for the destination register r_{dest} in time before the next rising clock edge.
3. At the next rising edge of the clock, the result is stored into the destination register r_{dest}.

Formal definition of FSMDs

The FSMD model and its formal definition that follows was introduced by Gajski and Ramachandran [7] as a universal model that can represent all digital systems.

Given the definitions below:

- A set of storage variables VAR,
- A set of expressions $EXP = \{f(x, y, z, ...)|x, y, z \in VAR\}$,
- A set of storage assignments $A = \{X \Leftarrow e|X \in VAR, e \in EXP\}$,
- A set of status signals as the logical relation between two expressions from the set EXP as $STAT = \{Rel(a, b)|a, b \in EXP\}$,

an FSMD is formally defined as the quintuple:

$$< S, I \times STAT, O \times A, f, h >$$

where

- S is a set of states,
- $I \times STAT$ is a set of input values extended to include status expressions,
- $O \times A$ is a set of output values extended to include storage assignments,
- $f : S \times (I \times STAT) \mapsto S$ is the next-state function, and

- $h : S \times (I \times STAT) \mapsto O \times A$ is the output function.

ASMD

In practice, the behavior of an FSMD is described with an algorithmic state machine with a data path (ASMD). An ASMD chart representation is a descriptive and readable way of completely describing the behavior of an FSMD that can be easily transformed to VHDL code. This representation will thus be extensively used for the description of models in this thesis.

An ASMD chart is comprised of a network of ASMD blocks. An ASM block has one state box and an optional network of decision boxes and conditional output boxes. Figure 2.4 presents the structure of a single ASMD block:

- The state box gives a symbolic name to the state and lists the Moore outputs as RT operations.
- The decision box tests an input condition and determines the exist path of the ASMD block.
- The conditional output box lists the Mealy outputs and can be only placed on the exit path of an ASMD chart.

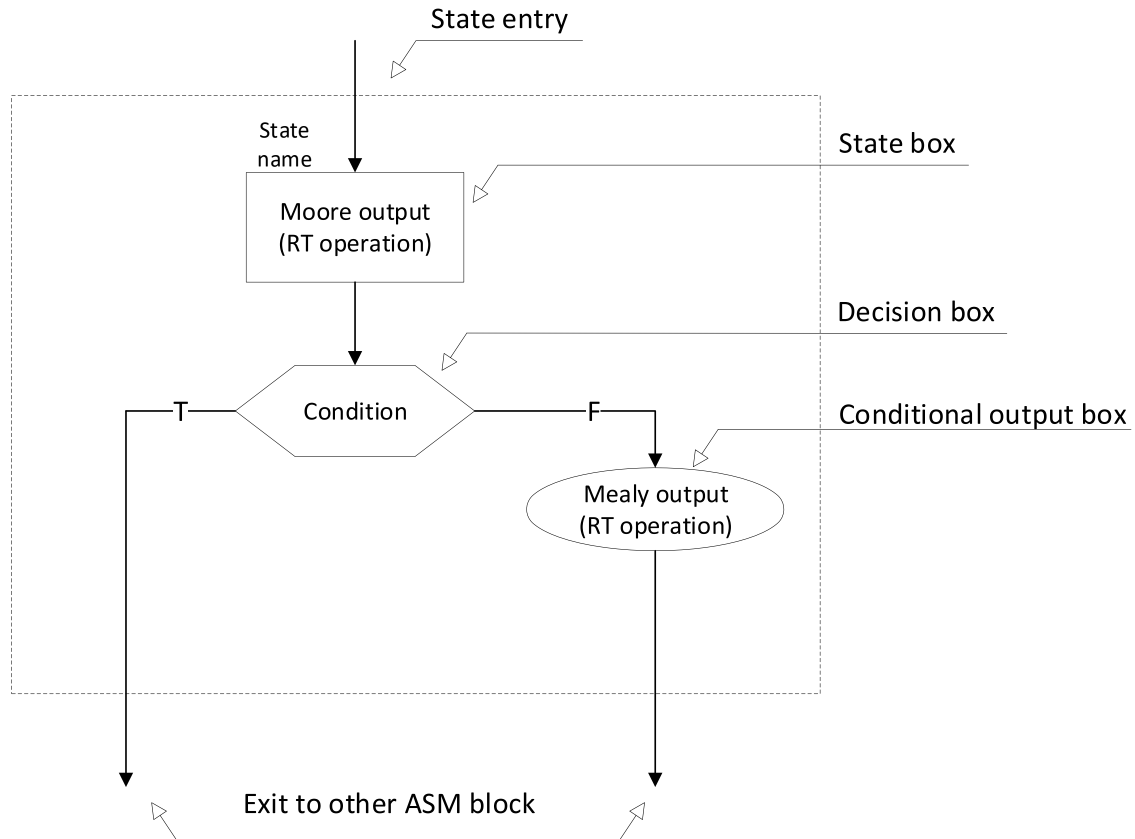


Figure 2.4.: ASMD block diagram with its elements indicated [6].

The relationship between the elements of the ASMD block and the formal definitions of an FSMD are depicted in Figure 2.5 and described as follows:

- RT operations are **storage assignments**.
- The right side of an RT operation is an **expression**.
- The left side of an RT operation is a **storage variable** or an output of the FSMD.
- The condition in a decision box is a **status signal**.

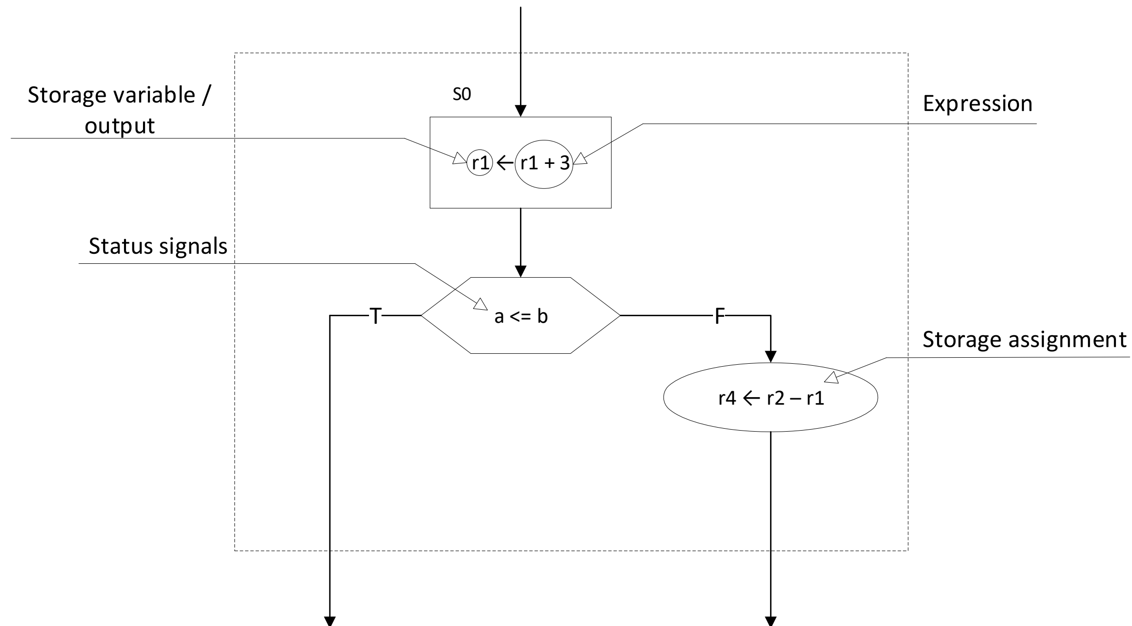


Figure 2.5.: ASMD block diagram with the formal definitions of an FSMD marked.

2.1.4. FPGA synthesis

Synthesis is a process that transforms the behavioral or structural view at a high level of abstraction of the designed system into a gate-level structural representation using the primitive cells available in the chosen device technology. To manage the complexity, the synthesis process is typically split into several steps, each performing a transformation which adds more detail to the design, thus generating a description in a lower level of abstraction.

- **High-level synthesis** transforms an algorithm into an RT-level description with an architecture consisting of a data path and a control path.
- **RT-level synthesis** transforms a behavioral RT-level representation into a structural implementation using components from a RT-level library. These components can be functional units which implement the operators encountered in the HDL code, routing units like multiplexers which implement the routing structure of the design, and storage units. At this level, optimization techniques like common code elimination and operator sharing can be applied to enhance the performance of the circuit.
- **Gate-level synthesis** or logic synthesis is the process of generating a structural implementation of the design using gate-level components, such as nor and nand gates.

The circuit can be optimized with respect to area (gate count), speed (propagation delay) or by obtaining an optimal area-delay trade-off.

- **Technology mapping** is the process by which the gate-level netlist is mapped onto the cells of the chosen device technology. This is the only step which is device technology dependent [6].

2.2. Related work

A number of research efforts have explored artificial benchmark generation due to a lack of publicly available benchmark circuits for testing synthesis tools. Benchmark concepts presented in [8]–[10] are targeted at testing logic synthesis processes.

Iwama and Hino [8] propose an approach to generate benchmark circuits that are functionally equivalent to real circuits by applying network transformation rules to an initial circuit. A concept for generating clones of a real circuit is proposed in [9], where the characteristics of a real circuit is extracted and clones generated from this circuit. A graph-based generation method is proposed in [10].

The FSMD formalization efforts by Gajski and Ramachandran [7] have been a starting point for the development of the concept presented in this thesis.

3. Requirements

The objective of this work is to propose and implement a method for generating benchmark circuits for testing the performance of synthesis tools. The concept shall be based on the generation of digital systems of random-based interconnected FSMDs that statistically reflect the properties of real-world digital designs. In order to be able to explore the space of all possible FSMDs in a structured way, every step or decision in the generation process shall include parameters, parameter ranges and probabilities of the parameter values.

The following subsections detail the sets of requirements that shall be fulfilled by the developed environment.

Model generation

- The environment shall be able to generate random-based interconnected FSMDs with respect to the chosen parameter space.
- An abstract model shall be provided for the representation of the generated circuit.
- A set of parameters that control the model generation shall be defined.
- The model shall be defined independently of the VHDL representation.
- The environment shall be able to generate RT operations.

Conversion to VHDL model

- The environment shall be able to parse the abstract model and generate VHDL code.
- The generated VHDL code shall be synthesizable.
- The generated VHDL code shall be human-readable.
- The generated VHDL code shall be designed according to common coding practices.
- The generated VHDL code shall structure the system hierarchically.

Synthesis of VHDL models

- The environment shall be able to control the software suite that synthesizes the circuit using console-based commands.
- The environment shall be able to run the synthesis process on the generated VHDL code and store the results.
- It shall be possible to run the synthesis process for multiple devices.

Software environment

- Matlab shall be used as the development platform. The reasons is its widespread usage in the academia, including at HAW Hamburg. This enables the sharing of knowledge and collaboration on the topic.
- The Vivado Design Suite by Xilinx shall be used for synthesizing the circuits.
- The implementation shall be modular and extensible.
- The generated systems and related data shall be stored in a structured way.
- The configuration of the environment shall be separated from the model generation.

4. Concept

Based on the requirements, a flowchart that represents the general concept of the environment to be developed is presented in Figure 4.1. There are three major sequential processes involved as part of a single run of the environment:

1. The model generation, which is the focus of this chapter, is the process which creates a model of a system based on a set of parameters that describe its various characteristics.
2. The VHDL generation process converts the model into its VHDL representation and can accept parameters that control the code generation.
3. The synthesis step of the flow is the interface to the software suite and controls the synthesis process via the available API. It is implementation specific and will thus be described in the next chapter.

This split of responsibilities allows to separate the creation of a model from its VHDL representation, the synthesis process and their respective parameters.

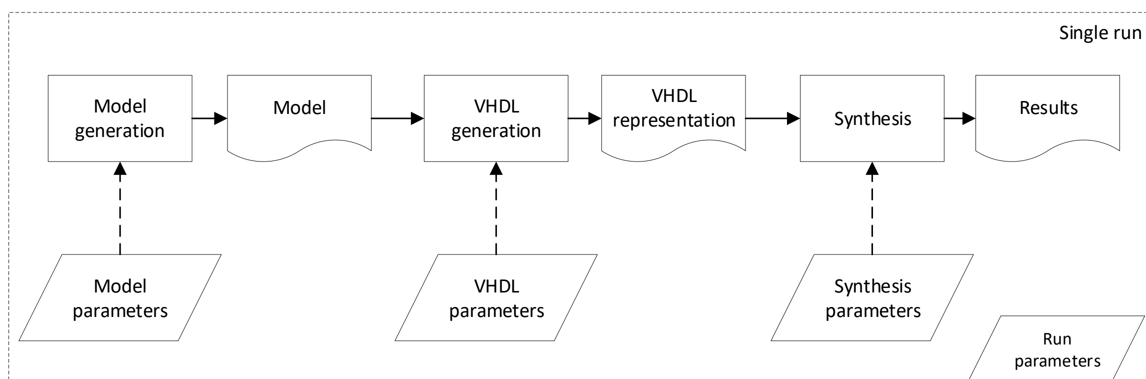


Figure 4.1.: Overall concept of the environment to be developed as a flowchart.

4.1. Model generation

This thesis proposes a model generation concept based on combining multiple ASMDs into a system. The concept is developed in a way that allows the parameterization of every step in the generation process, where randomness is added to the model by assigning random values to the control parameters.

The next sections present the model concept in a top-down approach.

4.1.1. System model

The top-level entity of the model generation concept is the system. An example of a system is presented in Figure 4.2. Every system is uniquely defined by

1. A set of inputs
2. A set of outputs
3. A set of ASMDs that implement the logic of the system
4. A set of interconnections (signals) between the I/O interfaces of the system and the separate ASMDs

The model generation process follows the order defined above for the generation of a system. First, the I/O interface is defined, second, the ASMDs are generated, and finally, the interconnections are created.

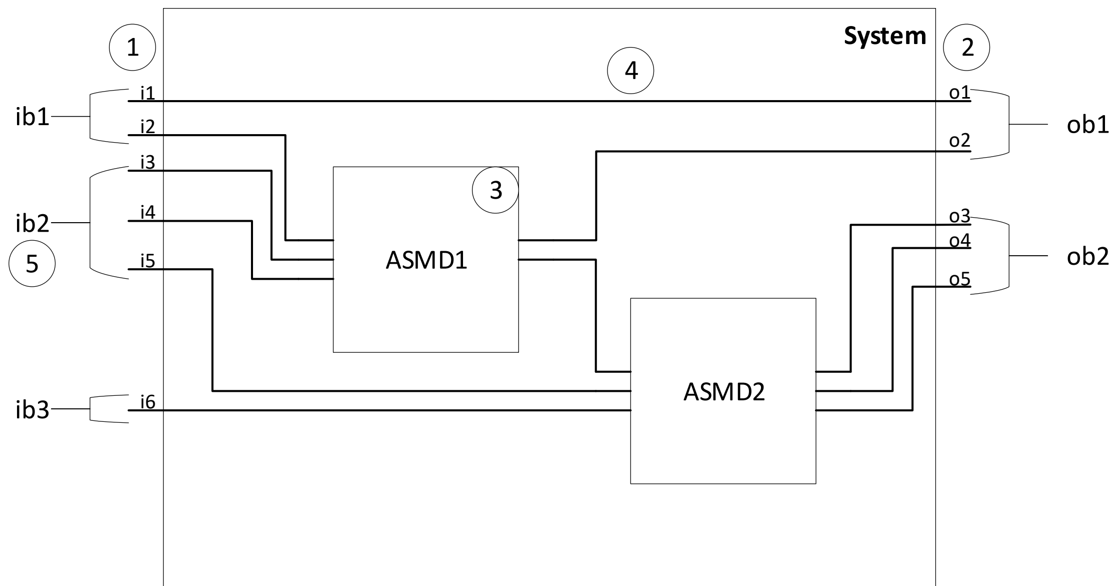


Figure 4.2.: Example of a system with its elements marked: 1 - inputs, 2 - outputs, 3 - ASMDs, 4 - interconnections, 5 - I/O bundles.

I/O interface of modules

During the generation of a system, the number of inputs and outputs a system can have is controlled by parameters which define ranges of possible sizes.

In order to emulate the structure of a bus, both the inputs and the outputs of the modules defined in the concept (system and ASMDs) can be grouped into bundles (Figure 4.2). The size of every bundle is randomly generated from a parameter that defines the range of possible sizes.

The rules for the creation of bundles are that each pin can be part of only one bundle and that a bundle can contain pins of a single type (input or output). The following two strategies for generating bundles are proposed:

Random For a bundle of a defined size, add to it a random pin that is not part of any other bundle, until the bundle is full.

Consecutive For a bundle of a defined size, add to it consecutive pins that are not part of any other bundle, until the bundle is full.

Interconnection generation

After the I/O structure of the system has been defined and the ASMDs generated, a strategy for connecting the I/O interfaces of the different modules has to be applied.

An interconnection in a system can be uniquely defined by the following elements:

- The module, the bundle of the module and the pins of bundle that drive the interconnection
- The module, the bundle of the module and the pins of the bundle that are on the receiving side of the interconnection

The following strategies can be used for the generation of interconnections:

Completely random The receivers and drivers of all modules can be connected at random. This can have the following four effects: drivers can remain unconnected, receivers can remain unconnected, a single driver can be connected to multiple receivers and multiple drivers can be connected to the same receivers. All these four effects are plausible for a real-world design, but this approach is not used.

Greedy random routing For every receiver in the system, a driver is randomly picked and connected to the receiver until all receivers are connected. This the strategy used in the proof of concept.

Bus A standardized bus interface like the Advanced eXtensible Interface (AXI) [11] can be applied

System generation parameters

The concept uses the four following types of parameters:

- The *value* parameter stores a specific value.
- The *range* parameter stores a range defined by a minimum and maximum value.
- The *probability* parameter stores the probability of a certain generation event happening.
- The *enumeration* parameter stores a set of named values.

Table 4.1 presents an overview of parameters used in the generation of a system.

Table 4.1.: System generation parameters.

Parameter name	Parameter type	Description
Input count	Range	Amount of inputs of a system
Input bundle size	Range	Amount of elements in an input bundle
Input bundle generation strategy	Enumeration	Strategy used for the generation of input bundles
Output count	Range	Amount of outputs of a system
Output bundle size	Range	Amount of elements in an output bundle
Output bundle generation strategy	Enumeration	Strategy used for the generation of output bundles
ASMD count	Range	Amount of ASMDs contained in a system
ASMD interconnection strategy	Enumeration	Strategy used for the generation of interconnection within a system

4.1.2. ASMD model

ASMDs are building blocks used in the system model. An ASMD is uniquely defined by the following elements, which can be identified in Figure 4.3:

- A set of inputs
- A set of outputs
- A set of storage variables, representing RT-level registers
- A state adjacency matrix, which stores the transitions between the states
- A set of states, modeled as ASMD blocks

The model generation process follows the order defined above for the generation of an ASMD. First, the I/O interface is defined, second, the storage variables are defined according to parameters that control their count and size, third, the state adjacency matrix is generated as described below, and finally the ASMD blocks of the states are generated.

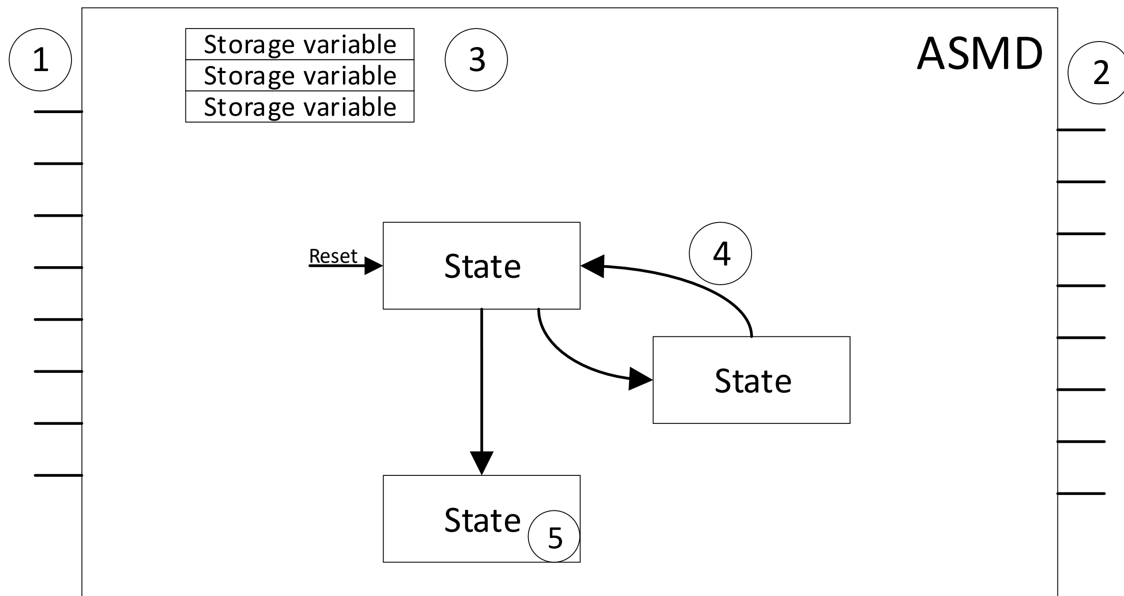


Figure 4.3.: Block diagram of an ASMD model.

Adjacency matrix generation strategy

An adjacency matrix is a square matrix, the elements of which indicate whether pairs of vertices are adjacent or not in a graph. A state adjacency matrix is thus created to indicate the transitions between states in a FSMD. The concept assumes that the first state in the adjacency matrix is the reset state.

A rule for creating ASMDs is that the exit path of an ASMD block must always lead to a state box, which means that there must be at least one transition from every state. Thus, the following strategies can be used for the generation of an adjacency matrix:

Complete interconnection strategy Every state has a transition to every other state, including itself. In this case, all elements of the adjacency matrix are 1.

Transition to itself and to next state Every state has a transition to itself and to the next state.

ASMD generation parameters

Table 4.2 presents an overview of parameters used in the generation of an ASMD.

Table 4.2.: ASMD generation parameters.

Parameter name	Parameter type	Description
Input count	Range	Amount of inputs of an ASMD
Input bundle size	Range	Amount of elements in an input bundle
Input bundle generation strategy	Enumeration	Strategy used for the generation of input bundles
Output count	Range	Amount of outputs of an ASMD
Output bundle size	Range	Amount of elements in an output bundle
Output bundle generation strategy	Enumeration	Strategy used for the generation of output bundles
Storage variable count	Range	Amount of storage variables of an ASMD
Storage variable size	Range	Size in bits of a storage variable
State count	Range	Amount of states of an ASMD
Adjacency matrix generation strategy	Enumeration	Strategy used for the creation of the adjacency matrix

4.1.3. State model

The model of an ASMD state is based on the ASMD block and is shown in Figure 4.4. Its elements are

- The state box, which stores the state name and the Moore assignments (RT operations) of the state
- The so-called decision tree, which models the next-state function of the underlying FSMD and stores the Mealy assignments of the state

The state model generation starts with the state box and ends with the generation of the decision tree. Part of this process is the generation of assignments, which is discussed next.

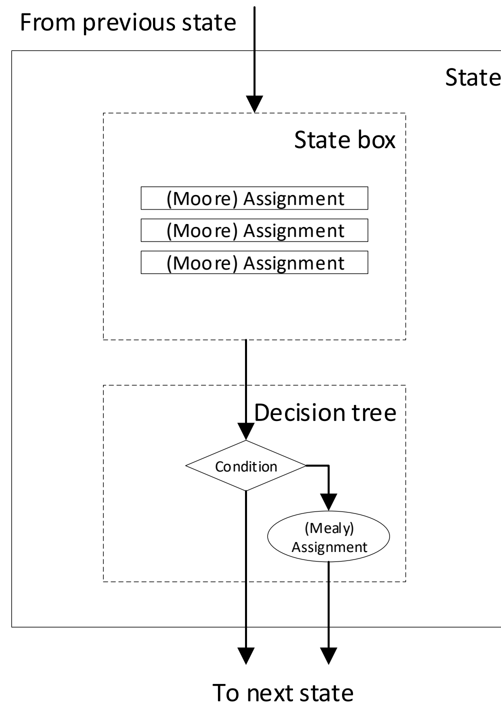


Figure 4.4.: Structure of an ASMD state.

Assignment generation

This section discusses the generation of Moore and Mealy assignments, and conditions (here called relations) as part of the decision tree generation.

As part of an ASMD block, an assignment has the form of a register transfer operation:

$$target \leftarrow expression$$

A few examples of register transfer operations are shown below [6]:

- $r \leftarrow 2$: The constant 2 is stored in the register r.
- $r \leftarrow r$: The content of the register r is stored back into itself.
- $r_1 \leftarrow r_2$: The content of the register r_2 is stored into the register r_1.
- $r \leftarrow r + 3$: The content of the register r is increased by 3 an stored back into itself.

A given expression can be built out of the following two elements:

- Operators, which are symbols indicating a mathematical operation.
- Operands, which are input values for the operators. In the case of an ASMD, there are two types of operands: storage variables (registers) and ASMD outputs.

Thus, the target of an assignment can be a single operand, and an expression can contain a combination of operators and operands. Relations, which are used as conditions in the decision tree, are generated the same way as expressions, but have a different set of operators available.

Two approaches for generating expressions and relations were analyzed:

- The first approach is to store an expression (and assignment) as text. This is the simplest solution for simple expressions (at most one operator), but becomes complex to generate and parse for expressions with more than one operator.
- The second approach is to store and generate expressions using binary expression trees. A binary expression tree is a kind of a binary tree that can be used to represent expressions and relations by storing operands as leaves of the binary tree, whereas the other nodes contain operators. An example of an expression tree for the expression $(r_1 - r_2) * 3 + 2$ is presented in Figure 4.5.

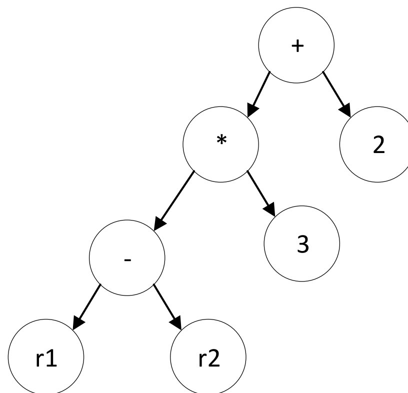


Figure 4.5.: Binary expression tree of the expression $(r_1 - r_2) * 3 + 2$.

The second approach was selected and implemented in the proof of concept due to its flexibility and scalability. There are two parameters defined that control the generation of the binary expression tree—the depth of the tree and the operators used. This way, the difference between an expression tree and a relation tree is only the operators used.

A list of operators that can be used in expressions and relations are listed below:

- Arithmetic operators: $+$ (addition), $-$ (subtraction), $*$ (multiplication), $abs(\cdot)$ (absolute value)
- Logical operators: and, or, $nand$, nor, xor, $xnor$, not
- Shift operators: $>>$ (right shift), $<<$ (left shift)
- Relational operators, which are only used in conditions (relations): $=$ (equals), $/=$ (not equals), $<$ (less than), $>$ (greater than)

Decision tree generation

The same binary tree approach is used to represent the decision tree, where nodes of the tree can be relations, mealy outputs and transitions. The type of node can then be parsed by the VHDL generator and relevant code is generated.

State generation parameters

Table 4.3 presents an overview of parameters used in the generation of a state and the generation of expressions and relations. These parameters allow for a flexible adaptation of the expression and relation binary trees.

Table 4.3.: State generation parameters.

Category	**Parameter name**	**Parameter type**	**Description**
Decision stage parameter	Mealy output probability	Probability	Probability of a decision stage to contain a mealy output
Expression parameter	Self assignment probability	Probability	Probability of generating a self-assignment expression
Expression and relation parameter	Expression or relation tree depth	Range	Depth of the binary expression trees
Expression and relation parameter	Constant operand probability	Probability	Probability of a constant to be used as operand
Expression and relation parameter	Constant operand range	Range	The allowed range of values for constant operands
Expression and relation parameter	Allowed operators	List	List of allowed operators

5. Implementation

The implementation stage of the thesis consist of the development of a proof of concept (POC) that takes the conceptual view and translates it into an implementation. The aim of the POC is to verify that the concept proposed is feasible and can have practical potential.

An environment is implemented that allows to run the entire flow proposed as a concept (Figure 4.1). A single run of the environment is defined by the sets of parameters used for the model generation, the VHDL generation and the synthesis process. During a run of the environment, multiple systems can be generated and a set of results for the specified parameters is obtained. Due to the intrinsic randomness of the model generation, no two runs with the same set of parameters will generate the same system, except if the same seed is used to initialize the pseudorandom number generator used.

In line with the requirements, Matlab is used for the development process. The way the concept was defined, an object-oriented (OO) implementation is the most suitable approach. The reason for this is that the concept is split into entities that contain data (which define their state) and have methods associated for the transformation of this data. Additionally, the OO approach adds a level of flexibility to the implementation, where different generation strategies or structures can be used simply by changing the class to be used.

5.1. Architecture and run flow

The starting point for the development of the architecture of the application is the concept proposed. The entities defined in the concept were mapped onto classes and then implementation-specific helper classes were created to implement the logic of the application. While the class diagram of the entire application can be found in Appendix B, separate implementation details are discussed in the following sections.

The top-level class of the application is the `SingleRun`, which is responsible for running the application flow and storing the generated data. Algorithm 1 is used for running the flow once.

Algorithm 1 Single run algorithm

Create and define the run parameters
Instantiate a `SingleRun` container
Generate all systems
Generate the VHDL code for every system
Run synthesis on every system model
Print run report
Save run to permanent storage

5.2. Parameters

An extensive system of parameters was defined in the Chapter 4. These parameters were mapped onto classes, as presented in Figures B.5 and B.7.

Additionally, implementation-specific parameter classes were introduced for the VHDL generation process and the synthesis run. These parameters are predominantly constant strings that define the names that should be used in the VHDL generation process and an overview is available in Figure B.6.

5.3. Model generation

The model class diagram was created based on the objects introduced in the concept and is presented in Figures B.1 and B.2.

In order to be able to generate the entities of the model in a flexible way, a system of generators was implemented as shown in Figure B.3. This system is based on an idea similar to the builder pattern [12], where a generator class is instantiated with the parameters it requires to be able to generate complex objects of a specific type without using constructors with a large amount of parameters.

Algorithm 2 shows the flow used for generating the system model.

Algorithm 2 System generation algorithm

Instantiate a `SystemGenerator` with a set of system parameters
for all systems to be generated **do**
  Generate system:
    Randomly generate the properties of the system using the given parameters
    Generate the system input pins
    Generate the system input bundles using an `IOBundleGenerator`
    Generate the system output pins
    Generate the system output bundles using an `IOBundleGenerator`
    Instantiate an `AsmdGenerator` with a set of ASMD parameters
    for all ASMDs to be generated **do**
      Generate ASMD
    Generate the system interconnections using an `InterconnectionGenerator`

Algorithm 3 describes in detail the flow used for the generation of an ASMD model.

Algorithm 3 ASMD generation algorithm

Generate ASMD:
  Randomly generate the properties of the ASMD using the given parameters
  Generate the ASMD input pins
  Generate the ASMD input bundles using an `IOBundleGenerator`
  Generate the ASMD output pins
  Generate the ASMD output bundles using an `IOBundleGenerator`
  Create ASMD storage variables
  Generate the state adjacency matrix using an `AdjacencyMatrixGenerator`
  Instantiate an `StateGenerator` with a set of State parameters
  for all states to be generated **do**
    Generate state

Algorithm 4 shows the flow used for the generation of a state model.

Algorithm 4 State generation algorithm

Generate state:
  Create a `StateBox`
  for all targets of assignments **do**
    Generate an (Moore) assignment using an `AssignmentGenerator`
    Store the assignment in the `StateBox`
  Generate a state decision stage:
    for all transitions from the state **do**
      Recursively generate a binary decision tree
      that contains conditions, (Mealy) assignments and transitions

Expression tree generation

Algorithm 4 mentions the generation of Moore assignments, Mealy assignments and conditions (relations). The generation process is conceptually similar between the three entities, where for a defined depth of the binary expression tree, internal nodes of the binary tree are recursively created by randomly picking an operator and leaf nodes are created by randomly picking an available operand.

5.4. VHDL generation

The purpose of the VHDL generation step is to transform the model of the system into synthesizable VHDL code. For the POC, the two-segment VHDL description model with Mealy output support as defined in [6] is used for the transformation of the ASMD models. The two-segment model combines all combinational logic of the data and control path into one process and all the registers of both paths are merged into another process.

This task is completed by the `VhdlGenerator` class (Figure B.10) by parsing the system model in a top-down manner. Algorithm 5 describes this process.

The parameters used for the VHDL generation are stored in the `VhdlParameters` class. These parameters define the variable names and data types to be used by the `VhdlGenerator` class.

In order to create a VHDL representation of the binary expression and relation trees, the `VhdlGenerator` uses an inorder traversal of the trees.

Algorithm 5 VHDL generation algorithm

Parse ASMDs:
    Generate ASMD library declarations
    Generate ASMD entity declaration:
        for all asmd ports **do**
            Generate port declaration
    Generate ASMD architecture:
        Generate state types
        Declare state registers
        Declare storage variables
        Generate process for the state and data registers
        Generate process for the combinational circuit:
            for all states **do**
                Parse moore box
                Parse decision stages
Parse system:
    Generate system library declarations
    Generate system entity declaration:
        for all system ports **do**
            Generate port declaration
    Generate system architecture:
        Declare all components (ASMDs)
        Declare interconnection signals
        Instantiante all ASMDs
        Parse all interconnections and generate signal assignments

5.5. Synthesis

After a VHDL file of a system is available, the synthesis process can be started. This thesis uses the Vivado Design Suite from Xilinx, which is a computer-aided design software used for the synthesis and analysis of HDL designs. The `XC7A100TCSG324-3` Artix-7 FPGA is used as the target device. This device is used as an example and can be easily changed in the synthesis parameters.

The task of running the synthesis is completed by the `SynthesisRunner` class (Figure B.9) and the respective parameters are stored in the `SynthesisParameters` class (Fig-

ure B.6). These parameters are constant strings that are used for the generation of synthesis scripts.

In order to control the Vivado Design Suite from a shell environment using Matlab, Vivado must be run in non-project mode [13]. In non-project mode, tcl commands can be run using the Vivado Design Suite Tcl shell. Therefore, the `SynthesisRunner` first generates a tcl script that contains commands for the design software and then runs the script via a batch file, that also has to be generated. The order of execution of the synthesis step is presented in Algorithm 6.

Algorithm 6 Synthesis algorithm

Create a directory for the system to be synthesized
Save the VHDL file to the directory
Create a directory for the Vivado project
Generate tcl script, which control the Vivado Suite:
  Define the project output directory
  Setup the design sources and constraints
  Run synthesis and generate default utilization report
  Run implementation
Generate batch script, which runs the tcl script
Run batch script

5.6. Data storage

At the end of the run, the generated data is stored in the `Runs` directory in the project. For every run, a directory is created. In the run directory, for every system that is part of the run, a separate directory is created, where the following data is stored:

- The generated VHDL file
- The Vivado project and the generated scripts
- The run object as a `run.mat` file. This file contains all the data related to the run and can be explored using the Matlab Variable window.

6. Conclusion

Based on the initial idea and the developed requirements, this thesis proposes a concept for the generation of random-based models aimed at evaluating the performance of logic synthesis algorithms.

The concept describes how to generate ASMDs and connect them as part of a larger system. Different questions were raised during the concept development phase and the answers have become rules and generation strategies for various parts of the model. For example, binary trees are used as tools to describe elements of an ASMD.

An extensive set of parameters was introduced, which allows to randomize and control the generation process, thus analyzing a specific subset of the space of possible digital systems.

The proof of concept was successfully implemented and now allows the user to specify a set of parameters and run the entire generation and synthesis process, thus managing to achieve the goals set by the requirements.

6.1. Discussion

The next step for this project would be to use the developed environment to evaluate the performance of synthesis algorithms. The POC is already in a state where it can answer questions regarding the performance of synthesis algorithms, like

- What is the used area per state with a growing model size?
- What is the synthesis time per state?
- How does the size of the design depend on the depth of expression trees?

A necessary next step would be to extend the functionality of the synthesis runner with an automated collection of performance indicators, e.g., synthesis time, number of RT elements used, cell usage to be able to automate the evaluation process.

Further possible improvements would be

- to augment the concept with the usage of IP modules
- to include a formal description of the model in terms of structure, the space of parameters, their range and probabilities
- to implement additional strategies for the generation of interconnections in the system
- to implement a graphical user interface (GUI) for modifying the parameters used
- to implement additional VHDL generators which use different description models
- to include parameters for the VHDL generator that would allow it to use different conditional statements, like `case` and `if-then-else` statements.

References

[1] *Xilinx 7 Series FPGAs Data Sheet: Overview (DS180)*. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (visited on 2020-09-10).

[2] S. M. Trimberger, „Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology“, *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, 2015-03, Conference Name: Proceedings of the IEEE, ISSN: 1558-2256. DOI: 10.1109/JPROC.2015.2392104.

[3] R. Mueller, J. Teubner, and G. Alonso, „Data processing on FPGAs“, *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 910–921, 2009-08-01, ISSN: 2150-8097. DOI: 10.14778/1687627.1687730. [Online]. Available: https://doi.org/10.14778/1687627.1687730.

[4] *7 Series FPGAs Configurable Logic Block User Guide (UG474)*. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf (visited on 2020-09-10).

[5] *SDAccel Environment Profiling and Optimization Guide (UG1207)*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1207-sdaccel-optimization-guide.pdf (visited on 2020-08-19).

[6] P. Chu, *RTL hardware design using VHDL : coding for efficiency, portability, and scalability*. Hoboken, N.J: Wiley-Interscience, 2006, ISBN: 9780471786412.

[7] D. D. Gajski and L. Ramachandran, „Introduction to high-level synthesis“, *IEEE Design Test of Computers*, vol. 11, no. 4, pp. 44–54, 1994, ISSN: 0740-7475. DOI: 10.1109/54.329454.

[8] K. Iwama and K. Hino, „Random Generation of Test Instances for Logic Optimizers“, in *31st Design Automation Conference*, ISSN: 0738-100X, 1994-06, pp. 430–434. DOI: 10.1145/196244.196452.

[9] M. D. Hutton, J. Rose, J. P. Grossman, and D. G. Corneil, „Characterization and parameterized generation of synthetic combinational benchmark circuits“, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, pp. 985–996, 1998-10, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/43.728919.

[10] D. Stroobandt, P. Verplaetse, and J. v. Campenhout, „Generating synthetic benchmark circuits for evaluating CAD tools“, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 9, pp. 1011–1022, 2000-09, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/43.863641.

[11] *AXI Reference Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf (visited on 2020-09-20).

[12] *Design Patterns: Builder*. [Online]. Available: https://refactoring.guru/design-patterns/builder (visited on 2020-10-12).

[13] *Vivado Design Suite User Guide: Design Flows Overview*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug892-vivado-design-flows-overview.pdf (visited on 2020-10-15).

[14] *Vivado Design Suite User Guide: Synthesis*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf (visited on 2020-09-10).

Appendix A.

Setup guide

The project requires the following software to be installed on the machine:

- Matlab R2018a (or above) with the Statistics and Machine Learning Toolbox
- Vivado 2019.1 (or above)

To start the project,

1. Copy the project directory to a local disk.
2. Open the directory in Matlab.
3. Run `startup.m` to initialize the project.
4. Modify the runParameters.synthesisParameters.vivadoSettingsFilePath to the Vivado installation location
5. Adjust the run parameters in `main.m` and start the script to run the complete flow.

Appendix B.

Class diagrams

This section of the appendices contains all the class diagrams of the implementation. The class diagrams are split as follows:

Model: General The class structure of the model introduced in the concept (Figure B.1)

Model: Operations The class diagram of the operations model (Figure B.2)

Model: Generators The class diagram of the generators (Figure B.3)

Model: IO The class diagram of the I/O model of the modules (Figure B.4)

Parameters: Types The class diagram of the types of parameters defined: range, value, probability and enumerations (Figure B.5)

Parameters: Run view The class diagram of the parameters used in a single run (Figure B.6)

Parameters: System view The class diagram of the parameters used in the generation of a system (Figure B.7)

Other: Run The class diagram of the single run system (Figure B.8)

Other: Synthesis The class diagram of the synthesis runner (Figure B.9)

Other: VHDL The class diagram of the VHDL generator (Figure B.10)

Other: Helpers The class diagram of helper classes (Figure B.11). These classes include a binary tree data structure, a string builder for building the VHDL code and a progress bar class to display the progress of the run.

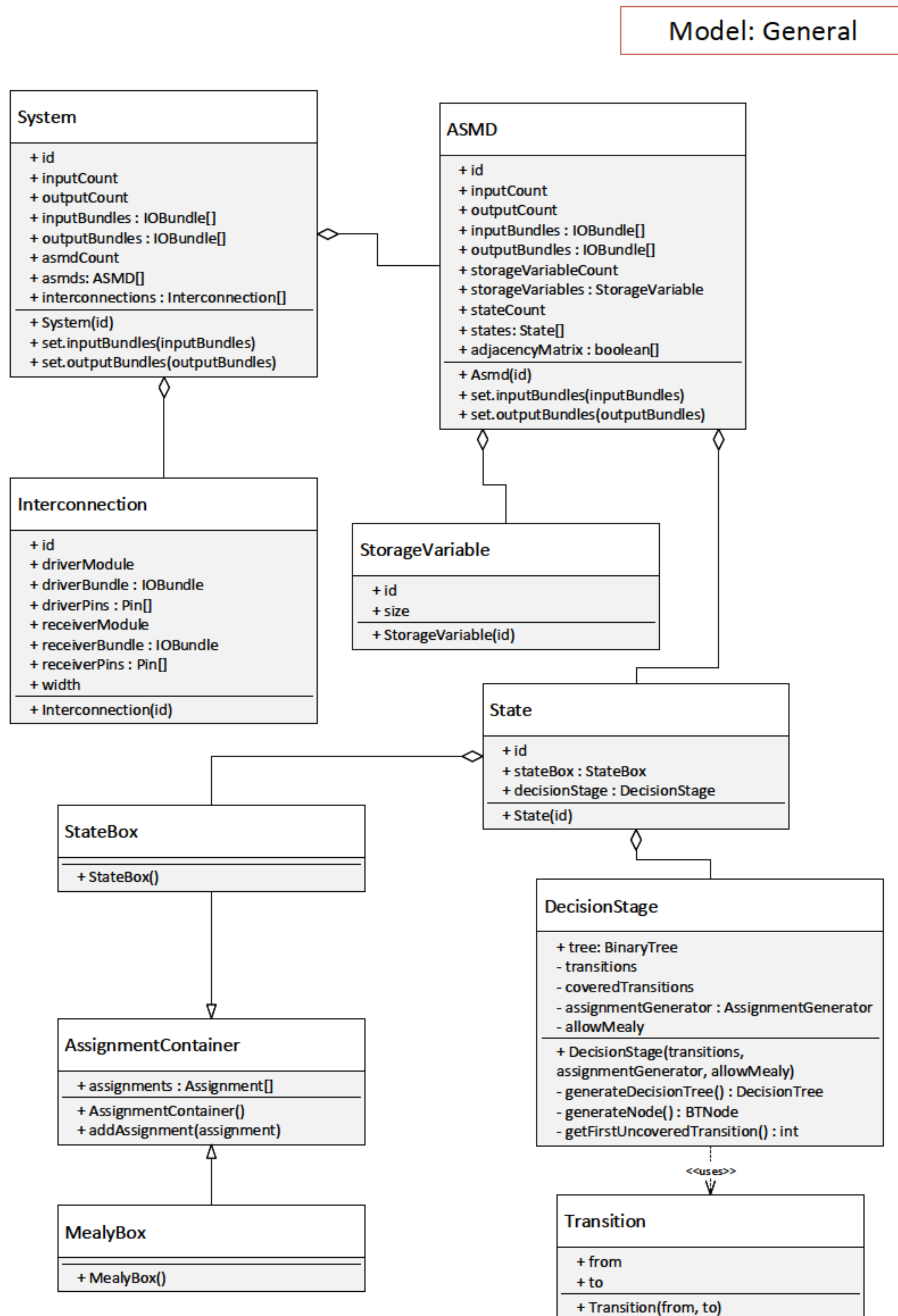


Figure B.1.: Class diagram of the concept model.

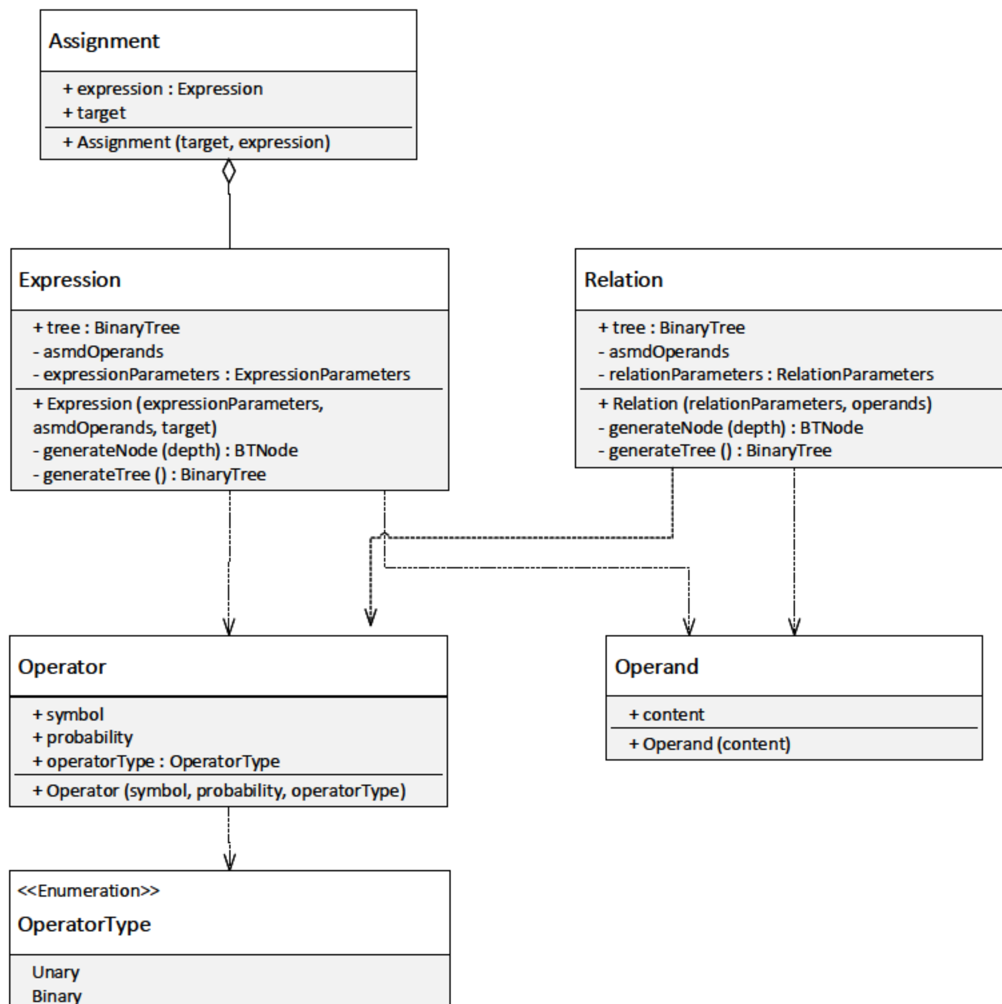


Figure B.2.: Class diagram of the operations model.

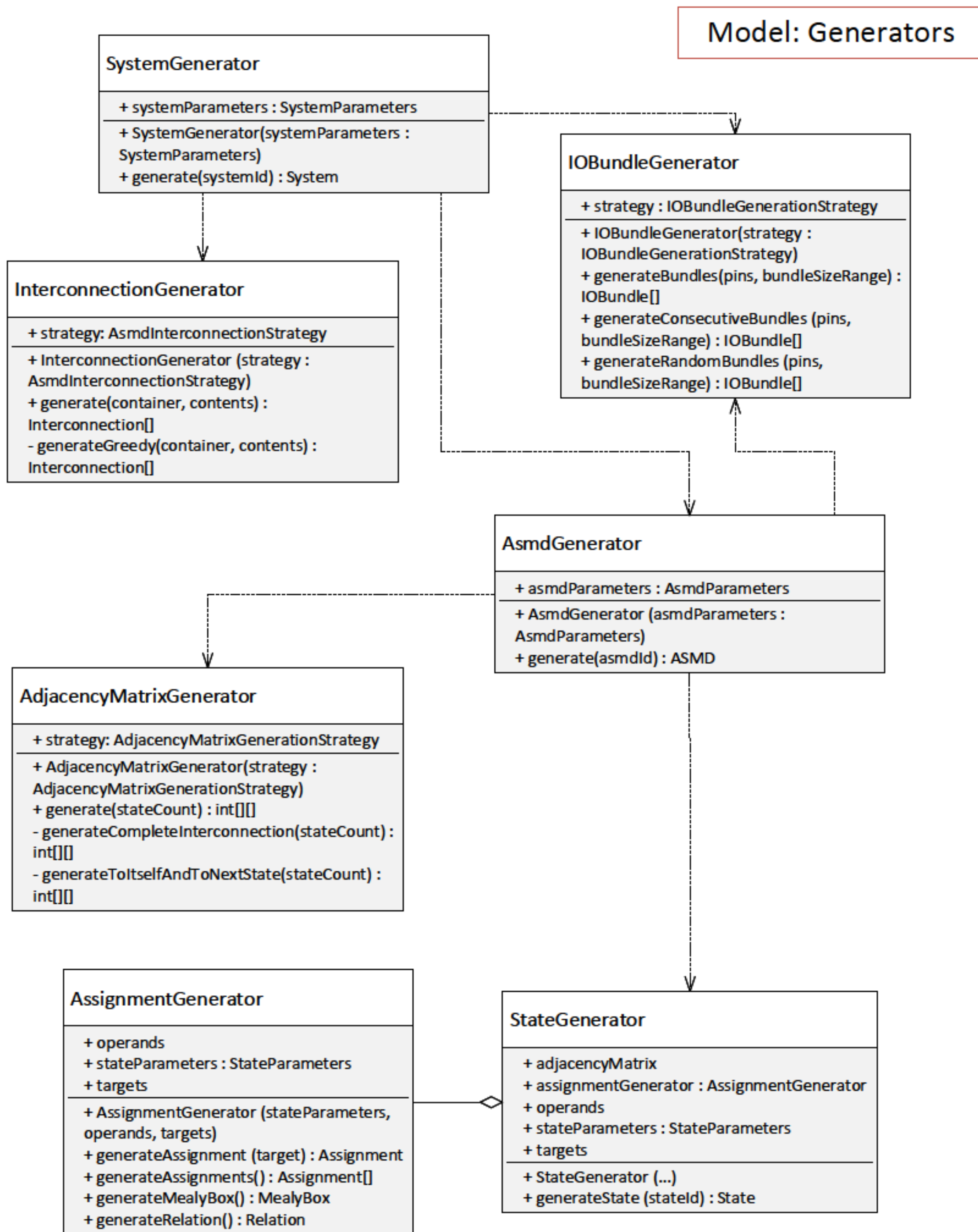


Figure B.3.: Class diagram of the generators.

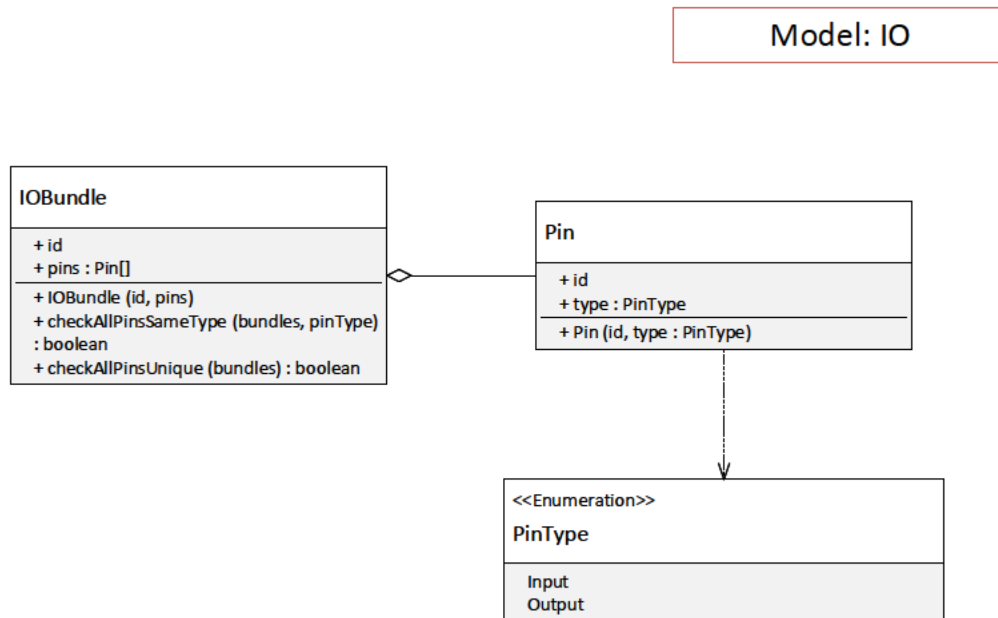


Figure B.4.: Class diagram of the I/O model.

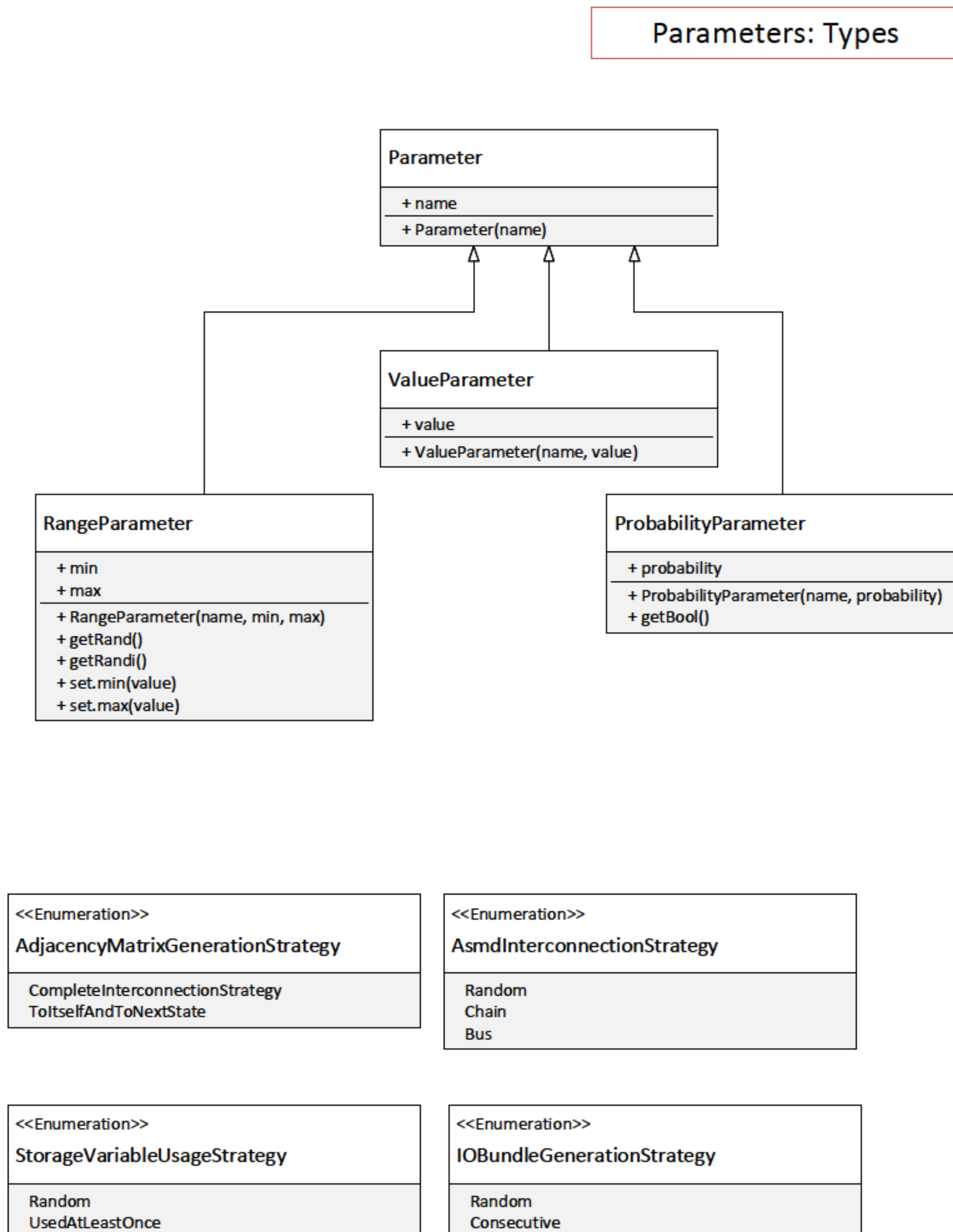


Figure B.5.: Class diagram of the parameter types.

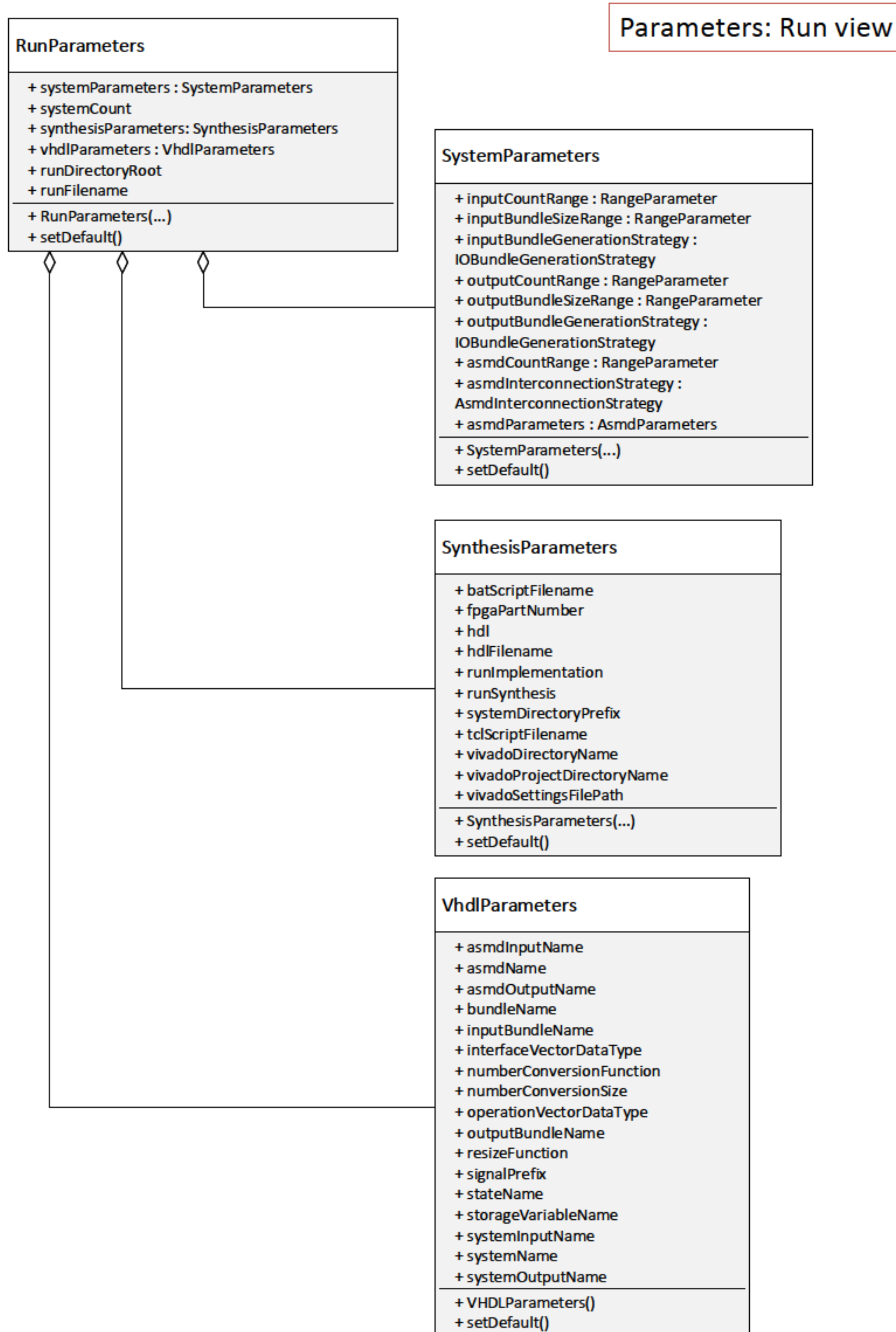


Figure B.6.: Class diagram of the run parameters.

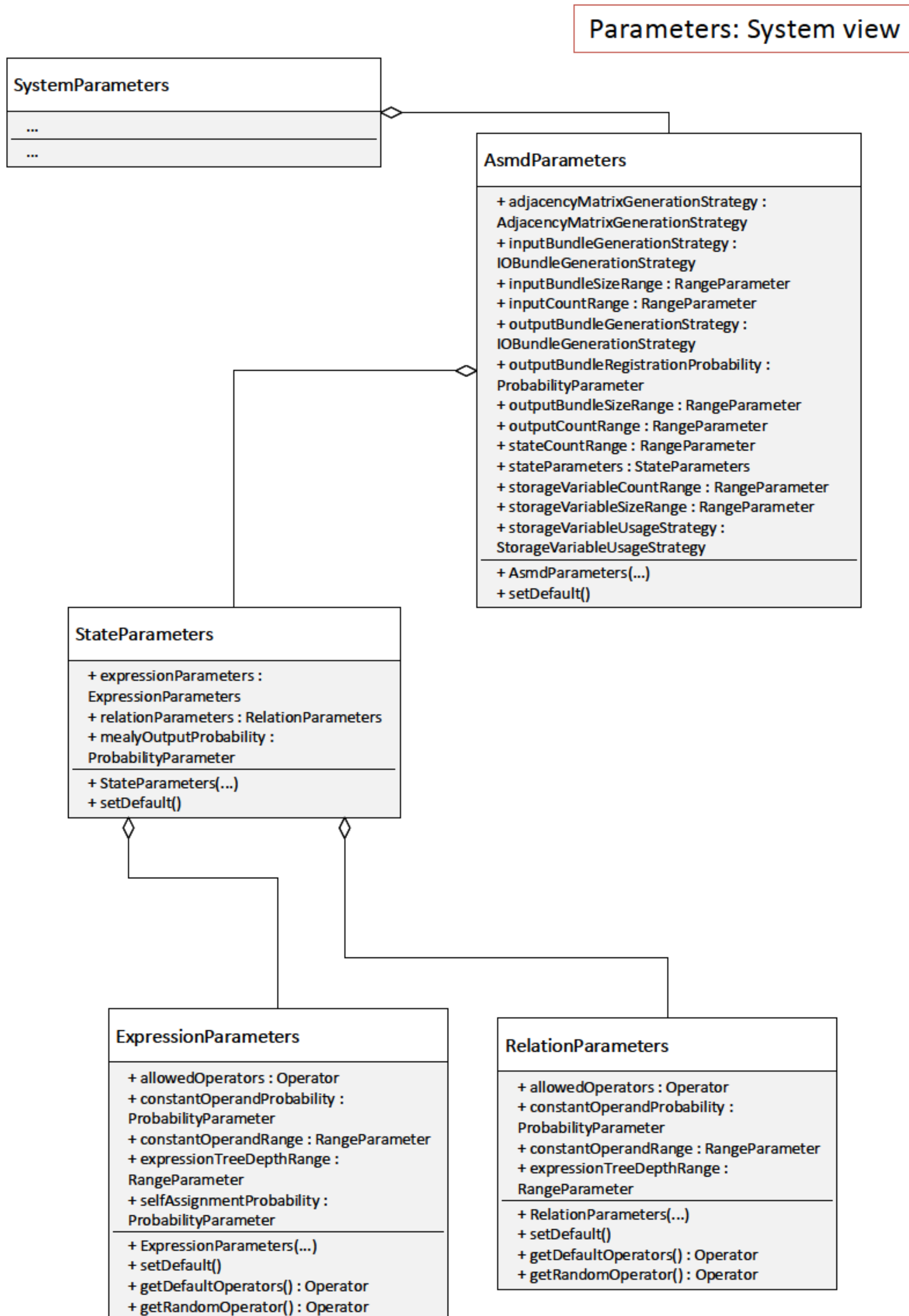


Figure B.7.: Class diagram of the system parameters.

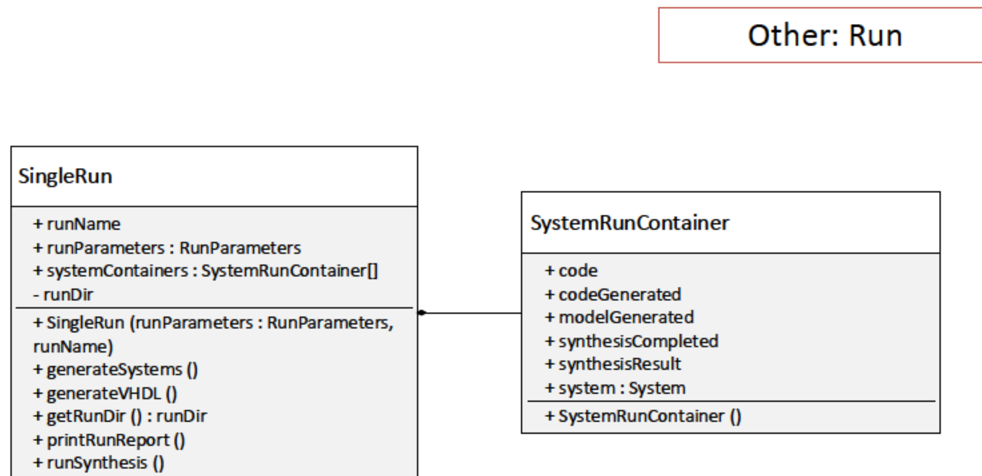


Figure B.8.: Class diagram of the single run system.

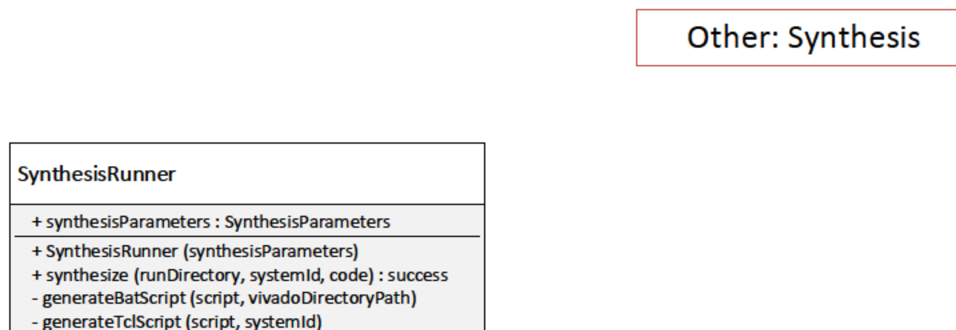


Figure B.9.: Class diagram of the synthesis runner.

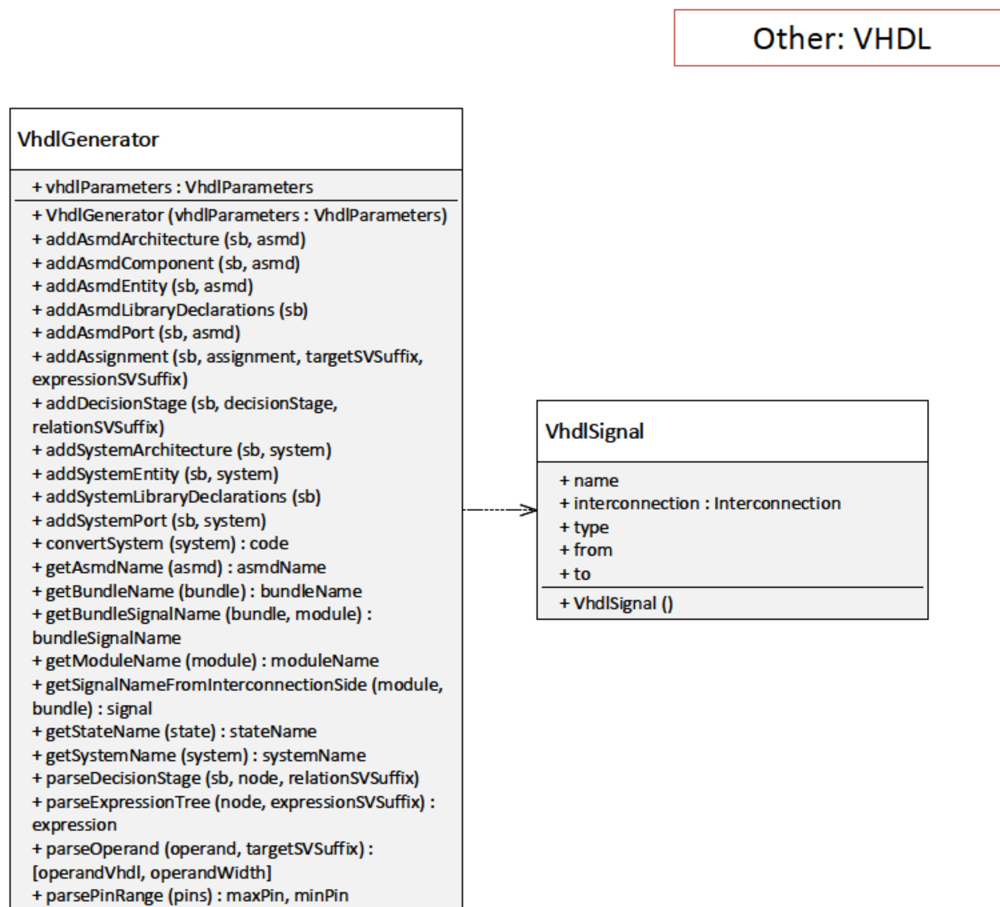


Figure B.10.: Class diagram of the VHDL generator.

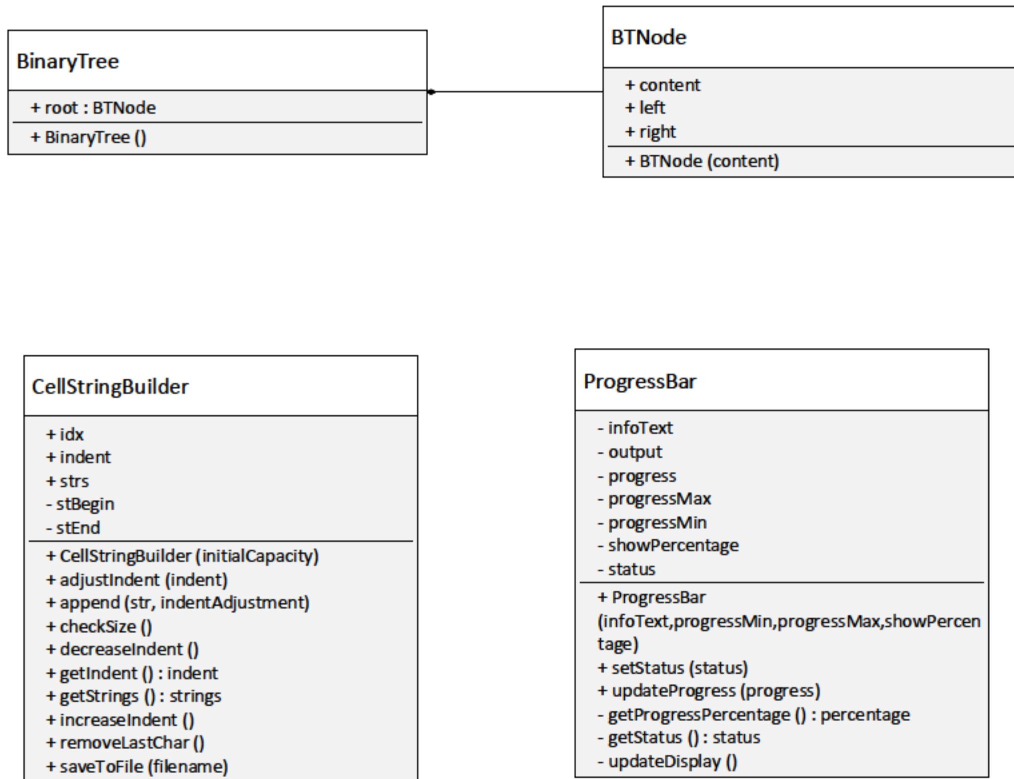


Figure B.11.: Class diagram of helper classes.

Appendix C.

DVD contents

This bachelor thesis comes packaged with additional files that are stored on a DVD. The directory tree below describes the delivery package.

```
DVD root
├── Bachelor Thesis Lorant Vrinceanu.pdf ............ This document
└── SBE
    ├── main.m .............................. The starting point of the application
    ├── startup.m
    ├── updatePath.m
    ├── Runs .......... Output of the application: generated code and Vivado projects
    │   └── example_run .......... An example run containing 3 generated systems
    └── Sources ...................................... Source code of the project
        ├── 2_Parameters
        │   └── ...
        ├── 3_Model
        │   ├── ...
        │   ├── Generators
        │   │   └── ...
        │   ├── Helpers
        │   │   └── ...
        │   ├── IO
        │   │   └── ...
        │   └── Operations
        │       └── ...
        ├── 4_HDL
        │   └── ...
        ├── 5_Synthesis
        │   └── ...
        ├── 6_Run
        │   └── ...
        └── 9_Helpers
            ├── DataStructures
            │   └── ...
            ├── ProgressBar
            │   └── ...
            └── StringOperations
                └── ...
```

Nomenclature

ASM Algorithmic State Machine

ASMD Algorithmic State Machine with a Data path

CLB Configurable Logic Blocks

DSP Digital Signal Processing

FPGA Field Programmable Gate Arrays

FSM Finite State Machine

FSMD Finite State Machine with Data path

GUI Graphical User Interface

HDL Hardware Description Language

I/O Input/Output

IP Intellectual Property

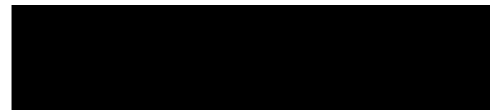
OO Object-oriented

POC Proof of Concept

RT Register Transfer

Declaration

I declare within the meaning of part 16(5) of the General Examination and Study Regulations for Bachelor and Master Study Degree Programmes at the Faculty of Engineering and Computer Science and the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.



Hamburg, November 2, 2020	
City, Date	Signature