Cell Libraries and Verification

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 2 november 2011 om 16.00 uur

door

Matthias Raffelsieper

geboren te Recklinghausen, Duitsland

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. H. Zantema
en
prof.dr.ir. J.F. Groote

Copromotor:
dr. M.R. Mousavi

IPA dissertation series 2011-15

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-2799-1

Contents

Preface **iii**

1 Introduction **1**

2 Introduction to Cell Libraries **7**
- 2.1 Different Views of a Cell 7
- 2.2 Transistor Netlist 8
- 2.3 Verilog Simulation Description 11

3 Equivalence Checking in Cell Libraries **15**
- 3.1 Semantics of VeriCell 16
- 3.2 Encoding VeriCell into Boolean Transition Systems 26
- 3.3 Equivalence Checking VeriCell and Transistor Netlist Descriptions 29
- 3.4 Experimental Results 30
- 3.5 Summary 31

4 Efficient Analysis of Non-Determinism in Cell Libraries **33**
- 4.1 Order-Independence of VeriCell Descriptions 34
- 4.2 Order-Independence of Transistor Netlists 44
- 4.3 Using Non-Determinism to Reduce Power Consumption 61
- 4.4 Summary 71

5 Relating Functional and Timing Behavior **75**
- 5.1 Timing Checks 76
- 5.2 Module Paths 83
- 5.3 Summary 92

6 Productivity Analysis by Context-Sensitive Termination **95**
- 6.1 Term Rewriting, Specifications, and Productivity 96
- 6.2 Productivity of Orthogonal Specifications 103
- 6.3 Productivity of Non-Orthogonal Specifications 113
- 6.4 Proving Productivity of Hardware Cells 119
- 6.5 Summary 123

7 Productivity Analysis by Outermost Termination **125**
- 7.1 Proving Productivity by Balanced Outermost Termination 125
- 7.2 Transformational Outermost Termination Analysis 131
- 7.3 Summary 149

8 Conclusion **151**

Bibliography **155**

Summary **163**

Curriculum Vitae **165**

A Nangate Open Cell Library License **167**

Preface

While finishing my diploma thesis at the RWTH Aachen, I was approached by my then supervisor Jürgen Giesl whether I would be interested in a PhD position in Eindhoven. Thanks to one the regular TeReSe meetings, that in Aachen were dubbed the "Dutch term rewriting community meetings" (including Aken), I was able to meet in Eindhoven with my future supervisors Hans Zantema and Jan Friso Groote, and also with Chris Strolenberg who would be involved in the ValiChip project. Due to the friendly atmosphere and the interesting ValiChip project, that would combine theory with practical applications from the hardware domain, I accepted the position and moved to Eindhoven.

In the beginning, it took me some time to become acquainted with the hardware domain, in particular cell libraries, that was to be the topic of my research. However, both Chris Strolenberg and Jan-Willem Roorda of the industrial partner Fenix Design Automation gave me good pointers, for which I am very thankful. Furthermore, I am grateful for their expert input during the rest of the ValiChip project, which led to the research questions treated in this thesis. It was quite sad that the economic crisis did not spare Fenix Design Automation, so that it had to go out of business in 2009.

I would also like to thank my daily supervisors and (co-)promotors Hans Zantema and MohammadReza Mousavi, who both gave me good advice and helpful technical feedback. Especially, I appreciated their different backgrounds which complemented each other well and that I could always walk into their offices with any problems I had. It was a pleasure working together with both Hans and Mohammad, and resulted in numerous co-authored publications which form the basis of this thesis.

My second promotor was Jan Friso Groote, head of the OAS group and the later MDSE section. He entrusted my daily supervision with Hans and Mohammad, which I believe was a wise choice. I enjoyed working in his group, and profited from the discussions I had with him.

The constructive comments and suggestions I received from the members of my thesis committee, Twan Basten, Hubert Garavel, Jürgen Giesl, Jan Friso Groote, MohammadReza Mousavi, and Hans Zantema, helped to improve the text substantially. I am therefore indebted to the aforementioned people for carefully reviewing my manuscript. Also, I would like to thank Wan Fokkink for agreeing to serve as an opponent in the defense.

During the last four years, I enjoyed meeting numerous people, mostly as members of the MDSE section, on joint projects, at conferences, and through the IPA research school. I really appreciated the inspiring conversations I had with them, which sometimes also went beyond just technical discussions.

Finally, I want to thank my family and friends. I feel deep gratitude towards you for supporting me both before and throughout my PhD period. Thank you.

Matthias Raffelsieper, September 2011

Chapter 1

Introduction

Electronic devices have become ubiquitous in modern life. This trend is expected to continue, with devices becoming even faster and even smaller at the same time. Such a development would not have been possible without the tremendous advances in the implementation of digital logic. Moore's Law [Moo65, Sch97], which states that the transistor count of integrated circuits doubles approximately every 18 months, continues to explain these trends, despite having been declared expired a number of times. The most prominent force driving Moore's law is technology scaling; smaller structures consume less power and operate faster. Furthermore, as more transistors are available, there is more opportunity to parallelize computations, thereby leading to another performance improvement.

Due to the increase in transistor count, the functionality implemented in a specific piece of hardware become larger and more complex. Thus, techniques are sought to verify hardware correct, as bugs are inevitable once a certain size or complexity of the hardware design has been reached. One approach to guarantee correct behavior is by testing. However, testing can only investigate a limited number of test cases, thus there can still be untested corner cases in which the device does not function properly. Formal verification is another approach that is not limited to pre-conceived test cases. Instead, it tries to mathematically prove that a model of the device behaves as expected under all circumstances. This approach of formally verifying correctness is the technique used in this thesis to guarantee correctness of hardware devices. Guaranteeing correctness is especially important for hardware, since more and more safety-critical applications make use of it, where a failure can be devastating.

Computer hardware is nowadays mainly designed in a top-down fashion: First, high level descriptions are created which are iteratively refined into a gate-level description (consisting only of simple logic functions and registers) and finally into transistors. This last refinement step, going from gates to transistors, is often performed using cell libraries, which are a collection of basic building blocks, called *cells*, that are both described at a higher abstraction level and at the level of transistors. Hence, it is vital for the correct functioning of a hardware design that these libraries are correct and always describe the same behavior. The main focus of this thesis is to present methods that formally prove different cell descriptions to be consistent with each other. In this way, a hardware design that works correctly at a higher description level is guaranteed to still be correct when implemented. Additionally, some techniques are presented to analyze other aspects of cells, such as power consumption and stabilization, i.e., that always stable output values are computed.

Consistency Checking in Cell Libraries

The first technique presented in this thesis verifies that both the simulation level description of cells and the transistor implementations correspond to each other. If this were not the case then a device that seemed to be implemented correctly in simulation runs could fail when implemented as hardware. To faithfully represent both descriptions, the simulation description and the transistor netlist, it is imperative that all possible behaviors that could occur are also present in the model created for them. Otherwise, the model is not general enough and might therefore hide inconsistent behaviors. In the considered cell libraries, a source of non-determinism, i.e., the possibility of multiple behaviors, is a race between inputs that are changed simultaneously. Due to physical effects, such changes are not perfectly synchronized, but arrive in some random order. Such non-determinism is undesired as the final result of a computation cannot be predicted anymore. Most simulators for Verilog [IEE06], a commonly used hardware description language, only implement one fixed order of considering inputs to maximize performance. This however is not backed by the behavior of real hardware; thus it might lead to a mismatch between the simulation and the real behavior.

Faithfully modeling all possible behaviors of hardware descriptions is addressed in this thesis by two methods. First, a formal semantics for a subset of Verilog, large enough to cover most cells from industrial cell libraries, is presented that allows for all possible behaviors. If such a model can be verified to always correspond to the transistor netlist implementation, then also any simulation will do so, since it is contained in the possible behaviors modeled. Implementing and checking such a semantics is however usually very inefficient. Thus, the method only implements one possible behavior. To guarantee that this restriction is not hiding allowed behavior, this thesis presents an efficient technique to analyze non-determinism of cell libraries, both for simulation and transistor level descriptions. If it can be established that the behavior is the same for all possible input orders, then the modeled behaviors and the actually possible behaviors are the same.

Energy consumption of chips has become a very prominent issue lately for numerous reasons. The high integration level of hardware is causing thermic problems if too much heat is produced, which could possibly destroy the chip. Also, mobile applications are ever growing, which are powered by batteries. In these mobile devices, long battery life is desirable, however the weight of the batteries is also limited. Therefore, in mobile applications, energy consumption is a key factor. To analyze the power consumed by a cell, the cell is usually triggered with a large number of input sequences (all input sequences, if possible) in every possible state of the cell and its power consumption is being measured. This thesis proposes a way to reduce the number of required power characterizations by making use of the non-determinism analysis. To this end, the analysis is extended to also consider the number of wires charged when some inputs are changed. Then, only those situations have to be measured where these chargings differ, i.e., for situations resulting in the same power consumption, only one representative has to be considered. A related technique does not determine equal power consumption, but instead the computation that consumes the minimal amount of power without affecting functionality. By enforcing this computation, one can therefore save power without affecting the cell's externally visible behavior. This again makes use of the extended non-determinism analysis, and selects among functionally equivalent computations the one consuming the minimal amount of power.

Hardware descriptions are not only required to be correct from a functional perspective, but also timing must be verified. In case a cell does not satisfy certain timing assumptions, such as independence of the arrival order of certain inputs or stability of outputs, then it cannot be used in designs requiring precisely these assumptions. Timing descriptions are often decoupled from the functional description, however timing and functioning are of course highly related. This link is taken into account in this thesis. It is observed that non-determinism in cells, as already introduced above, has a relation to the constraints on stability windows of its inputs, called the *setup* and *hold* times in hardware description languages such as Verilog. These constraints on the inputs of a cell rule out certain sources of non-determinism, thus a cell, which can behave non-deterministically, becomes deterministic when these constraints are satisfied. Therefore, the analysis of non-determinism takes timing information into account. On the contrary, if a cell is found to be non-deterministic even when considering the timing information, then it might be the case that the cell has not been characterized sufficiently, i.e., that additional timing restrictions have to be imposed.

Another requirement on a chip design is that it meets certain desired performance goals. Because hardware designs are mostly synchronous (with respect to a clock signal), all computations have to be finished before the next cycle. Otherwise, old or intermediate values would be used, thereby invalidating the functionality. From a cell level perspective, it is therefore interesting to check how long a change at some input signal takes until all output signals of the cell have reached their final response value. Using this information, one can then go up the design hierarchy to approximate the performance of the complete chip design using the timing information of the cells. For this purpose, each cell is accompanied by a number of *module paths*, which are paths from input to output signals associated with delays. However, as in the case of setup and hold times, these module paths are not linked to the functional description of the cell, i.e., they could possibly not reflect the actual behavior of the cell. In this thesis, a method is presented to check whether both are describing the same cell, by requiring that the module paths and the functional description correspond to each other. This is done on the one hand by verifying that a specified module path exists in the cell, i.e., that a change in the specified input can have an effect on the value of the specified output. If this were not the case, then the module path would describe unobservable behavior, and thus overconstrain the design of the whole chip. In the worst case, a chip design could be found not to meet the desired performance goals due to such a *false path*. On the other hand, one wants to make sure that for every possible path through a cell timing information is specified, since otherwise no delay is considered and therefore the performance is overapproximated. To this end, a technique is described that enumerates all possible paths through a cell where a changing input can have an effect on the value of an output. Therefore, delays have to be determined for these paths. This allows to ensure that all possible delay behaviors have been considered.

Hardware and Streams

To compute output values from some input values, hardware implemented as transistor netlist is often employing feedback loops, where the output of some function is also being used as its input. This way, for example, on-chip memories (such as flip-flops) are implemented. Thus, it is also interesting to study whether such computations will eventually produce a stable output value, or whether they keep looping forever. This question is, on an abstract level, the same as that of *productivity of streams*. There, one is given with a system to compute a stream, i.e., an infinite sequence of (output) values, and is interested whether each of its elements is eventually stable. Hardware can be viewed as a stream function, which has a number of input streams from which it then computes a number of output streams. In this thesis, productivity of streams is studied on a very abstract level, namely that of *Term Rewrite Systems*. This is a very basic, but powerful technique to describe computations by a number of rules. In this setting, productivity has already been studied before. Some of the previous work abstracted away the concrete values or were restricted to deterministic specifications only. This is different in techniques for checking productivity presented in this thesis, which especially allow arbitrary Boolean streams. Thereby, productivity analysis can be used to prove stabilization of hardware circuits for arbitrary sequences of input values. Removing the approximations however comes with a price, in this case the question whether a given specification is productive or not becomes undecidable. Despite this negative theoretical result, there are numerous systems for which an answer can be given. This is based on the advances in termination analysis, for which nowadays powerful automatic tools are available. Thus, the approach is to transform the productivity question into a termination question, so that existing tools can be employed.

Structure of the Thesis

This thesis first gives a brief introduction into the functional descriptions encountered in cell libraries in Chapter 2. As stated above, these will be the main focus of this thesis. Chapter 3, which is based on [RRM09], then presents a technique, together with an implementation and an evaluation thereof, to verify that both simulation description and transistor netlist description of a hardware cell describe the same functional behavior. There, it is observed that cells can behave non-deterministically. This is the topic of investigation in Chapter 4, which is based on [RMR$^+$09, RMZ10, RMZ11, RM11]. There, techniques are presented that identify non-determinism that can lead to different functional behavior, and techniques to identify functionally equivalent behavior that differs in power consumption. All of these techniques have been implemented and have been evaluated on industrial cell libraries. In Chapter 5 some non-functional descriptions are considered, namely timing checks and module paths. As already discussed above, they do have a connection to the functional descriptions, in that they also describe functional behavior. The chapter presents techniques to verify that the functional behavior described by these non-functional descriptions is consistent with the simulation description. Also these techniques were implemented and evaluated for industrial cell libraries. They were previously described in [RMR$^+$09, RMZ10, RMZ11, RMS10].

Productivity is studied in Chapters 6 and 7, which, as already mentioned above, can be used to prove stabilization of hardware circuits. In Chapter 6, which presents work from [ZR10a, Raf11], productivity is proven using context-sensitive termination analysis. The approach is also applicable to non-orthogonal specifications, which are natural when checking stabilization of hardware. For both orthogonal and non-orthogonal specifications a tool has been developed that tries to automatically prove productivity. An example application of the latter tool is presented in Section 6.4, where it is proven that an implementation of a scanable D flip-flop taken from an industrial cell library always computes a stable next state.

The technique presented in Chapter 7 was previously presented in [RZ09, ZR10b] and relies on outermost termination to prove productivity. For that purpose, it also presents a transformation from outermost termination problems into standard termination problems, such that state-of-the-art termination provers can be used. This transformation has been implemented in a tool which participated in the outermost category of the annual termination competition 2008 [Wal09] (see [MZ07] for more details on the termination competition), proving some examples outermost terminating for which no other tool was able to do so.

Finally, the thesis is concluded in Chapter 8, where also some possible topics for future work are discussed.

Chapter 2

Introduction to Cell Libraries

A *cell library* is a collection of different combinatorial and sequential elements, called *cells*, that are used to realize larger chip designs. Examples of combinatorial elements are logic functions such as an `and` gate or an `xor` gate, whereas sequential elements, such as a flip-flop, provide some kind of memory. Ultimately, a cell is described as a number of lithographic masks, that are read by a wafer stepper to produce the physical implementation of the final chip. Hence, this description is the most important from a manufacturer's point-of-view. However, for a designer working at a higher abstraction level this description is useless, as it does not allow to evaluate the function of the designed chip. Thus, cells are also described from a functional perspective to allow simulation of the final design.

A cell library is typically used by compilers that take as input a higher-level description of a chip design and create a netlist description containing numerous instances of the available cells. Cell libraries are usually provided by external sources and are specifically designed and optimized for a single production technique. To simplify the layout of cells in a chip, all cells have the same height and their power supply connections are at the top and at the bottom. Thereby, cells can be aligned in rows and their power supplies can be connected easily.

In order to produce a final chip design, multiple constraints have to be met, such as timing, total area of the silicon required to implement the chip, constraints regarding power consumption, and more. Furthermore, the chip design should of course implement the desired functionality. To check whether these requirements are satisfied for a concrete chip design, different kinds of information are needed. For example, *timing closure*, the process of repeatedly altering a chip design until it meets its timing constraints, needs detailed information about the timing behavior of the chip's components, i.e., the cells. In contrast, the verification that the logic is correct does not need any timing information; instead the precise computation is interesting. Such pieces of information, relevant to certain parts of the chip design process, are stored in so called *views* in the cell library.

2.1 Different Views of a Cell

For a single cell, a cell library contains numerous different descriptions of it. In this thesis, these are divided into *functional* descriptions, that describe the logic operations performed by the cell, and *non-functional* descriptions, that describe other aspects such as timing, layout, or power consumption. The thesis' main focus is on the functional descriptions, non-functional ones are only considered later and with respect to a functional description. Therefore, in the remainder of this chapter, the functional descriptions will be explained further. They can be subdivided into *transistor netlists* and *simulation descriptions*. A transistor netlist of a cell describes the transistors that are present in the physical implementation of that cell and the interconnections among the transistors. Thus, such a description is very close to the finally manufactured chip. On the other hand, a simulation description only models the functional behavior, but is not required to model the exact working of the physical implementation. Thus, in such a simulation description special constructs offered by the language are frequently being used, an example being the User Defined Primitives (UDPs) of Verilog [IEE06]. These help to speed up simulations, but are not easily mapped into a hardware implementation. This thesis will only cover simulation descriptions in a subset of the commonly used language Verilog [IEE06]. However, with some adaptations also other languages, such as VHDL [IEE09], could be used instead.

2.2 Transistor Netlist

Ultimately, any chip design will be implemented using transistors for the logic. A transistor netlist describes these transistors and how they are connected. In the current *CMOS* (Complementary Metal Oxide Semiconductor) design style, two different kinds of transistors are used, *PMOS* (P-channel Metal Oxide Semiconductor) and *NMOS* (N-channel Metal Oxide Semiconductor). In a very rough abstraction, both can be seen as switches, where an NMOS transistor conducts between its *source* and *drain* whenever the *gate* has a high voltage (which is interpreted as a logic 1). A PMOS transistor on the other hand is conducting between source and drain whenever the gate voltage is low (representing a logic 0). MOS transistors also have a fourth connection, called *bulk* or *substrate*, which defines the reference for the gate voltage, but is usually not of relevance in integrated circuits such as cells.

The description of a cell in terms of a transistor netlist is then a number of PMOS and NMOS transistors, together with a number of connections between source, drain, and gate terminals of these transistors, implementing the logic of the cell. Since this is implemented in CMOS style, the logic in the p-doped part and in the n-doped part are *complementary* (hence the 'C'), meaning that a conducting path exists through the PMOS transistors to the high voltage rail (also called V_{dd}, carrying a logic 1) only if there is no conducting path through the NMOS transistors to the low voltage rail (also called V_{ss} and carrying a logic 0), and vice versa. If this were not the case, then a direct connection between high and low voltage rail would exist, giving a short-circuit which would burn the chip.

A transistor netlist description is usually given in the input language of the tool *SPICE* (Simulation Program with Integrated Circuits Emphasis) [NP73]. In contrast to the SPICE tool, the exact working of a transistor is not of importance in this thesis and a transistor is, as already explained above, simply viewed as a switch. Thus, in this thesis, a logic function is read from a SPICE netlist.

An example of a (simplified) SPICE netlist taken from a cell library is presented in Figure 2.1. It shows the transistors of a D flip-flop taken from the Nangate Open Cell Library [Nan08]. First, it defines two global signals, the high voltage source `VDD` and the low voltage source `VSS`. After that, the subcircuit named `DFF_X1` is defined, which has as interface the input signals `CK` and `D`, the output signals `Q` and `QN`,

```
.GLOBAL VDD
.GLOBAL VSS

.SUBCKT DFF_X1 CK D Q QN VDD VSS
M_instance_184 VSS CK net_000 VSS NMOS
M_instance_191 net_001 net_000 VSS VSS NMOS
M_instance_197 VSS net_004 net_002 VSS NMOS
M_instance_204 net_003 D VSS VSS NMOS
M_instance_209 net_004 net_000 net_003 VSS NMOS
M_instance_215 net_005 net_001 net_004 VSS NMOS
M_instance_220 VSS net_002 net_005 VSS NMOS
M_instance_226 net_006 net_004 VSS VSS NMOS
M_instance_230 net_007 net_001 net_006 VSS NMOS
M_instance_236 net_008 net_000 net_007 VSS NMOS
M_instance_240 VSS net_009 net_008 VSS NMOS
M_instance_246 net_009 net_007 VSS VSS NMOS
M_instance_254 VSS net_007 QN VSS NMOS
M_instance_261 Q net_009 VSS VSS NMOS
M_instance_267 VDD CK net_000 VDD PMOS
M_instance_274 net_001 net_000 VDD VDD PMOS
M_instance_281 VDD net_004 net_002 VDD PMOS
M_instance_288 net_010 D VDD VDD PMOS
M_instance_293 net_004 net_001 net_010 VDD PMOS
M_instance_299 net_011 net_000 net_004 VDD PMOS
M_instance_305 VDD net_002 net_011 VDD PMOS
M_instance_311 net_012 net_004 VDD VDD PMOS
M_instance_316 net_007 net_000 net_012 VDD PMOS
M_instance_322 net_013 net_001 net_007 VDD PMOS
M_instance_327 VDD net_009 net_013 VDD PMOS
M_instance_333 net_009 net_007 VDD VDD PMOS
M_instance_339 VDD net_007 QN VDD PMOS
M_instance_346 Q net_009 VDD VDD PMOS
.ENDS
```

Figure 2.1: Simplified SPICE netlist of a D flip-flop taken from the Nangate Open Cell Library

and the two voltage sources. Inside the body of the subcircuit, which ends at the keyword `.ENDS`, the transistors are defined. Every transistor is assigned a name, which has to start with the letter `M`. After the name of a transistor, its four connections are given, which are *drain*, *gate*, *source*, and *bulk*. The bulk connection is not of importance here, and it is always connected to the corresponding voltage rail (`VSS` for NMOS transistors, `VDD` for PMOS transistors). Finally, the type of the transistor is given. In the actual SPICE descriptions, there are additional parameters given after the type in the form of equations, describing physical properties of the transistor. However, these parameters are not of importance when viewing transistors as switches. As an example, the first line `M_instance_184 VSS CK net_000 VSS NMOS` instantiates an NMOS transistor that connects the low voltage rail `VSS` to the internal signal `net_000` in case the interface signal `CK` is high. Together with the PMOS transistor `M_instance_267 VDD CK net_000 VDD PMOS`, which connects the high voltage rail `VDD` to `net_000` in case `CK` is low, it forms an *inverter* which provides the logic negation of signal `CK` on signal `net_000`.

Viewing transistors as switches has already been done by Bryant [Bry87]. In that paper, an algorithm is given to create from a given transistor netlist a description consisting of Boolean equations. These equations describe the logic function of the transistor netlist and hence are taken as the netlist's semantics in the remainder of this thesis. For the example netlist given in Figure 2.1, the following equations are created after simplification (where the logic "and" conjunction $\wedge$ binds stronger than the logic "or" disjunction $\vee$, as usual):

$$\begin{array}{lcllcl}
\mathrm{net\_004} & \equiv & \neg\mathrm{CK}\wedge\neg\mathrm{D} & \vee & \mathrm{CK}\wedge & \mathrm{net\_004}\\
\mathrm{net\_009} & \equiv & \neg\mathrm{CK}\wedge\ \mathrm{net\_009} & \vee & \mathrm{CK}\wedge & \mathrm{net\_004}\\
\mathrm{Q} & \equiv & \neg\mathrm{CK}\wedge\neg\mathrm{net\_009} & \vee & \mathrm{CK}\wedge & \neg\mathrm{net\_004}\\
\mathrm{QN} & \equiv & \neg\mathrm{CK}\wedge\ \mathrm{net\_009} & \vee & \mathrm{CK}\wedge & \mathrm{net\_004}
\end{array}$$

In these equations, it can be observed that the equations for variables $\mathrm{net\_004}$ and $\mathrm{net\_009}$ implement two latches, with inverted enable signals. The latch $\mathrm{net\_004}$ outputs the negated value of input D if the clock input CK is 0 and it keeps its old value if the clock input CK is 1. For the latch $\mathrm{net\_009}$, the old output value is kept if the clock input CK is 0 and it sets its output value to the value of $\mathrm{net\_004}$ if the clock input CK is 1. The output Q is assigned the negated value of $\mathrm{net\_009}$, since the two transistors `M_instance_261` and `M_instance_346` form an inverter with Q as output and $\mathrm{net\_009}$ as input. Finally, the output QN always has the same value as $\mathrm{net\_009}$, as it is the negation of $\mathrm{net\_007}$, which in turn is the negation of $\mathrm{net\_009}$. Therefore, the output QN always is the negation of output Q, as expected.

Note that the method of [Bry87] to extract equations from SPICE netlists works with ternary values, where the third value X denotes "an uninitialized network state or an error condition caused by a short circuit or charge sharing" as is described in [Bry87]. However, it can easily be detected that an equation never outputs the value X. This was used above; thus the variables can only be one of the binary values 0 and 1.

2.3 Verilog Simulation Description

To allow simulations of a chip design implemented as cells without resorting to simulating the numerous transistors, cells are also described at a higher abstraction level. For this purpose, the standardized hardware description language *Verilog* [IEE06] is often employed. However, the Verilog language allows for descriptions at various abstraction levels, hence only a certain subset of this language is used. This subset is called VeriCell in the rest of this thesis and is described below. It differs from other subsets of Verilog, such as for example the synthesizable register-transfer level subset described in [IEE05], in that it does not cover behavioral descriptions. Instead, VeriCell focuses on the constructs found in cell library descriptions such as built-in and user-defined primitives, which are not contained in other subsets of Verilog.

The values that signals can take in a VeriCell description are the *ternary constants* $\mathbb{T} = \{0, 1, \mathsf{X}\}$. Here, the values 0 and 1 behave like the values false and true of the Boolean values $\mathbb{B}$, respectively. The third value X is usually understood as representing an unknown value, however the Verilog standard defines it as a third logic value unrelated to both 0 and 1. It should be remarked that the language Verilog also allows a fourth value Z, which represents a high impedance. However, for the VeriCell subset of Verilog this value is equivalent to the value X. This can easily be seen for the considered built-in primitives from Tables 7-3 and 7-4 in the Verilog standard [IEE06]. For user-defined primitives, this is explicitly stated in [IEE06, Clause 8]. Therefore, the value Z is not considered any further.

Furthermore, VeriCell descriptions contain single-bit variables (e.g., `CK`, `D`) ranging over $\mathbb{T}$, built-in primitives (e.g., **`not`**, **`and`**), and user-defined primitives (UDPs). All of these components are defined in a single module, which constitutes the cell. As an example, the VeriCell description of a D flip-flop taken from the Nangate Open Cell Library [Nan08] is given in Figure 2.2. This description has been simplified by leaving out some details that are not of relevance here and writing it in a more compact form.

The example cell is defined in the **`module`** named `DFF_X1`. In parentheses, the interface of the module is defined, which are those variables connecting the module to its environment. The example module has two inputs, the variables `CK` and `D`, and two outputs, variables `Q` and `QN`, declared in the **`input`** and **`output`** lines, respectively. After these declarations, instances of primitives are created. In VeriCell, it is required that every primitive instance has a unique output variable, i.e., no two primitive instances share a common output. Primitives **`not`** and **`buf`** are built-in primitives. The built-in primitive of **`buf`** copies the input (the last argument) to its output (the first argument), whereas the buit-in primitive **`not`** provides the negation of the input on its output. Further built-in primitives that are allowed in the VeriCell subset are **`and`**, **`nand`**, **`or`**, **`nor`**, **`xor`**, and **`xnor`**, all of which behave as suggested by their name. The syntax and semantics of all of these built-in primitives is defined formally in [IEE06, Clause 7].

The line `seq43(IQ, nextstate, CK)` instantiates a *User Defined Primitive (UDP)*, whose function has to be defined in the source code. This is done between the keywords **`primitive`** and **`endprimitive`**. The UDP is first given a name, in this case `seq43`, which is used in modules to instantiate it. Afterwards, the declaration of the interface and the direction of the variables (input or output) is declared, as is the case for modules. It should be noted that UDPs must always have exactly one output, which must always be the first argument. The number of inputs is allowed to be arbitrary in this thesis (at least one), which corresponds to the general

```
primitive seq43 (IQ, nextstate, CK);
  output IQ; reg IQ;
  input nextstate, CK;

  table
// nextstate  CK   : @IQ :  IQ
          0    r   :   ? :   0;
          1    r   :   ? :   1;
          0    *   :   0 :   0;
          1    *   :   1 :   1;
          *    ?   :   ? :   -;
          ?    f   :   ? :   -;
  endtable
endprimitive

module DFF_X1 (CK, D, Q, QN);
  input CK, D;
  output Q, QN;

  seq43(IQ, nextstate, CK);
  not(IQN, IQ);
  buf(Q, IQ);
  buf(QN, IQN);
  buf(nextstate, D);
endmodule
```

Figure 2.2: Simplified VeriCell description of a D flip-flop taken from the Nangate Open Cell Library

definition of UDPs in the Verilog standard. Note that the standard allows simulators to impose an upper bound on the number of UDP inputs (which must be at least 9), but this does not change the treatment of UDPs presented in this thesis and makes it independent from any specific implementation. The declaration **reg** Q indicates that the UDP is sequential, i.e., the UDPs output does not only depend on the values of the inputs, but also on the previous value of the output, which therefore has to be stored. After the declarations, the logic function of the UDP is defined by means of a table. This table has a column for each input of the UDP, a column separated by a colon for the previous output value, and another column separated by a colon to denote the new output value. The idea of such a row is that whenever the actual input values match the entries of that row and the previous output value matches the entry in that column of the row, then the new output value is set to the value specified in the last column.

Entries in the input column can either match a single value (called *level* specification) or a transition of values (called *edge* specification). For example, the level specification 0 matches exactly that value, whereas the level specification ? matches any value. To match transitions, the first option is to use the syntax (kl) with level specifications k and l. Then, an input changing from an old value $v \in \mathbb{T}$ to a new value $w \in \mathbb{T}$ is matched by the specification (kl) if $v \neq w$, v is matched by k, and

w is matched by l. It should be noted that the requirement $v \neq w$ is not demanded in the Verilog standard [IEE06], but is imposed by all Verilog simulators that were tested. Additional abbreviations of common edge specifications exist. In the example, the specification `r` is an edge specification (rising edge), which is equivalent to the edge specification `(01)`. Other edge specifications used there are `f` (falling edge) which is equivalent to `(10)`, and `*`, which is equivalent to `(??)` (i.e., a transition from some value to any other value). Any row in a UDP may contain at most one edge specification. If there exists an edge specification, then the whole row is called *edge-sensitive*. Otherwise, if there are only level specifications, the row is called *level-sensitive*.

Since the new value of the output is yet to be determined, the column matching the previous output value may only use level specifications. Finally, the last column, denoting the new output value, may only contain single value level specifications, which are the specifications `0`, `1`, and `x`. Additionally, it is permitted to put the special specification `-`, which can be read as "no change". This specification indicates that the old value of the output is also the new value of the output. As an example, the last row `? f : ? : -` of the UDP shall be considered. This row states that if the input `CK` makes a transition from 1 to 0, then regardless of values of the signal `nextstate` and the previous output value the new output value is the same as the old output value. Here, it can be seen that adding the specification `-` can make a UDP definition more compact. If it were not allowed, then one could remove the `-` by expanding the previous output value specification `?` to all three single value specifications `0`, `1`, and `x` and then copying the same value into the last column, so that one would replace for example the last row by the following three rows:

```
?   f   :   0 :   0;
?   f   :   1 :   1;
?   f   :   x :   x;
```

To evaluate a UDP, one therefore searches for a matching row in its table, and takes the output that is denoted in that rows last column. If none of the rows of a UDP matches, then the standard defines the new output to be X. In case multiple rows match, the standard imposes some rules to select the row to be used. These rules will be explained in full detail later in Section 3.1.

The syntax of Verilog, and hence also that of the VeriCell subset, is defined formally in the standard [IEE06]. However, the semantics is not defined formally. In the case of VeriCell descriptions, the exact semantics of UDPs, which is described informally in [IEE06, Clause 8], is ambiguous. Hence, to be able to verify VeriCell descriptions, a formal semantics is defined in Section 3.1 which is used throughout this thesis.

CHAPTER 3

Equivalence Checking in Cell Libraries

As explained in the previous chapter, cell libraries contain multiple functional descriptions for each cell. Therefore, it should be ensured that every cell has the same behavior in all of these descriptions. If this is not the case, then a design that worked for example in a simulation might fail when produced as a chip, incurring huge costs.

This chapter addresses the problem of verifying that the functional description given in Verilog (or, more precisely, in the VERICELL subset introduced in Section 2.3) exhibits the same behavior as the transistor netlist description. This chapter is based on [RRM09] and presents an operational semantics for VERICELL and encodes it into Boolean equations. Together with the Boolean equations created from the transistor netlist, which are extracted using the algorithm of [Bry87] as discussed in Section 2.2, equivalence can be checked using a model checker, such as for example the NuSMV model checker [CCG+02] or the Cadence SMV model checker [McM97], which are specialized on transition systems described as Boolean equations.

The syntax of Verilog, and therefore also the syntax of the VERICELL subset, is formally defined in the IEEE Verilog standard [IEE06]. However, the semantics of this language is left ambiguous in certain parts. It is only explained how certain example situations should be treated, which leaves room for different interpretations. Quite a few publications exist that try to fill the semantic gap, for example in [Dim01, Gor95, HBJ01]. However, they usually address higher level constructs and not those elements found in cell libraries; especially, they do not consider the User Defined Primitives (UDPs). An approach covering some aspects of UDPs is reported in [WW98]. This approach, however, is mainly geared towards an encoding of Verilog into gate level networks (via Ordered Ternary Decision Diagrams, OTDDs). To that end, [WW98] uses heuristics/pattern recognition to detect more complex functions, such as multiplexers and **`xor`** gates, in the Verilog description. The encoding itself is however not formalized and hence it is unclear how the problems that were identified in the semantics given below are dealt with. An example are multiple inputs to a cell changing at the same time, which gives rise to non-deterministic behavior, as will be shown. Furthermore, the goal of [WW98] was to create correct-by-construction gate level descriptions, whereas here the goal is to enable formal verification of given Verilog cells.

```
module flip_flop (q, d, ck, rb);
  output q; input rb, d, ck;

  not (ckb, ck);
  latch (iq  , d , ck , rb);
  latch (qint, iq, ckb, rb);
  buf (q, qint);
endmodule

primitive latch (Q, D, CK, RB);
  output Q; reg Q; input D, CK, RB;
  table
//  D      CK      RB      :    Q  :   Q'
    0     (?1)     ?       :    ?  :    0;
    1     (?1)     1       :    ?  :    1;
    ?     (?0)     ?       :    ?  :    -;
    ?      *       0       :    0  :    -;
    ?      ?      (?0)     :    ?  :    0;
    ?      0      (?1)     :    ?  :    -;
    0      1      (?1)     :    0  :    -;
    1      1      (?1)     :    ?  :    1;
    *      0       ?       :    ?  :    -;
    *      ?       0       :    0  :    -;
   (?0)    1       ?       :    ?  :    0;
   (?1)    1       1       :    ?  :    1;
  endtable
endprimitive
```

Figure 3.1: VeriCell description of a resettable flip-flop

Traditional equivalence checking techniques used for higher level descriptions, e.g., those based on [vE00], are not applicable to this problem, as they rely on certain structures (for example a synchronous gate-level model and a given set of flip-flops) to perform matching and to apply retiming. However, in the presented setting of cell libraries, no such generic structures exist and the elements are custom made.

3.1 Semantics of VeriCell

The language VeriCell is a subset of the Verilog Hardware Definition Language, which is defined in the IEEE standard 1364-2005 [IEE06]. It was already explained in Section 2.3 that VeriCell consists of the built-in primitives, user defined primitives (UDPs), and of modules that define the interconnection of these primitives. An example VeriCell program is given in Figure 3.1, which defines a flip-flop that can be reset.

In the remainder only sequential UDPs will be considered, i.e., UDPs that may contain edge specifications matching input transitions and which furthermore may match the previous value of the output in order to determine their next output value, cf. Section 2.3. This is a syntactic restriction and does not influence the semantics: any combinational UDP can be converted into a sequential UDP by ignoring the previous value of the output, which can be achieved by adding a new penultimate entry ? in every row. For sequential UDPs, the full syntax given in the standard is included in the syntax of VeriCell. Below however, the handling of initial UDP output values is not presented, since it is a rarely used feature and can easily be accommodated by adjusting the initial configuration in which evaluations start.

Preliminaries

The semantics of VeriCell is defined in an operational style by transforming configurations. In order to define the semantics, first some notations used in the remainder of the section are introduced.

All variables in Verilog can have one of the four values Z, 0, 1, or X. However, for the primitives allowed in the VeriCell subset of Verilog, the values Z and X always have the same meaning, representing an unknown value. Therefore, only the *ternary values* $\mathbb{T} = \{0, 1, X\}$ are considered. Here, the values 0 and 1 correspond to the values false and true of the Booleans $\mathbb{B}$, respectively. The value X is intended to represent an unknown Boolean value. Hence, the usual Boolean operations are extended in a pessimistic way, i.e., $\neg X = X$, $0 \wedge X = 0$, $1 \wedge X = X$, and $X \wedge X = X$. All other basic Boolean functions on ternary values can be derived from these definitions. Note however that the Verilog standard [IEE06] defines the value X to be a third value, unrelated to the Boolean values 0 and 1. In this thesis, its intended interpretation is that it stands for an unknown Boolean value; however there are even different interpretations in different application domains. For example, the value X also be viewed as a “don’t care” during synthesis, see for example [Tur03] for an in-depth discussion of the problems with the value X that occur in higher-level Verilog descriptions. In VeriCell, problems with the value X occur since it can be explicitly matched by UDPs. Hence, a UDP can behave completely different from when an X value is instantiated arbitrarily with either 0 or 1. Thus, the semantics presented here treats the value X as a separate third value, which is exactly the same as in the Verilog standard. A single ternary value $y \in \mathbb{T}$ is also called a *level*, whereas a pair of two ternary values $(y^p, y) \in \mathbb{T} \times \mathbb{T}$, representing a transition, is also called an *edge*.

Given a VeriCell program, let UDPs (UDPs_n) denote the set of UDPs in the program (that have exactly n inputs). The set Prims denotes the set of all primitives that are used in the program, comprising both UDPs and built-in primitives.

Multiple inputs of a UDP can change simultaneously, even if the inputs of the complete cell are required to only change one at a time. Since UDPs only allow one input to transition, this is treated by considering the changing inputs sequentially. This order is assumed to be subject to influences that are not under the designer’s control, hence the order is assumed to be random. To represent these orders, *permutations* are used. For a given number $n \in \mathbb{N}$ of inputs, Π_n denotes the set of all permutations of the set $\{1, \ldots, n\}$. To also be able to represent parts of a permutation, these are generalized to *lists*. A list $\ell \in \mathcal{L}_n$ is a sequence of numbers from the set $\{1, \ldots, n\}$ without duplicates, and the set $\mathcal{L}_n$ represents all such lists. The empty list is denoted by nil, a list having first element j and a tail list ℓ' is denoted by $j : \ell'$. As a notational convention, it is allowed to leave out the trailing nil of a non-empty list, thus, for example, the list $1 : 2 : \mathsf{nil}$ may also be written as $1 : 2$. The length of a list is the number of elements it contains. This can be defined inductively by $|\mathsf{nil}| = 0$ and $|j : \ell'| = 1 + |\ell'|$. Then, permutations are those lists $\pi \in \mathcal{L}_n$ with $|\pi| = n$. A list $\ell = j_1 : \cdots : j_{|\ell|} \in \mathcal{L}_n$ can be constructed by *concatenating* two lists $\ell_1, \ell_2 \in \mathcal{L}_n$, denoted $\ell_1 +\!\!+ \ell_2$, if $\ell_1 = j_1 : \ldots j_k$ and $\ell_2 = j_{k+1} : \cdots : j_{|\ell|}$ for some $1 \leq k \leq |\ell|$.

Next, the semantics of the primitives is defined. This semantics specifies the output value of a primitive given the previous and current values of the inputs and the previous value of the output, which can be expressed by a denotation function. Afterwards, this semantics is lifted to capture the instantiation of primitives and their interconnections. The latter takes the form of deduction rules.

Output Value of Primitives

For the semantics of built-in primitives the straight-forward intuitive semantics given in [IEE06, Tables 7-3 and 7-4] is formalized. However, no such definition exists for UDPs, which therefore will be given below. Then, at the end of this sub-section, the built-in primitives are included to create an evaluation function for both built-in and user defined primitives.

Non-Deterministic Output Value Computation of UDPs

The idea of a UDP is to look up the corresponding output value in the table that is given in its declaration for given previous and current values of its inputs and its previous output value, as described in the Verilog standard [IEE06, Clause 8]. The standard requires that level-sensitive rows take precedence over edge-sensitive rows, i.e., if there are both a level-sensitive and an edge-sensitive row applicable to the current input values, then the output is determined by the level-sensitive row. For example, consider a UDP containing the two rows `(0?) : 0 : 1` and `1 : ? : 0`. If the previous output value of this UDP is 0 and the input changes from 0 to 1 then both rows are applicable. But due to the above-mentioned requirement the output must always be 0, since this is the output of the level-sensitive row.

However, the standard does not define how to handle the case of multiple inputs changing at the same time. To this end, the outcome of several Verilog simulators such as CVer [Pra07], ModelSim [Men08], VeriWell [Wel08], and Icarus [Wil07] were compared (unfortunately, some simulators such as Verilator [Sny08] do no support UDPs). It was observed that each of these simulators implements a slightly different semantics for UDPs, where the open source simulator CVer and the commercial simulator ModelSim provide the outcome that is consistent with what is specified in the standard and also closest to the intuition of the designers. One particular difference is the order used by the simulators to evaluate multiple changing inputs. The selection of such an order is not required for the semantics presented here, hence the semantics allows for each of the different simulator behaviors in this respect.

Ultimately, the concrete order used for evaluation should not be affecting the computation results in cases where it cannot be controlled. Otherwise, a simulator that always chooses exactly one order of evaluating changing inputs cannot faithfully model the behavior of the finally produced chip, where the input changes might occur in different orders at different times. This problem is addressed in Chapters 4 and 5, which present analysis techniques to guarantee equivalent behavior for all possible orders.

The level specifications `0`, `1`, and `x` that may occur in a truth table of a primitive directly correspond to the ternary values 0, 1, and X, respectively. Hence, for $l \in \{\mathtt{0}, \mathtt{1}, \mathtt{x}\}$ and $y \in \mathbb{T}$, the predicate $\mathrm{match}(l, y)$ is defined to be true if and only if l and y correspond. This is formally defined in Table 3.1. The additional level specifications `b` and `?` are syntactic sugar, where the first one corresponds to both 0

Table 3.1: Matching of UDP specifications to inputs

$$y^p, y, o^p \in \mathbb{T},\ e \in \mathbb{T}\times\mathbb{T},\ i_1,\ldots,i_n \in \mathbb{T}\cup(\mathbb{T}\times\mathbb{T})$$

$$\begin{array}{lcllcl}
\mathrm{match}(\mathtt{0},y) & \doteq & y=\mathtt{0} & \mathrm{match}(\mathtt{1},y) & \doteq & y=\mathtt{1}\\
\mathrm{match}(\mathtt{x},y) & \doteq & y=\mathtt{x} & \mathrm{match}(\mathtt{b},y) & \doteq & \mathrm{match}(\mathtt{0},y)\vee\mathrm{match}(\mathtt{1},y)\\
\mathrm{match}(\mathtt{?},y) & \doteq & \mathsf{true} & & &
\end{array}$$

$$\begin{array}{lcl}
\mathrm{match}(\mathtt{(vw)},(y^p,y)) & \doteq & y^p\neq y\wedge\mathrm{match}(\mathtt{v},y^p)\wedge\mathrm{match}(\mathtt{w},y)\\
\mathrm{match}(\mathtt{r},e) & \doteq & \mathrm{match}(\mathtt{(01)},e)\\
\mathrm{match}(\mathtt{f},e) & \doteq & \mathrm{match}(\mathtt{(10)},e)\\
\mathrm{match}(\star,e) & \doteq & \mathrm{match}(\mathtt{(??)},e)\\
\mathrm{match}(\mathtt{p},e) & \doteq & \mathrm{match}(\mathtt{(01)},e)\vee\mathrm{match}(\mathtt{(0x)},e)\vee\mathrm{match}(\mathtt{(x1)},e)\\
\mathrm{match}(\mathtt{n},e) & \doteq & \mathrm{match}(\mathtt{(10)},e)\vee\mathrm{match}(\mathtt{(1x)},e)\vee\mathrm{match}(\mathtt{(x0)},e)
\end{array}$$

$$\mathrm{matchRow}(s_1\ldots s_n : s_{n+1} : o, (i_1,\ldots,i_n), o^p) \doteq \bigwedge_{1\le j\le n}\mathrm{match}(s_j,i_j)\wedge\mathrm{match}(s_{n+1},o^p)$$

and 1 and the latter one corresponds to all of the ternary values. Thus, $\mathrm{match}(\mathtt{b},y)$ is true if and only if y is either 0 or 1, whereas $\mathrm{match}(\mathtt{?},y)$ is always true, regardless of the value of y.

An edge specification in a UDP has the general form $\mathtt{(}vw\mathtt{)}$, where v and w are level specifications. Given two values $y^p, y \in \mathbb{T}$, the predicate $\mathrm{match}(\mathtt{(}vw\mathtt{)},(y^p,y))$ is defined to be true if and only if $\mathrm{match}(v,y^p)$ and $\mathrm{match}(w,y)$ are true and $y^p \neq y$. This latter requirement, which states that indeed a transition must take place, is left ambiguous by the standard, however it is enforced by all simulators of Verilog that were tested, listed previously. The remaining edge specifications `r`, `f`, `p`, `n`, and $\star$ are expressed using the above and their definitions as given in [IEE06, Table 8-1], as can be seen in Table 3.1. For example, for the rising specification `r` the predicate $\mathrm{match}(\mathtt{r},(y^p,y))$ is true if and only if $\mathrm{match}(\mathtt{(01)},(y^p,y))$ is true, which in turn is true if and only if y^p is 0 and y is 1, whereas $\mathrm{match}(\star,(y^p,y)) = \mathrm{match}(\mathtt{(??)},(y^p,y))$ is true if and only if y^p and y are different values.

The above matching of single values is combined into a predicate matchRow that checks whether a row of a UDP is applicable for a certain vector of inputs. This predicate, whose formal definition is given in Table 3.1, has as first argument a row of a UDP, in which at most one edge specification may occur. As the second argument, it takes a tuple that contains both levels and edges, where there must be at most one edge. This tuple must have the same length as the number of inputs of the UDP. The last argument of matchRow is the previous output value, which must be a level. It follows from the definition of match that a level specification only matches a level and an edge specification only matches an edge. Therefore, the row matches the inputs if all level inputs have been matched by a level specification and the edge specification, in case the row is edge-sensitive, matches an edge at the same position in the vector of inputs. Furthermore, there must be at most one edge in the inputs, since there is at most one edge specification allowed in a row. To illustrate this, consider the UDP from Figure 3.1. For the input values `D` $= 1$, `CK` $= (1,0)$, and `RB` $= 1$, the previous output value `Q` $= 0$, and the row in line 16, it holds that $\mathrm{matchRow}(\mathtt{?\ (?0)\ ?\ :\ ?\ :\ -}, (1,(1,0),1),0)$ is true. But if the input `CK` is the level 0 and the other values are kept equal, then $\mathrm{matchRow}(\texttt{? (?0) ? : ? : -}, (1,0,1), 0)$ is false since $\mathrm{match}(\texttt{(?0)}, 0)$ is false (this even holds when identifying 0 with $(0,0)$, since it is required that the values are different for an edge to match, as discussed previously).

The output of a UDP row is given by the ternary value that corresponds to the level specification in the last column. Additionally, the special symbol "–" is also allowed there. This value indicates that no change happens, i.e., the output value is the same as the previous output value. Thus, the row `? * 0 : 0 : -` occurring in line 17 of the flip-flop example in Figure 3.1 could also be written as `? * 0 : 0 : 0`, as was already discussed in Section 2.3.

Using this and the matching of rows, it would be desired to construct a function that computes the output of a UDP given the previous and current values of the inputs and the previous value of the output. For this purpose, the Verilog standard does define two rules that govern the selection of a row when computing a new output value. The first requirement is that level-sensitive rows take precedence over edge-sensitive rows, as already mentioned above. The second requirement is that two rows of the same type (i.e., either both level-sensitive or both edge-sensitive with the edge in the same column) that are both applicable to some inputs and previous output, must not disagree on the output value.

These rules however are not precise enough to induce a function for the computation of a UDP's next output value. If multiple inputs of a UDP change at the same time, then it is not clear how to handle this. In order for the semantics to be as general as possible, it is therefore assumed that non-deterministically an order is chosen and according to this order the changed inputs are considered one change at a time. Hence, the semantics of a UDP is defined to be a function that is parametrized by the current UDP, the previous and current values of the inputs, the previous output value, and some order. Such an order is given by a permutation of the numbers from 1 to n that dictates the order of checking the inputs whether their values have changed. This gives the following signature of the evaluation function $[\![\cdot]\!]$ for a UDP $\texttt{udp} \in \mathrm{UDPs}_n$: $[\![\cdot]\!] : \mathrm{UDPs}_n \times (\mathbb{T} \times \mathbb{T})^n \times \mathbb{T} \times \mathcal{L}_n \to \mathbb{T}$. Here, permutations are generalized to lists without duplicates, to allow for the below recursive definition, given some previous and current input values $i_1^p, i_1, \ldots, i_n^p, i_n \in \mathbb{T}$, some previous output $o^p \in \mathbb{T}$, and some list $j : \ell \in \mathcal{L}_n$:

$$\begin{aligned}
[\![\texttt{udp}, ((i_1^p, i_1), \ldots, (i_n^p, i_n)), o^p, \mathsf{nil}\ \ ]\!] &\doteq o^p \\
[\![\texttt{udp}, ((i_1^p, i_1), \ldots, (i_n^p, i_n)), o^p, j : \ell]\!] &\doteq \\
&[\![\texttt{udp}, ((i_1^p, i_1), \ldots, (i_j, i_j), \ldots, (i_n^p, i_n)), o', \ell]\!]
\end{aligned}$$

The next output value o' is defined as follows. If $i_j^p = i_j$, then $o' = o^p$, i.e., the value remains unchanged. Otherwise, o' is defined to be the corresponding output value of either a level-sensitive row r of the UDP `udp` for which $\mathrm{matchRow}(r, (i_1^p, \ldots, i_j, \ldots, i_n^p), o^p)$ is true, or o' is the corresponding output value of an edge-sensitive row r of the UDP for which $\mathrm{matchRow}(r, (i_1^p, \ldots, (i_j^p, i_j), \ldots, i_n^p), o^p)$ is true and for all level-sensitive rows r' of the UDP the property $\mathrm{matchRow}(r', (i_1^p, \ldots, i_j, \ldots, i_n^p), o^p)$ is false. If no such row exists, then the next output value o' is defined to be X.

As an example, consider the user defined primitive `latch` given in Figure 3.1. Its output value shall be determined for the input values $\texttt{D} = (1, \mathsf{X})$, $\texttt{CK} = (1,0)$, and $\texttt{RB} = (1,1)$ and the previous output value $\texttt{Q} = 0$. If the order $\pi = 2 : 1 : 3$ is used, then the first intermediate output value is 0, since line 16 satisfies matchRow(`? (?0) ? : ? : -`$, (1,(1,0),1), 0)$, as already illustrated before. Now the previous value of `CK` is updated and the change of input `D` is considered. Here, line 22 matches, giving a next output value `Q` $= 0$. Finally, since the input `RB` did not change, this is also the output of this instance given these inputs, the previous output, and the considered order.

However, when changing the order to $\pi' = \mathrm{id} = 1 : 2 : 3$, then the following computation gives as new output value X, which shows that the order can change the result of a computation.

$$\begin{array}{lll}
 & [\![\texttt{latch}, ((1,\mathsf{X}),(1,0),(1,1)), 0, 1:2:3]\!] & \\
= & [\![\texttt{latch}, ((\mathsf{X},\mathsf{X}),(1,0),(1,1)), \mathsf{X}, 2:3]\!] & \text{(no match, default)} \\
= & [\![\texttt{latch}, ((\mathsf{X},\mathsf{X}),(1,1),(1,1)), \mathsf{X}, 3]\!] & \text{(line 16 matches)} \\
= & [\![\texttt{latch}, ((\mathsf{X},\mathsf{X}),(1,1),(1,1)), \mathsf{X}, \mathsf{nil}]\!] & \text{(3}^{\text{rd}}\text{ input unchanged)} \\
= & \mathsf{X} &
\end{array}$$

Thus, UDPs can behave differently depending on the order in which input changes are processed. This order is chosen non-deterministically, which therefore also makes the evaluation of UDPs non-deterministic. As mentioned previously, this is investigated further in Chapters 4 and 5. Chapter 4 presents an efficient technique to guarantee that the output value of a UDP is independent of the concrete order chosen. This however is not the case very often, since certain order-dependencies such as the above, where the relative order of the clock and data inputs matters, are expected. This information is contained in the timing checks, which constrain the possible orders and are therefore incorporated into the analysis of order-independence in Chapter 5.

Output Value Computation of Built-In Primitives

Finally, the function $[\![\cdot]\!]$ is extended to also incorporate the semantics of built-in primitives. In this way, a function $[\![\cdot]\!] : \mathrm{Prims} \times (\mathbb{T} \times \mathbb{T})^n \times \mathbb{T} \times \mathcal{L}_n \to \mathbb{T}$ is obtained, that uses the semantics of the built-in primitives as given in [IEE06, Table 7-3 and Table 7-4]. Note that all built-in primitives are combinational, therefore the list indicating the order of evaluating inputs, the previous input values, and the previous output value can simply be ignored for them. This can be seen in their definitions show below, which use only the operations $\wedge$ and $\neg$ extended to the ternary values $\mathbb{T}$, as given in the Preliminaries. There, the previous input values $i_1^p, i_2^p \in \mathbb{T}$ and the previous output value $o^p \in \mathbb{T}$, as well as the orders $\pi_j \in \Pi_j$ for $j = 1, 2$ are ignored; the output values are only computed from the input values $i_1, i_2 \in \mathbb{T}$.

$$\begin{array}{lllll}
[\![\texttt{buf}, \big((i_1^p, i_1)\big), o^p, \pi_1]\!] & \doteq & i_1 & & \\
[\![\texttt{not}, \big((i_1^p, i_1)\big), o^p, \pi_1]\!] & \doteq & \neg\, i_1 & & \\
[\![\texttt{and}, \big((i_1^p, i_1), (i_2^p, i_2)\big), o^p, \pi_2]\!] & \doteq & i_1 \wedge i_2 & & \\
[\![\texttt{nand}, \big((i_1^p, i_1), (i_2^p, i_2)\big), o^p, \pi_2]\!] & \doteq & \neg(i_1 \wedge i_2) & & \\
[\![\texttt{or}, \big((i_1^p, i_1), (i_2^p, i_2)\big), o^p, \pi_2]\!] & \doteq & i_1 \vee i_2 & \doteq & \neg(\neg i_1 \wedge \neg i_2) \\
[\![\texttt{nor}, \big((i_1^p, i_1), (i_2^p, i_2)\big), o^p, \pi_2]\!] & \doteq & \neg(i_1 \vee i_2) & \doteq & \neg i_1 \wedge \neg i_2 \\
[\![\texttt{xor}, \big((i_1^p, i_1), (i_2^p, i_2)\big), o^p, \pi_2]\!] & \doteq & i_1 \oplus i_2 & \doteq & \neg\big(\neg(i_1 \wedge \neg i_2) \wedge \neg(\neg i_1 \wedge i_2)\big) \\
[\![\texttt{xnor}, \big((i_1^p, i_1), (i_2^p, i_2)\big), o^p, \pi_2]\!] & \doteq & \neg(i_1 \oplus i_2) & \doteq & \neg(i_1 \wedge \neg i_2) \wedge \neg(\neg i_1 \wedge i_2)
\end{array}$$

The Verilog standard allows for an arbitrary number of output variables for the **`buf`** and **`not`** primitives, and an arbitrary number of input values (at least 1) for the built-in primitives **and**, **nand**, **or**, **nor**, **xor**, and **xnor**. Multiple output variables are treated by copying the primitive instantiations, so that every instantiation has exactly one output variable. If only a single input value is given for any of the (normally binary) primitives, then this input value is also the output value. More than 2 input values provided for such a primitive are treated by repeating the evaluations multiple times, for example, $[\![\texttt{and}, \big((i_1^p, i_1), (i_2^p, i_2), (i_3^p, i_3)\big), o^p, \pi_3]\!] = \left[\!\!\left[\texttt{and}, \Big([\![\texttt{and}\big((i_1^p, i_1), (i_2^p, i_2)\big) o^p, \pi_2]\!], (i_3^p, i_3)\Big), o^p, \pi_2\right]\!\!\right] = i_1 \wedge i_2 \wedge i_3$, where the order $\pi_2 \in \Pi_2$ can be chosen arbitrarily, since it is ignored.

Simulation Semantics

After having defined how to evaluate a single primitive, the semantics of a complete VeriCell program is given. Such a program usually contains a number of instantiations of primitives. As already noted above, the possible values of variables are the ternary values in $\mathbb{T}$. To keep track of such values in a current configuration, *variable valuations* are defined. A variable valuation val is a partial function mapping identifiers to values from $\mathbb{T}$. If for some identifier x the value $val(x)$ is not defined, then it defaults to X. A variable valuation is denoted as a set of pairs of identifiers and values, e.g., $\{(x, 0)\}$ represents the variable valuation that maps the identifier x to 0 and all other identifiers to the default value X. To update variable valuations, the operation juxtaposition is used. Given two variable valuations val_1 and val_2, this operation is defined as $val_1\ val_2(x) = val_2(x)$ if $val_2(x)$ is defined, otherwise $val_1\ val_2(x) = val_1(x)$. For example, $\{(\texttt{d}, 0), (\texttt{ck}, 1)\}\ \{(\texttt{ck}, \mathsf{X}), (\texttt{rb}, 0)\} = \{(\texttt{d}, 0), (\texttt{ck}, \mathsf{X}), (\texttt{rb}, 0)\}$.

The simulation semantics of Verilog is sketched in [IEE06, Clause 11], however it does not deal with the details of when and how to update values. For example, it is not defined when a transition of a variable can be observed by primitives that have this variable as an input. Therefore, when the simulation semantics sketched in [IEE06, Clause 11] is ambiguous, the formal semantics is based on the observations of simulators. As stated before, the choices regarding ambiguities match the interpretation used by CVer and ModelSim.

The execution is split into three different phases, namely, *execute*, *update* and *time-advance*. Execute and update phases are performed iteratively until the current state is stabilized. Only then, the time-advance phase advances the global simulation clock. In the execute phase, all *active* primitives, i.e., primitives for which an input has changed its value, compute their new output values. The output values are stored in a separate location and are not directly used for the evaluation of other primitives, thereby modeling a parallel execution of the primitives. In the update phase, which follows the execute phase, all these values are stored as the new values of the variables. This can again make primitives active, in this case another execute phase is performed.

For example, in the module `flip_flop` shown in Figure 3.1 a change of the variable `d` might result in a change of the variable `iq`. Since this variable is used as an input to another primitive instantiation with the output variable `qint`, this primitive will be activated and executed. The computation is repeated until no more updates are pending and no primitives are active anymore. Then, the third phase, called time advance phase, is entered in which the global simulation time advances and new inputs are applied. These can again activate primitives in the program, since

Table 3.2: Deduction Rules for the Semantics of VeriCell Programs

(ex)
$$\frac{\pi \in \Pi_n}{\begin{array}{l}\langle t, \mathit{prev}, \mathit{cur}, \mathit{act} \uplus \{\mathrm{p}(o, i_1, \ldots, i_n)\}, \mathit{up}\rangle_E \quad \rightarrow \\ \qquad \langle t, \mathit{prev}, \mathit{cur}, \mathit{act}, \mathit{up}\ \{(o, [\![\mathrm{p}, ((\mathit{prev}(i_1), \mathit{cur}(i_1)), \\ \qquad\qquad \ldots, (\mathit{prev}(i_n), \mathit{cur}(i_n))), \mathit{cur}(o), \pi]\!])\}\rangle_E\end{array}}$$

(ex-up)
$$\frac{}{\langle t, \mathit{prev}, \mathit{cur}, \emptyset, \mathit{up}\rangle_E \rightarrow \langle t, \mathit{cur}, \mathit{cur}, \emptyset, \mathit{up}\rangle_U}$$

(up)
$$\frac{\mathit{up} \neq \emptyset}{\begin{array}{l}\langle t, \mathit{prev}, \mathit{cur}, \emptyset, \mathit{up}\rangle_U \rightarrow \\ \qquad \langle t, \mathit{prev}, \mathit{cur}\ \ \mathit{up}, \mathit{sens}(\mathit{cur}, \mathit{cur}\ \ \mathit{up}), \emptyset\rangle_U\end{array}}$$

(up-ex)
$$\frac{\mathit{act} \neq \emptyset}{\langle t, \mathit{prev}, \mathit{cur}, \mathit{act}, \emptyset\rangle_U \rightarrow \langle t, \mathit{prev}, \mathit{cur}, \mathit{act}, \emptyset\rangle_E}$$

(time)
$$\frac{}{\begin{array}{l}\langle t, \mathit{prev}, \mathit{cur}, \emptyset, \emptyset\rangle_U \rightarrow \\ \qquad \langle t+1, \mathit{cur}, \mathit{cur}\ \ \overrightarrow{\mathit{in}_{t+1}}, \mathit{sens}(\overrightarrow{\mathit{in}_t}, \overrightarrow{\mathit{in}_{t+1}}), \emptyset\rangle_E\end{array}}$$

for example the input d might be assigned a different value, so that the execute phase is entered again.

Configurations, i.e., operational states of the simulation semantics, comprise a natural number t, denoting the current simulation time, and the previous and the current valuations of variables in the module, denoted by prev and cur, respectively. Another component of a configuration is a set act of primitive instantiations that have to be evaluated due to the change of some input. Furthermore, a third variable valuation up is contained that collects the updates to be performed. Finally, in order to distinguish the current phase, a flag called phase is introduced that is either E for Execute or U for Update. The flag phase does not model the time-advance phase, since this phase it is modeled as a transition from an update to an execute phase. A configuration is written as $\langle t, \mathit{prev}, \mathit{cur}, \mathit{act}, \mathit{up}\rangle_{\mathit{phase}}$.

Initially, a Verilog program starts in a configuration where the time is 0, all variables have the value X, no primitives are active, and no updates have to be executed, which is denoted by $\langle 0, \emptyset, \emptyset, \emptyset, \emptyset\rangle_U$. Starting in this initial configuration, the deduction rules presented in the remainder of this section are used to transform a current configuration into a next configuration until this is not possible anymore.

It is assumed that a sequence $\overrightarrow{\mathit{in}_1}, \overrightarrow{\mathit{in}_2}, \ldots$ of variable valuations called *input vectors* is given. These variable valuations only assign values to the external inputs declared in the module of the VeriCell program. They are applied whenever the simulation time is allowed to advance.

The operational semantics of a VeriCell program is defined by the deduction rules given in Table 3.2.

In the execution phase, an active primitive is non-deterministically chosen, executed, and removed from the set of active primitives. This is expressed by rule (ex) in Table 3.2. There, π is an arbitrarily chosen order for the evaluation of inputs. Note that this order may differ for each evaluation of a primitive.

When no more active primitives exist then the simulation changes into the update phase. During this change the current transitions of variables are cleared, since all primitives having a changed variable as input have been executed. The new previous values are contained in the cur variable valuation, hence this variable valuation will be used as the new previous variable valuation, as can be seen in rule (ex-up) in Table 3.2.

In the update phase, the new output values are written back into the current variable values. Furthermore, all primitives are activated that have a changed variable as one of their inputs. This is accomplished by the rule (up) in Table 3.2, where it is required that $up \neq \emptyset$, i.e., there must be pending updates. The function $sens$ computes for two variable valuations the set of primitives that have a changed variable as an input. Formally, this function is defined as $sens(val_1, val_2) = \{\texttt{p}(o, i_1, \ldots, i_n) \in \text{Prims} \mid \exists 1 \leq j \leq n : val_1(i_j) \neq val_2(i_j)\}$.

When all updates have been performed, then the execute phase is entered again if there are active primitives. This is expressed in the rule (up-ex) in Table 3.2, which is only applicable if $act \neq \emptyset$. Otherwise, if there are no active primitives, then time advances and the new input values are applied. Furthermore, the new previous values are the current values, because the current state is stable and any present changes can be disregarded. The changed inputs can however activate primitives. To determine these, the function $sens$ is used again in the rule (time) in Table 3.2.

If no input vector $\overrightarrow{in_{t+1}}$ is available anymore, then the simulation terminates. The trace of output values generated by a certain trace of input vectors is defined to consist of the values in the stable states, i.e., states in which the time can advance or simulation terminates.

To illustrate this semantics, the example given in Figure 3.1 is considered again. Let $\overrightarrow{in_1} = \{(\texttt{d}, 0), (\texttt{ck}, 1)\}$ and $\overrightarrow{in_2} = \{(\texttt{d}, 1)\}$ be the external inputs for time steps 1 and 2, respectively. These inputs allow the following simulation starting from the initial configuration, where `latch` is abbreviated to `l`:

$\langle 0, \emptyset, \emptyset, \emptyset, \emptyset \rangle_U$

Time Advance, applying inputs $\overrightarrow{in_1}$, and activating instantiations

$\rightarrow \langle 1, \emptyset, \{(\texttt{d}, 0), (\texttt{ck}, 1)\}, \{\texttt{l(iq,d,ck,rb)}, \textbf{not}\texttt{(ckb,ck)}\}, \emptyset \rangle_E$

Execution of `l(iq, d, ck, rb)` with order $4:3:2:1$

$\rightarrow \langle 1, \emptyset, \{(\texttt{d}, 0), (\texttt{ck}, 1)\}, \{\textbf{not}\texttt{(ckb,ck)}\}, \{(\texttt{iq}, 0)\} \rangle_E$

Execution of **not**`(ckb,ck)`

$\rightarrow \langle 1, \emptyset, \{(\texttt{d}, 0), (\texttt{ck}, 1)\}, \emptyset, \{(\texttt{iq}, 0), (\texttt{ckb}, 0)\} \rangle_E$

Execute $\rightarrow$ Update with clearing of edges

$\rightarrow \langle 1, \{(\texttt{d}, 0), (\texttt{ck}, 1)\}, \{(\texttt{d}, 0), (\texttt{ck}, 1)\}, \emptyset, \{(\texttt{iq}, 0), (\texttt{ckb}, 0)\} \rangle_U$

Updating values of `iq` and `ckb`, activating `l(qint, iq, ckb, rb)`

$\rightarrow \langle 1, \{(\texttt{d}, 0), (\texttt{ck}, 1)\}, \{(\texttt{d}, 0), (\texttt{ck}, 1), (\texttt{iq}, 0), (\texttt{ckb}, 0)\},$
$\{\texttt{l(qint,iq,ckb,rb)}\}, \emptyset \rangle_U$

Update $\rightarrow$ Execute

$\rightarrow \langle 1, \{(\texttt{d}, 0), (\texttt{ck}, 1)\}, \{(\texttt{d}, 0), (\texttt{ck}, 1), (\texttt{iq}, 0), (\texttt{ckb}, 0)\},$

$$\{\texttt{l(qint,iq,ckb,rb)}\}, \emptyset\rangle_E$$

Execution of `l(qint, iq, ckb, rb)` with order $1:2:3:4$

$$\rightarrow \langle 1, \{(\texttt{d},0),(\texttt{ck},1)\}, \{(\texttt{d},0),(\texttt{ck},1),(\texttt{iq},0),(\texttt{ckb},0)\}, \emptyset, \{(\texttt{qint},\mathsf{X})\}\rangle_E$$

Execute $\rightarrow$ Update, clearing edges

$$\rightarrow \langle 1, \{(\texttt{d},0),(\texttt{ck},1),(\texttt{iq},0),(\texttt{ckb},0)\}, \{(\texttt{d},0),(\texttt{ck},1),(\texttt{iq},0),(\texttt{ckb},0)\}, \emptyset, \{(\texttt{qint},\mathsf{X})\}\rangle_U$$

Updating value of `qint` which activates no instantiations

$$\rightarrow \langle 1, \{(\texttt{d},0),(\texttt{ck},1),(\texttt{iq},0),(\texttt{ckb},0)\}, \{(\texttt{d},0),(\texttt{ck},1),(\texttt{iq},0),(\texttt{ckb},0),(\texttt{qint},\mathsf{X})\}, \emptyset, \emptyset\rangle_U$$

Time Advance, applying inputs $\overrightarrow{in_2}$, and activating instantiations

$$\rightarrow \langle 2, \{(\texttt{d},0),(\texttt{ck},1),(\texttt{iq},0),(\texttt{ckb},0),(\texttt{qint},\mathsf{X})\}, \{(\texttt{d},1),(\texttt{ck},1),(\texttt{iq},0),(\texttt{ckb},0),(\texttt{qint},\mathsf{X})\}, \{\texttt{l(iq,d,ck,rb)}\}, \emptyset\rangle_E$$

$$\rightarrow \ldots$$

$$\rightarrow \langle 2, \{(\texttt{d},1),(\texttt{ck},1),(\texttt{iq},\mathsf{X}),(\texttt{ckb},0),(\texttt{qint},\mathsf{X})\}, \{(\texttt{d},1),(\texttt{ck},1),(\texttt{iq},\mathsf{X}),(\texttt{ckb},0),(\texttt{qint},\mathsf{X})\}, \emptyset, \emptyset\rangle_U$$

This simulation terminates in the last configuration, since only two input vectors were given. The observed trace of values is given by variable valuations representing the current values in the two stable configurations that were reached, which are those configurations where the time-advance rule is applicable (or no more inputs are available). Thus, the observed trace of the above simulation consists of the three valuations $\emptyset$, $\{(\texttt{d},0),(\texttt{ck},1),(\texttt{iq},0),(\texttt{ckb},0),(\texttt{qint},\mathsf{X})\}$, and $\{(\texttt{d},1),(\texttt{ck},1),(\texttt{iq},\mathsf{X}),(\texttt{ckb},0),(\texttt{qint},\mathsf{X})\}$.

Also the simulation semantics allows non-determinism, namely the choice of the primitive instance to be executed next in rule (ex) of Table 3.2. The rule (ex-up) however, that switches from the Execute into the Update phase, is only applicable when all instances have been executed. Furthermore, the results of executing the instances are not directly written back into the state, but instead they are collected in the update variable valuation. Because it is assumed that every variable is used as output of at most one primitive, this valuation is the same regardless of the order of executing primitives. This restriction could also be removed. The Verilog standard defines that in case of multiple primitives providing a value for a single variable, a resolution function should be applied. This resolution function should therefore be included in the rule (ex) of Table 3.2. All resolution functions are combinational, thus including them would still create the same update valuation regardless of the order of execution.

Hence, the non-determinism in choosing a primitive instance to execute next does not have an influence on the computed results. For this reason, the presented VeriCell semantics does not define an order. A simulator implementing this semantics may simply choose one order of executing primitive instances without affecting the resulting trace of stable values.

3.2 Encoding VeriCell into Boolean Transition Systems

To convert a VeriCell program into a Boolean Transition System (BTS), i.e., a transition system with vectors of Booleans as configuration, first the ternary values have to be represented as Booleans. For this purpose, a *dual-rail* encoding of these values as in [Bry87] is used, where each variable v is interpreted as a pair of Boolean variables (v_L, v_H). The first variable v_L represents whether the variable v might be 0, the second variable v_H represents whether v might be 1. Then, for the constants in $\mathbb{T}$ it holds that that $0 = (\mathsf{true}, \mathsf{false})$, $1 = (\mathsf{false}, \mathsf{true})$, and $\mathsf{X} = (\mathsf{true}, \mathsf{true})$. The fourth value $\mathsf{Z} = (\mathsf{false}, \mathsf{false})$ is considered to be illegal and will never arise as a result of the encoding presented in this section. For every variable v occurring in the module of the VeriCell program, another pair of variables $v^p = (v_L^p, v_H^p)$ is introduced, which represents the previous value of the variable. These are required to detect edges, i.e., certain transitions from a specified old value to a new value.

To combine Boolean and ternary values, the implication operation $\rightarrow: \mathbb{B} \times \mathbb{T} \rightarrow \mathbb{T}$ is defined for a ternary value $y \in \mathbb{T}$ as $\mathsf{false} \rightarrow y = \mathsf{X}$ and $\mathsf{true} \rightarrow y = y$. Furthermore, a meet operator $\sqcap : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T}$ is defined as $(u_L, u_H) \sqcap (v_L, v_H) = (u_L \wedge v_L, u_H \wedge v_H)$ and (in)equality of ternary values is defined as $(u_L, u_H) \neq (v_L, v_H) = \neg\big((u_L, u_H) = (v_L, v_H)\big) = (u_L \oplus v_L) \vee (u_H \oplus v_H)$ for all ternary values $(u_L, u_H), (v_L, v_H) \in \mathbb{T}$ (where $\oplus$ denotes the Boolean xor operation, defined as $\mathsf{false} \oplus x = x$ and $\mathsf{true} \oplus x = \neg x$ for $x \in \mathbb{B}$). Note that by this definition the value X is a neutral element for $\sqcap$, i.e., $\mathsf{X} \sqcap y = y$ for all ternary values y.

Encoding of UDPs

To encode the UDPs of a VeriCell program, first the matching of row patterns has to be encoded. This can be done in a straightforward way. For level specifications, an encoding to match a ternary variable $v = (v_L, v_H)$ is defined as follows:

$$\begin{array}{rcrlrcl} L(\mathtt{0}, v) & \doteq & v_L \wedge & \neg v_H & L(\mathtt{b}, v) & \doteq & v_L \oplus v_H \\ L(\mathtt{1}, v) & \doteq & \neg v_L \wedge & v_H & L(\mathtt{?}, v) & \doteq & \mathsf{true} \\ L(\mathtt{x}, v) & \doteq & v_L \wedge & v_H & & & \end{array}$$

For edges, only the case of an edge specification of the form `(ab)`, with `a` and `b` being level specifications, has to be considered since all other edge specifications can be expressed as disjunctions of edge specifications having this shape, cf. Table 3.1. The encoding works by simply matching the two level specifications against the previous and the current value of the input. Furthermore, as discussed already in Section 2.3, it has to be expressed that really a change has occurred. This can be encoded by requiring the previous and the current value to be different. Putting this together, the encoding of an edge specification for pairs v^p, v of ternary variables representing the previous and the current values, respectively, is defined as follows:

$$E(\mathtt{(ab)}, v^p, v) \;\doteq\; (v^p \neq v) \wedge L(\mathtt{a}, v^p) \wedge L(\mathtt{b}, v)$$

These encodings are then used to encode the evaluation of rows contained in a UDP. If $r = s_1 \ldots s_n : s_{n+1} : s_{n+2}$ is a row from a UDP with n inputs, let $r|_j$ denote the j-th column of this row, i.e., $r|_j = s_j$ for all $1 \leq j \leq n+2$. For such a row r, the encoding P is true if and only if the row matches when considering a changed input value at some position $1 \leq j \leq n$. In case r is a level sensitive row, then P is defined as follows, where o is the previous output value, and where $\overrightarrow{i^p} = (i_1^p, \ldots, i_n^p)$

and $\overrightarrow{i} = (i_1, \ldots, i_n)$ are the previous and current inputs, respectively:

$$P(r, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \doteq \bigwedge_{\substack{1 \leq m \leq n \\ m \neq j}} L(r|_m, i^p_m) \wedge L(r|_j, i_j) \wedge L(r|_{n+1}, o)$$

For an edge-sensitive row r, the encoding $P(r, o, \overrightarrow{i^p}, \overrightarrow{i}, j)$ is defined similarly, only there $L(r|_j, i_j)$ has to be replaced by $E(r|_j, i^p_j, i_j)$.

Using the encoding P to determine whether a row is applicable, another encoding Row can be defined. The purpose of this encoding is to determine an output value w.r.t. some row, given the previous and current input values and the previous output value. Here, the property holds that the output of this encoding is X in case the row is not applicable. Again, the number $1 \leq j \leq n$ denotes the position where a change in input values is currently being considered.

$$\mathrm{Row}(r, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \doteq P(r, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \rightarrow O(r, o)$$

Above, the value $O(r, o)$ is the corresponding ternary value to the level specification in the last column of the row, or it is o if the last column of the row contains the symbol "–". As an example, this encoding is applied to the first row of the UDP `latch` in line 14 of Figure 3.1, where a change in the input `ck` is considered (i.e., $j = 2$):

$$\begin{aligned}
&\mathrm{Row}(\texttt{0 (?1) ? : ? : 0}, \texttt{iq}, (\texttt{d}^p, \texttt{ck}^p, \texttt{rb}^p), (\texttt{d}, \texttt{ck}, \texttt{rb}), 2) = \\
&\qquad \texttt{d}^p_L \wedge \neg\texttt{d}^p_H \wedge ((\texttt{ck}^p_L \oplus \texttt{ck}_L) \vee (\texttt{ck}^p_H \oplus \texttt{ck}_H)) \wedge \neg\texttt{ck}_L \wedge \texttt{ck}_H \rightarrow (1, 0)
\end{aligned}$$

In Section 3.1 it was noted that the order of evaluating multiple changed inputs is not fixed by the standard. Hence, there the order was a parameter of the semantics. Experience shows that a naive enumeration of all possible orders immediately results in an intractable state-space. Therefore, in this section, the order of evaluating changed input coordinates is fixed to be the reverse order of inputs as given in the definition of a UDP. This corresponds to the behavior exhibited by most of the considered simulators. In Chapter 4 (more precisely, in Section 4.1), analysis techniques will be presented that guarantee independence of the outcome from the concrete evaluation order. Usually this independence is desired, as otherwise the computation result depends on the non-deterministic choices.

The idea of the UDP encoding is to check recursively for a changed input at the current position of the input. If the currently considered input has changed, then the previous output is updated to the new output and the recursion continues for the next position. Otherwise, when the current input is unchanged, then also the output value remains unchanged and the recursion directly advances to the next position.

Formally, given an instance of a UDP $\texttt{udp} \in \mathrm{UDPs}_n$ with output variable o, previous inputs $\overrightarrow{i^p} = (i^p_1, \ldots, i^p_n)$, and current inputs $\overrightarrow{i} = (i_1, \ldots, i_n)$, the encoding $[\![\texttt{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}]\!]_{\mathbb{B}\times\mathbb{B}} = [\![\texttt{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}, n]\!]_{\mathbb{B}\times\mathbb{B}}$ is defined as follows for $1 \leq j \leq n$:

$$\begin{aligned}
[\![\texttt{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}, 0]\!]_{\mathbb{B}\times\mathbb{B}} &\doteq o \\
[\![\texttt{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}, j]\!]_{\mathbb{B}\times\mathbb{B}} &\doteq \Big((i^p_j = i_j) \rightarrow [\![\texttt{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}, j-1]\!]_{\mathbb{B}\times\mathbb{B}}\Big) \\
&\quad \sqcap \Big((i^p_j \neq i_j) \rightarrow [\![\texttt{udp}, o', \overrightarrow{i^p}', \overrightarrow{i}, j-1]\!]_{\mathbb{B}\times\mathbb{B}}\Big)
\end{aligned}$$

In the previous values $\overrightarrow{i^p}'$ the changed value i_j replaces the previous value i^p_j, i.e., $\overrightarrow{i^p}' = (i^p_1, \ldots, i_j, \ldots, i^p_n)$. Furthermore, the value of o' is the corresponding value that results from the UDP when considering the change in input j. For this purpose, let $rl_1, \ldots, rl_{k_l}$ be all level-sensitive rows of the UDP and let $re_{1,j}, \ldots, re_{k_e,j}$ be all edge-sensitive rows of the UDP that have an edge specification in column j. Then, the value of o' is defined to be the following:

$$\begin{aligned} o' \doteq & \bigsqcap_{1 \le j_l \le k_l} \mathrm{Row}(rl_{j_l}, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \\ & \sqcap \left[\Big(\neg \bigvee_{1 \le j_l \le k_l} P(p(rl_{j_l}), o, \overrightarrow{i^p}, \overrightarrow{i}, j) \Big) \rightarrow \bigsqcap_{1 \le j_e \le k_e} \mathrm{Row}(re_{j_e,j}, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \right] \end{aligned}$$

Since this definition consists of meets of implications, the output will be X, i.e., $(\mathsf{true}, \mathsf{true})$, if none of the rows is applicable. This is as required in the standard.

Encoding of Cells

To encode the behavior of a module representing a cell, one could encode the simulation rules given in Section 3.1. However, a much simpler encoding is possible when restricting to VeriCell programs that do not contain delays. Under this assumption, the new values can directly be written back into the current variables, when the equations that result from the encoding are evaluated in parallel. This leads to an encoding which only needs the two dual-rail pairs $prev_j$ and cur_j for all of the $n+m$ variables of the module (which is assumed to contain n primitive instantiations and m external inputs) and furthermore m dual-rail pairs inp_j for $n < j \le n+m$ that represent the external inputs to the module.

The next value for a variable $prev_j$ simply has to copy the current value of the variable, so that it represent the previous value in the next iteration. Hence, for all $1 \le j \le n+m$ the new value of $prev_j$, which is denoted with a prime, is defined as follows:

$$prev'_j \doteq cur_j$$

For the current variables, two cases have to be distinguished. In case the variable cur_j is an output of a primitive instantiation p_j (which are assumed to be the first n variables), i.e., if $1 \le j \le n$, then the following definition is used.

$$cur'_j \doteq [\![p_j, cur_j, prev(in(p_j)), cur(in(p_j))]\!]_{\mathbb{B}\times\mathbb{B}}$$

In this definition, it is assumed that $prev(in(p_j))$ and $cur(in(p_j))$ are vectors of ternary variables and constants that represent the previous and current values of the inputs to p_j, respectively.

The second case computes the new value of an input. New input values are only applied when the current state is stable and time is allowed to advance. To detect this, a formula $time\_may\_advance$ is defined that is true if the current state is stable. As defined in Table 3.2, a state is stable if there are no more active primitives and no more updates that have to be executed. In the presented encoding, updates directly take place in the current variables. Therefore, only active primitive instantiations have to be detected. There are no more active instantiations if there are no changed values, i.e., the previous and the current values are the same.

$$time\_may\_advance \doteq \bigwedge_{1 \le j \le n+m} prev_j = cur_j$$

Using this formula, the update of the inputs can be encoded. As stated before, the inputs may only be updated if the current state is stable, otherwise the old value has to be kept. This is formalized in the definition below, where $n < j \leq n + m$:

$$\begin{array}{rcl} cur'_j & \doteq & \neg time\_may\_advance \rightarrow cur_j \\ & \sqcap & time\_may\_advance \rightarrow [inp_j]_{\mathbb{T}} \end{array}$$

In the above definition, $[inp_j]_{\mathbb{T}}$ maps the illegal pair $(\mathsf{false}, \mathsf{false})$ to the pair $(\mathsf{true}, \mathsf{true})$, representing that the value Z behaves like the value X in the VeriCell subset of Verilog. Formally, it is defined as $[v]_{\mathbb{T}} \doteq (v_L \vee \neg(v_L \wedge v_H), v_H \vee \neg(v_L \wedge v_H))$ for every dual-rail pair $v = (v_L, v_H)$. In this way, $[(\mathsf{false}, \mathsf{false})]_{\mathbb{T}} = (\mathsf{true}, \mathsf{true}) = \mathsf{X}$ and $[y]_{\mathbb{T}} = y$ for every $y \in \mathbb{T}$.

The presented encoding implements the decution rules shown in Table 3.2. To see this, one can identify the values in the next iteration, denoted with a prime above, and the set of updates up in the rules. These updates are only applied when all primitives have been evaluated. Furthermore, the explicit set of active primitives is not needed, since a primitive that is evaluated twice with the same values for its inputs will always return the same output value. This holds for built-in primitives since these are combinational (i.e., they implement a function of the input values only), but also for UDPs since they simply output their currently stored value if no input has changed (thus, in a second evaluation with the same input vector, the previously computed value is returned). Finally, detecting that the simulation time may be advanced is performed by determining whether all signals in the cell have become stable. If this is the case, then there cannot be any active primitives. Otherwise, another computation implementing the execute-update cycle of the rules in Table 3.2 is performed. Note that if those rules eventually apply the time advance rule, then this will also be the case in the above encoding: If there are no active primitives anymore, then the only signals that might still be different are those that are not used as input to any other primitive. Therefore, evaluating the presented encoding for one more iteration will set the previous values to the current values, hence the time can be advanced in this iteration by applying new input values.

3.3 Equivalence Checking VeriCell and Transistor Netlist Descriptions

The encoding presented in the previous section is used to translate Verilog descriptions into the input language of a symbolic model checker. The target application of this encoding is equivalence checking between the Verilog descriptions and the transistor netlist descriptions contained in a cell library, which is described in this section. To automatically create a transition system from a transistor netlist, standard techniques exist [Bry87, PJB$^+$95]. These will result directly in a next state function, i.e., applying the transition function to a state results in the next stable state.

This is different for the Boolean Transition Systems created from VeriCell descriptions: Here, a stable state only exists if time may advance. Therefore, the property to be verified is that for all stable states the outputs are equal. Furthermore, the comparison of the outputs should be restricted to those outputs different from X, since the value X is regarded as a “don't-care”. Finally, the model checking approach should support the assumption that at most one external input changes in every step of the global simulation time. This assumption, together with some more conditions on the usage of variables in the cell, can ensure deterministic computations. This assumption however will be refined later when the *timing checks* are considered in Chapter 5. Since not changing the inputs will not trigger any change in a current stable state, it is equivalent to require that exactly one input is changing every time the simulation time advances.

The LTL formula asserting that netlist and Verilog descriptions are equivalent, i.e., both implement the same functional behavior for non-X values, is then expressed as follows, where *outputs* is a set of corresponding pairs of outputs in the Verilog description and the transistor netlist:

$$\begin{aligned}&\big(\mathsf{G}\ \mathit{one\_input\_changes}\big) \rightarrow\\ &\quad\Big(\mathsf{G}\ \mathit{time\_may\_advance} \rightarrow\\ &\qquad\Big(\bigwedge_{(o_v,o_t)\in \mathit{outputs}} o_v \neq \mathsf{X} \wedge o_t \neq \mathsf{X} \rightarrow o_v = o_t\Big)\Big)\end{aligned}$$

The formula expressing that exactly one input changes per timestep, called *one_input_changes* above, is expressed as shown below:

$$\begin{aligned}&\mathit{one\_input\_changes} \doteq\\ &\bigvee_{n<j\leq n+m}\Big(([\mathit{inp}_j]_{\mathbb{T}} \neq \mathit{cur}_j) \wedge \bigwedge_{\substack{n<i\leq n+m\\ i\neq j}} ([\mathit{inp}_i]_{\mathbb{T}} = \mathit{cur}_i)\Big)\end{aligned}$$

Finally, also a property should be added stating that a stable state will always eventually be reached. This can be expressed as an LTL formula in the following way, again using the formula *time_may_advance* to indicate whether a current state is stable:

$$\big(\mathsf{G}\ \mathit{one\_input\_changes}\big) \rightarrow \big(\mathsf{G}\ \mathsf{F}\ \mathit{time\_may\_advance}\big)$$

In case the restriction to input traces where only one input is allowed to change per time step is not desired, the requirement $(\mathsf{G}\ \mathit{one\_input\_changes})$ can be removed from the above formulas.

3.4 Experimental Results

As an example set, the Nangate Open Cell Library [Nan08] was chosen due to its public availability and the contained cells were checked for equivalence. For this purpose, an optimized version of the encoding given in Section 3.2 (where some common subterms were identified and constants propagated) was used to create Boolean Transition Systems from the VeriCell descriptions, and a simplified implementation of [Bry87] was used to extract a transition system from the transistor netlist descriptions. These automatically generated transition systems were then used as input, together with the LTL formulas as described in the previous section, for the symbolic model checkers NuSMV [CCG+02] version 2.4.3 and Cadence SMV [McM97] version 10-11-02p46. NuSMV was run with cone-of-influence reduction and dynamic reordering, whereas Cadence SMV was used in its default settings. All of these experiments were performed on a Pentium 4 with 3 GHz having 1 GB of RAM and running Linux.

Almost all cells of the Open Cell Library are described in the VeriCell language, except for the cells `TBUF`, `TINV`, and `TLAT` which use the primitive **`bufif0`**. This primitive is currently not supported, since it distinguishes between X and Z and has non-deterministic behavior for certain input combinations. Also, the cells `ANTENNA` and `FILLCELL` were omitted, since they do not implement any functionality. Finally, the cells `LOGIC0` and `LOGIC1`, which are supposed to provide constant values, were not considered, as no transition system could be extracted from the transistor netlists given in the cell library.

All considered cells in the Nangate Open Cell Library can be shown equivalent when only changing one input in every time step. Of these 43 cells in the library that were considered, 42 were shown equivalent using Cadence SMV in less than one second. For the last cell, namely the cell `SDFFRS`, it took about 2.0 seconds to prove equivalence. When using NuSMV, the results are slightly worse, only 38 of the cells were shown equivalent within one second and the proofs of another 4 cells took between 2 and 4 seconds. Again, the cell `SDFFRS` took the most time with 7.8 seconds. Optimizing the desired properties specifically for NuSMV, by converting the equivalence property into an invariant checking problem and by creating a CTL formula for the property that always eventually a stable state is reached, then 40 cells were shown equivalent within one second and 2 of the remaining 3 cells were shown equivalent within 2 seconds. However, the cell `SDFFRS` still required approximately 4.1 seconds.

Another experiment conducted was to compare the flip-flop example given in Figure 3.1 with the cell `DFFR` taken from the Nangate Open Cell Library, which also is a flip-flop with reset. However, using the presented encoding an error trace could be found by both considered model checkers. Such a trace can easily be mapped back to a counterexample in the two cells. This counterexample shows that for the Nangate cell the input already becomes visible at the output at the positive edge of the clock, whereas the flip-flop from Figure 3.1 requires another negative edge (in fact, it implements a flip-flop that is clocking in a data value on a falling edge of the clock). Hence, these two flip-flops are not equivalent and must not be exchanged for one another.

Finally, experiments were run with a subset of sequential cells taken from an industrial cell-library provided by Fenix Design Automation. For all of these 26 cells, equivalence could be proven when considering only one input to change in every time step. Using Cadence SMV, 12 of these cells could be shown equivalent in less than one second. Of the remaining 14 cells, 9 took less than 2 seconds and the remaining 5 cells took between 3 and 15 seconds. For NuSMV the results are comparable when using the optimized encoding of the properties: 12 cells took less than one second, 7 of the remaining 14 cells took less than 2 seconds, and the remaining 7 cells took between 2 and 15.5 seconds.

3.5 Summary

This chapter presented a semantics for the VeriCell subset of Verilog that is commonly used in cell libraries. Thereby, VeriCell descriptions can automatically be converted into Boolean transition systems. These transition systems can then be used for model checking, of which one application is the equivalence checking between Verilog descriptions and corresponding transistor netlists contained in a cell library. This check runs fully automatically and requires no user intervention. Thereby, one can ensure that the Verilog description exhibits the same behavior as the implementation in silicon.

In the VeriCell subset of Verilog, the values X and Z have the same behavior, as required by the standard. However, as already described in Section 3.4, there are also cells using primitives such as **`bufif0`**, which can distinguish these two values. Thus, it would be interesting to extend the VeriCell subset to also include these built-in primitives. Including these primitives is not straightforward, as they introduce another source of non-determinism: the standard requires for certain input combinations that the output of **`bufif0`** can be either 0 or Z, for example. Such non-determinism cannot be ruled out by the timing checks presented in Chapter 5, thus also the encoding would have to become non-deterministic.

Another possible extension of the presented encoding are delayed updates, which do not cause a new value to be updated directly, but only after some specified amount of simulation time has passed. Incorporating delays into the formal semantics is straightforward, however these rules have to be encoded in the transition systems as well. The problem here is that the simulation time should not be represented in the encoding, since it would otherwise make the state space infinite. A possible solution would be to use variables counting down the number of time advance steps required before an update may take place. The practical relevance of delays seems low, as none of the cells considered during the experiments contained such delays. Hence, an implementation of this extension is not justified by the considered practical examples.

Equivalence of a VeriCell and a transistor netlist description was considered to hold when non-X values are equal. There are however other notions of equivalence that could be considered. For example, it could be the case that only during a "power-up" phase the behavior is different, which corresponds to the *alignability equivalence* of Pixley [Pix92]. However, these types of counterexamples were not observed in the considered examples.

Finally, Section 3.1 showed that the order of considering changed inputs of UDPs can influence the output of a UDP. When only using the fixed order of most simulators, which was used for the encoding in Section 3.2, then it can prevent finding counterexamples, i.e., situations where VeriCell description and transistor netlist description behave differently. This is especially interesting since bugs that stem from such an order dependence may not be found using simulators that use a fixed order. In the next chapter of this thesis, an efficient analysis technique is presented that addresses this problem. If order-independence of the UDP holds, then it is safe to only encode the simulator order. Otherwise, the internal state of the cell is not uniquely determined, which is undesired and should be removed. Such a removal is often possible by means of *timing checks* that impose restrictions on the environment of a cell. These will be discussed later in Chapter 5.

Chapter 4

Efficient Analysis of Non-Determinism in Cell Libraries

Non-determinism in some functional behavior allows to make arbitrary choices during a computation. This can have an effect on the resulting values, so that multiple possible behaviors arise, which is generally undesired. In Section 3.1 of the previous chapter it was illustrated that even in basic structures such as cells, non-determinism exists that can influence the values that are computed. This non-determinism is not accounted for in simulators; they just use an arbitrarily fixed resolution of the non-determinism to obtain a single execution trace. Thereby, many plausible behaviors may be hidden during simulation, only to reappear in the final layout and thereby jeopardizing correctness of the overall design.

An exhaustive search over all possible execution paths could theoretically solve this problem, ensuring that the behavior is equivalent regardless of the non-deterministic choices made. In practice, however, such a naive approach often leads to an intractable (symbolic) state space. To alleviate this problem, two main techniques are used: language-based restrictions to rule out irrelevant or impossible executions and reduction techniques, which only consider a fraction of the overall state space but still provide a sound result. In this chapter, techniques of the latter type, i.e., the reduction to a smaller problem, are presented. The main inspiration for the presented techniques stem from confluence checking techniques in term rewriting, see for example [BN98, Ter03] for an introduction. Extending the presented techniques by language-based restrictions in the form of Verilog timing checks will be considered later in Chapter 5.

In Section 4.1, which is based on [RMR$^+$09], the problem observed in the previous chapter, viz. the possible order-dependence of UDPs, will be treated. However, to keep the technique as general as possible, the exact UDP semantics as was presented in Section 3.1 is generalized. Only two properties of the UDP output computation are being used, which therefore eases adoption for different languages having a similar overall structure of the computation.

The non-determinism of UDPs depends on the exact order of treating changed inputs. This reflects that inputs do not change exactly simultaneously, but with a small, non-deterministic delay relative to each other. Thus, also for transistor netlists, the computations might lead to different outcomes if different orders of applying changed inputs are used. This non-determinism is considered by the analysis techniques in Section 4.2, first presented in [RMZ10] and later extended in [RMZ11]. Again, for these techniques the exact details of transistor netlists are abstracted away as much as possible, to enable more general applicability. To this end, transistor netlists are viewed as a transition system with *vectors* of inputs changing one coordinate at a time. Transistor netlists do not have an initial state, instead a computation may start in any state. When requiring order-independence for all states, then also transient initial states, i.e., states that only can be part of an initial computation but can never be reached again, are considered. However, these initial states often lead to order-dependence, since the transistor netlist has not yet been initialized correctly. Therefore, the analysis presented in Section 4.2 also includes a way to disregard these spurious order-dependencies.

Non-determinism leading to different computation results is undesired, as discussed above. However, non-determinism that does not affect the functional behavior can be useful, since it adds a "dont-care" dimension: Among the different possible computations, one can choose the computation that is optimal with respect to some other design goal. In Section 4.3, which is based on [RM11], power consumption is considered as such a design goal to optimize. In case computations exist that behave equivalently, but there is one computation that consumes less power, then one should change the design to force this computation to be chosen. Thereby, functionality of the design is not affected, but power consumption is reduced.

Another application area where non-determinism that does not affect the functional behavior can be put to use is power characterization, which is also described in Section 4.3. To determine the power consumption of a cell, multiple input patterns have to be simulated and the power consumption has to be measured for each. To reduce the number of input patterns required, the non-determinism analysis, that detects equivalent behaviors, is extended to also take abstract power consumption into account. Then, for different input patterns that always consume the same amount of abstract power, it is sufficient to only measure one of these equivalent patterns, since the others will exhibit the same power consumption.

4.1 Order-Independence of VeriCell Descriptions

Preliminaries

The representation in Section 3.1 of the permutations Π_n as a special instance of the lists $\mathcal{L}_n$, which contain numbers between 1 and n without duplicates, will also be used in this section. Additionally, the well known representation of permutations as the composition of *adjacent transpositions* will be used. A transposition is a permutation $(a\ b) \in \Pi_n$ and is defined as $(a\ b)\,(j) = j$ for all $1 \leq j \leq n$, $j \notin \{a, b\}$, $(a\ b)\,(a) = b$, and $(a\ b)\,(b) = a$. Such a transposition is called adjacent if $b = a + 1$. Composition of permutations will be denoted by juxtaposition, i.e., $(\pi_1\ \pi_2)\,(x) = \pi_1(\pi_2(x))$.

The semantics of UDPs as presented in Section 3.1 will be used here, too. However, the full formal definition is not required in this section, and can even be generalized. The previous and current input values are still represented as a vector of pairs of ternary values, i.e., $I_{\texttt{udp}} = \big(\mathbb{T} \times \mathbb{T}\big)^n$ for a UDP `udp` with n inputs. In the remainder, the subscript `udp` will be dropped whenever the UDP is clear from the context. Accessing a certain element of a vector $\vec{i} = \big((i_1^p, i_1), \ldots, (i_n^p, i_n)\big) \in I$ is done by *projection* functions: For $1 \leq j \leq n$, $\rho_j^p(\vec{i}) = i_j^p$ and $\rho_j(\vec{i}) = i_j$. To modify such a vector, *substitutions* are defined. A substitution is denoted by $\sigma = [a_1^p :=$

$v_1, \dots, a_k^p := v_k, b_1 := w_1, \dots, b_l := w_l]$, where $a_1, \dots, a_k, b_1, \dots, b_l \in \{1, \dots, n\}$, $v_1, \dots, v_k, w_1, \dots, w_l \in \mathbb{T}$, and it holds that if $a_i = a_j$ or $b_i = b_j$, then $i = j$. Applying a substitution to a vector is denoted $\vec{i}\sigma$ and is defined as $\rho_j^p(\vec{i}\sigma) = v_j$ if $j^p := v_j \in \sigma$, $\rho_j(\vec{i}\sigma) = w_j$ if $j := w_j \in \sigma$, and $\rho_j^p(\vec{i}\sigma) = \rho_j^p(\vec{i})$, $\rho_j(\vec{i}\sigma) = \rho_j(\vec{i})$ in the respective other cases.

The computation of the next output value is abstracted into functions $\Phi_j : I \times \mathbb{T} \to \mathbb{T}$ for $1 \leq j \leq n$. These functions consider the j-th input as changed and take into account the precedence of level-sensitive over edge-sensitive rows, as described in Section 3.1. This level of detail is not required here. Instead, only two properties about these functions are required, which are given next. The first requirement is that an unchanged j-th input implies that the output value is unchanged, too. This property clearly holds for the concrete semantics defined in Section 3.1.

Property 4.1.1. Let $1 \leq j \leq n$ and let $\vec{i} \in I$ such that $\rho_j^p(\vec{i}) = \rho_j(\vec{i})$. Then for all $o^p \in \mathbb{T}$, $\Phi_j(\vec{i}, o^p) = o^p$.

The second requirement on these functions is that they must only consider the previous output values, except for the position where the change is being considered. Intuitively, this expresses that the other changes have not yet occurred, thus they are not visible yet, or they have already occurred, hence the previous value has already been updated. This of course also holds for the concrete semantics defined for UDPs in Section 3.1, since in the definition of $[\![\cdot]\!]$ the predicate $\mathrm{matchRow}$ is only used with the previous input values, except for coordinate j. To make this requirement formal, it is required that a change on any other position does not affect the computation.

Property 4.1.2. Let $1 \leq j \leq n$, let $1 \leq k \leq n$ with $k \neq j$, let $\vec{i} \in I$, and let $v, o^p \in \mathbb{T}$. Then $\Phi_j(\vec{i}, o^p) = \Phi_j(\vec{i}[k := v], o^p)$.

Using the abstract functions Φ_j and substitutions to update the input values, the evaluation function $[\![\cdot]\!] : \mathrm{UDPs}_n \times I \times \mathbb{T} \times \mathcal{L}_n \to \mathbb{T}$ from Section 3.1 is defined as follows:

$$\begin{array}{lcl} [\![\mathtt{udp}, \vec{i}, o^p, \mathsf{nil}\ ]\!] & \doteq & o^p \\ [\![\mathtt{udp}, \vec{i}, o^p, j : \ell]\!] & \doteq & [\![\mathtt{udp}, \vec{i}[j^p := \rho_j(\vec{i})], \Phi_j(\vec{i}, o^p), \ell]\!] \end{array}$$

In the remainder, the considered UDP will often be clear from the context. Hence, in these cases the argument $\mathtt{udp}$ will be dropped, thus instead of $[\![\mathtt{udp}, \vec{i}, o^p, \ell]\!]$ the notation $[\![\vec{i}, o^p, \ell]\!]$ will be used.

Order Dependency Analysis

A UDP is said to be order-dependent, if there are two orders of evaluating it that lead to different results. Otherwise, if the order of evaluation does not affect the final outcome, it is said to be order-independent. Below, this intuitive property is made formal.

Definition 4.1.3. A UDP with n inputs is called *order-independent*, iff $[\![\vec{i}, o^p, \pi]\!] = [\![\vec{i}, o^p, \pi']\!]$ for all $\vec{i} \in I$, $o^p \in \mathbb{T}$, and $\pi, \pi' \in \Pi_n$. Otherwise, the UDP is called *order-dependent*.

```
 1  primitive ff_en(q, d, ck, en);
 2    output q; reg q;
 3    input d, ck, en;
 4
 5    table
 6      //  d  ck   en  : q : q+
 7          0 (01)  1   : ? : 0;
 8          1 (01)  1   : ? : 1;
 9          ? (10)  ?   : ? : -;
10          *  ?    ?   : ? : -;
11          ?  ?    0   : ? : -;
12          ?  ?    *   : ? : -;
13    endtable
14  endprimitive
```

Figure 4.1: UDP implementing a D Flip-Flop with Enable

This definition is illustrated next by means of an example. Consider the UDP `ff_en` shown in Figure 4.1. This UDP is order-dependent, which can be seen for $\vec{\imath} = \big((0,1),\ (0,1),\ (1,1)\big)$ (i.e., both inputs `ck` and `d` change from 0 to 1 while `en` stays constant 1), $o^p = \mathsf{X}$, $\pi = 1 : 2 : 3$, and $\pi' = 2 : 1 : 3$. For the order π, the change of input `d` is considered first, which does not change the output due to line 10. Afterwards, input `ck` changes its value, where now line 8 is applicable and sets the new output value to 1. This is also the final output value, since input `en` does not change its value. For the order π' however, the change of input `ck` is considered first. Here, still the old value of input `d` is visible, hence line 7 is used, which sets the output value to 0. The change of `d`, considered next, does not change this value, due to line 10. Finally, again the non-changing value of `en` does not affect the output value, so that 0 is the final output value for this order.

When only the relative behavior of the two inputs `d` and `en` is relevant, i.e., the value of input `ck` is considered stable, then the UDP `ff_en` of Figure 4.1 is order-independent, since it holds that $[\![\big((ck^p, ck^p), (d^p, d), (en^p, en)\big), o^p, \pi]\!] = [\![\big((ck^p, ck^p), (d^p, d), (en^p, en)\big), o^p, \pi']\!]$ for all values $ck^p, d^p, d, en^p, en, o^p \in \mathbb{T}$ and all orders $\pi, \pi' \in \Pi_n$. Naively, this can be checked by enumerating all possible pairs of orders π, π', but that is overly complicated. It is easy to see that a comparison with the *identity permutation* id, defined as $\mathrm{id}(j) = j$ for all $1 \leq j \leq n$, is sufficient.

Lemma 4.1.4. *A UDP with n inputs is order-independent iff $[\![\vec{\imath}, o^p, \pi]\!] = [\![\vec{\imath}, o^p, \mathrm{id}]\!]$ for all $\vec{\imath} \in I$, $o^p \in \mathbb{T}$, $\pi \in \Pi_n$.*

Proof. The "only if"-direction is a trivial consequence of Definition 4.1.3. The "if"-part follows from the transitivity of equality. □

Thus, instead of enumerating all pairs of orders, of which there are $O((n!)^2)$ many, all permutations have to be enumerated only once, giving a complexity of $O(n!)$ comparisons. As will be shown in the remainder of this section, this complexity can be reduced even further, to a quadratic number of comparisons. To this end, the *commuting diamond property* is introduced. Informally, this property expresses that the order of two given inputs does not influence the output.

$$\begin{tikzcd}
 & \vec{i}, o^p \arrow[dl, "j"'] \arrow[dr, "k"] & \\
\vec{i}[i^p := \rho_j(\vec{i})], o_j \arrow[dr, dashed, "k"'] & & \vec{i}[j^p := \rho_k(\vec{i})], o_k \arrow[dl, dashed, "j"] \\
 & \vec{i}[i^p := \rho_j(\vec{i}), j^p := \rho_k(\vec{i})], o &
\end{tikzcd}$$

Figure 4.2: Commuting Diamond Property

Definition 4.1.5. Let $udp \in \mathrm{UDPs}_n$.

Inputs $1 \leq j, k \leq n$, $j \neq k$ are said to have the *commuting diamond property*, denoted $j \diamond_{udp} k$, iff for all $\vec{i} \in I$, $o^p \in \mathbb{T}$:

$$[\![udp, \vec{i}, o^p, j : k]\!] = [\![udp, \vec{i}, o^p, k : j]\!]$$

The commuting diamond property is a well-known property from term rewriting, e.g., given in [BN98, Section 2.7.1]. The idea is that every two one-step rewrites (evaluations) can be joined again by executing the respective other step. Graphically, this is depicted in Figure 4.2, where only the inputs and the output value are denoted. The solid lines are universally quantified, whereas the dashed lines are existentially quantified.

Considering one-step evaluation as a rewrite step, the commuting diamond property implies confluence in the induced term-rewrite system, i.e., the final state (and hence especially the output) is unique regardless of the order of considering inputs.

This section proves the stronger result that, in the special case of UDP evaluation, the commuting diamond property and confluence *coincide*. This relies on the semantics of UDPs, or, more precisely, on Properties 4.1.1 and 4.1.2, and does not hold in the general setting of term rewriting. For sake of completeness, the proof of sufficiency of the commuting diamond property is also given below.

The formal definition of the commuting diamond property amounts to checking that in case of two simultaneous changes in the input, both orders of considering them lead to the same output. To put such evaluations into longer evaluations, where more elements exist in the list of input numbers to be considered, the following lemma shows that this does not change the behavior.

Lemma 4.1.6. *For a UDP with n inputs, $\vec{i} \in I$, $o^p \in \mathbb{T}$, and a list $\ell_1{+\!\!+}\ell_2 \in \mathcal{L}_n$ with $\ell_1 = k_1 : \ldots : k_{|\ell_1|}$,*

$$[\![\vec{i}, o^p, \ell_1{+\!\!+}\ell_2]\!] = [\![\vec{i}[k_r^p := \rho_{k_r}(\vec{i}) \mid 1 \leq r \leq |\ell_1|], [\![\vec{i}, o^p, \ell_1]\!], \ell_2]\!].$$

Proof. Induction is performed on $|\ell_1|$.

If $|\ell_1| = 0$, then $\ell_1 = \mathsf{nil}$, $[\![\vec{i}, o^p, \ell_1]\!] = o^p$, and $\vec{i}[k_r^p := \rho_{k_r}(\vec{i}) \mid 1 \leq r \leq |\ell_1|] = \vec{i}$. Hence, $[\![\vec{i}, o^p, \ell_1{+\!\!+}\ell_2]\!] = [\![\vec{i}, o^p, \ell_2]\!] = [\![\vec{i}, [\![\vec{i}, o^p, \ell_1]\!], \ell_2]\!]$.

Otherwise, let $|\ell_1| > 0$ and $\ell_1 = k_1 : \ell$ with $\ell = k_2 : \ldots : k_{|\ell_1|}$. Then $[\![\vec{i}, o^p, \ell_1]\!] = [\![\vec{i}, o^p, k_1 : \ell]\!] = [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell]\!]$ for $o' = \Phi_{k_1}(\vec{i}, o^p)$. Furthermore,

$[\![\vec{i}, o^p, \ell_1 +\!\!+ \ell_2]\!] = [\![\vec{i}, o^p, k_1 : \ell +\!\!+ \ell_2]\!] = [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell +\!\!+ \ell_2]\!]$. The induction hypothesis is applicable to ℓ, which proves the theorem:

$$\begin{array}{cl} & [\![\vec{i}, o^p, \ell_1 +\!\!+ \ell_2]\!] \\ = & [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell +\!\!+ \ell_2]\!] \\ \overset{\text{IH}}{=} & [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})][k_r^p := \rho_{k_r}(\vec{i}) \mid 2 \leq r \leq |\ell_1|], [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell]\!], \ell_2]\!] \\ = & [\![\vec{i}[k_r^p := \rho_{k_r}(\vec{i}) \mid 1 \leq r \leq |\ell_1|], [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell]\!], \ell_2]\!] \\ = & [\![\vec{i}[k_r^p := \rho_{k_r}(\vec{i}) \mid 1 \leq r \leq |\ell_1|], [\![\vec{i}, o^p, \ell_1]\!], \ell_2]\!] \end{array}$$

□

The main technical lemma, given next, states that the commuting diamond property is a necessary and sufficient condition to be able to swap the order of two inputs in a longer computation.

Lemma 4.1.7. *Consider a UDP with n inputs and let $j : k : \ell \in \mathcal{L}_n$.*
Then $j \diamond k$ holds, iff for all $\vec{i} \in I$ and $o^p \in \mathbb{T}$, $[\![\vec{i}, o^p, j : k : \ell]\!] = [\![\vec{i}, o^p, k : j : \ell]\!]$.

Proof. In the "only if"-direction, the following two computations have to be considered:

$$\begin{array}{rcl} [\![\vec{i}, o^p, j : k : \ell]\!] & = & [\![\vec{i}[j^p := \rho_j(\vec{i}), k^p := \rho_k(\vec{i})], o\ , \ell]\!] \\ [\![\vec{i}, o^p, k : j : \ell]\!] & = & [\![\vec{i}[j^p := \rho_j(\vec{i}), k^p := \rho_k(\vec{i})], o', \ell]\!] \end{array}$$

Lemma 4.1.6 allows to split these computations, and because $i \diamond j$ holds by assumption, $o = [\![\vec{i}, o^p, j : k]\!] = [\![\vec{i}, o^p, k : j]\!] = o'$. Since the remaining computation is the same, the "only-if" direction has been proven.

To show the "if"-direction, let $j \not\diamond k$. Then $\vec{i} \in I$ and $o^p, o, o' \in \mathbb{T}$ exist such that:

$$o = [\![\vec{i}, o^p, j : k]\!] \neq [\![\vec{i}, o^p, k : j]\!] = o'$$

Define $\vec{i'} = \vec{i}[r := \rho_r^p(\vec{i}) \mid 1 \leq r \leq n, r \notin \{j, k\}]$, i.e., set all current values to their previous values except for those on positions j and k. Due to Property 4.1.2 it still holds that $o = [\![\vec{i'}, o^p, j : k]\!]$ and $o' = [\![\vec{i'}, o^p, k : j]\!]$. Applying Lemma 4.1.6 gives the following two evaluations:

$$\begin{array}{rcl} [\![\vec{i'}, o^p, j : k : \ell]\!] & = & [\![\vec{i'}[j^p := \rho_j(\vec{i}), k^p := \rho_k(\vec{i})], o\ , \ell]\!] \\ [\![\vec{i'}, o^p, k : j : \ell]\!] & = & [\![\vec{i'}[j^p := \rho_j(\vec{i}), k^p := \rho_k(\vec{i})], o', \ell]\!] \end{array}$$

By the requirements on lists in $\mathcal{L}_n$, all remaining elements in ℓ are neither j nor k. Formally, let $\ell = k_1 : \ldots : k_{|\ell|} : \mathsf{nil}$, then $k_r \notin \{j, k\}$ for all $1 \leq r \leq |\ell|$. Hence, $\rho_{k_r}^p(\vec{i'}) = \rho_{k_r}(\vec{i'})$ holds by construction of $\vec{i'}$ for all $1 \leq r \leq |\ell|$. This allows to repeatedly apply Property 4.1.1 to prove this lemma:

$$\begin{array}{rcl} [\![\vec{i'}, o^p, j : k : \ell]\!] & = & [\![\vec{i'}[j^p := \rho_j(\vec{i}), k^p := \rho_k(\vec{i})], o\ , \ell]\!] \\ & = & o \\ & \neq & o' \\ & = & [\![\vec{i'}[j^p := \rho_j(\vec{i}), k^p := \rho_k(\vec{i})], o', \ell]\!] \\ & = & [\![\vec{i'}, o^p, k : j : \ell]\!] \end{array}$$

□

The above lemma is used to prove the main theorem of this section, showing that order-independence is equivalent to all pairs of inputs having the commuting diamond property.

Theorem 4.1.8. *A UDP with n inputs is order-independent, iff for all pairs of numbers $1 \le j < k \le n$ it holds that $j \diamond k$.*

Proof. To show the "only if"-direction, let $j \not\diamond k$. Define list $\ell = 1 : \ldots : j-1 : j+1 : \ldots : k-1 : k+1 : \ldots : n :$ nil. Then by construction both $\pi = j : k : \ell \in \Pi_n$ and $\pi' = k : j : \ell \in \Pi_n$. Lemma 4.1.7 shows that $\vec{i} \in I$ and $o^p \in \mathbb{T}$ exist such that $[\![\vec{i}, o^p, j : k : \ell]\!] \neq [\![\vec{i}, o^p, k : j : \ell]\!]$, which proves that the UDP is order-dependent.

To show the "if"-direction, assume that $j \diamond k$ for all $1 \le j < k \le n$. Let $\pi \in \Pi_n$ with $\pi = (a_1\ a_1{+}1) \cdots (a_q\ a_q{+}1)$. Induction on q is performed to prove the property of Lemma 4.1.4.

If $q = 0$, then $\pi = \mathrm{id}$ and hence trivially $[\![\vec{i}, o^p, \pi]\!] = [\![\vec{i}, o^p, \mathrm{id}]\!]$.

Otherwise, let $\pi' = (a_1\ a_1{+}1) \cdots (a_{q-1}\ a_{q-1}{+}1)$. Then for $\vec{i'} = \vec{i}[\pi(r)^p := \rho_{\pi(r)}(\vec{i}) \mid 1 \le r < a_q]$, $o = [\![\vec{i}, o^p, \pi(1) : \ldots : \pi(a_q{-}1)]\!]$, and $\ell = \pi(a_q{+}2) : \ldots : \pi(n)$ the following holds due to Lemmas 4.1.6 and 4.1.7, since $\pi'(a_q) \diamond \pi'(a_q{+}1)$ by assumption:

$$\begin{aligned} [\![\vec{i}, o^p, \pi]\!] &= [\![\vec{i}, o^p, \pi'\,(a_q\ a_q{+}1)]\!] \\ &= [\![\vec{i'}, o, \pi'(a_q{+}1) : \pi'(a_q) : \ell]\!] \\ &= [\![\vec{i'}, o, \pi'(a_q) : \pi'(a_q{+}1) : \ell]\!] \end{aligned}$$

Furthermore, for all $1 \le m \le n$ with $m \notin \{a_q, a_q + 1\}$, $\pi(m) = \pi'(m)$. Therefore, by Lemma 4.1.6, $[\![\vec{i'}, o, \pi'(a_q) : \pi'(a_q{+}1) : \ell]\!] = [\![\vec{i}, o^p, \pi']\!]$. By the induction hypothesis $[\![\vec{i}, o^p, \pi']\!] = [\![\vec{i}, o^p, \mathrm{id}]\!]$ holds, which proves the theorem. $\square$

Coming back to the problem stated at the beginning of this section, this theorem presents a method to check order-independence of UDPs in just $O(n^2)$ function comparisons. To this end, for every pair $1 \le j < k \le n$ of inputs two BDDs representing the functions $[\![\vec{i}, o^p, j : k]\!]$ and $[\![\vec{i}, o^p, k : j]\!]$ are constructed, which are then compared for equality. If every such pair of functions is equal, then order-independence of the UDP can be concluded, due to the above theorem. If however two functions are found that compute different outputs, then their XOR describes the counterexample states and hence the UDP is order-dependent. Furthermore, the construction in the proof of Lemma 4.1.7 allows to conclude that there is a previous output value and an input vector in which only the currently considered inputs are changed that lead to two different outputs depending on the order of the two considered inputs.

When applying this method to the UDP `ff_en` of Figure 4.1, then one finds, among others, also the previously described example for the input pair `d` and `ck` where both inputs change from 0 to 1.

For the pair `d` and `en` however, no order-dependence exists, as mentioned above. This is intuitively true because both changes in `d` and `en` will simply keep the current output value, since the output of a flip-flop is only changed on a positive edge of the clock. Due to the symbolic approach, this is checked within fractions of a second.

Verifying Counterexamples

Above, it was presented how to check order-independence of a UDP efficiently. However, a found counterexample, i.e., a situation in which two different orders of evaluating changed inputs lead to different output values of a UDP, might be spurious. On one hand, this is due to the fact that Verilog has a predetermined initial state, in which all signals have the value X, as was described in Chapter 3. From this initial state not all counterexample states have to be reachable. On the other hand, the analysis of UDPs assumes that all inputs to a UDP can change independently of each other. However, the logic driving these inputs in a module might prevent certain combinations. Therefore, the idea is to do a reachability analysis, to determine whether a found counterexample is spurious or not in a concrete cell instantiating the analyzed UDP.

Required Permutations for Reachability Analysis

Whether a counterexample is spurious depends on whether its starting state is reachable from the initial state of the whole cell. In contrast to the approach presented in Chapter 3, here all possible execution traces have to be considered, instead of just those that correspond to the order chosen by a simulator. As in that chapter, every evaluation of a UDP is treated as being independent of the others, i.e., for every evaluation of a UDP the order might be a different one from the order used in another evaluation. This models the behavior of uncontrollable external influences that determine the order.

Since non-determinism creates the possibility of a huge state space to explore, the amount of non-determinism in the generated model should be kept as small as possible. Therefore, not all orders are generated, but only as many orders as needed for the UDP to exhibit all different behaviors. A UDP always computes the same output value whenever two swapped input positions in the order have the commuting diamond property. Therefore, a set of equivalence classes is created with respect to the transitive closure of swapping neighboring inputs that have the commuting diamond property. For example, if $2 \diamond 3$ holds, then the permutations $2 : 3 : 1$ and $3 : 2 : 1$ are in the same equivalence class and only one of them has to be considered.

Definition 4.1.9. For a UDP with n inputs, the relation $\leftrightarrow$ on Π_n is defined as $\pi \leftrightarrow \pi'$, iff a $1 \leq k < n$ exists such that $\pi = \pi' \, (k \; k{+}1)$ and $\pi'(k) \diamond \pi'(k+1)$.

The equivalence relation $\equiv$ on Π_n is then defined as the reflexive transitive closure of $\leftrightarrow$.

This equivalence relation can then be used to partition the set of all permutations into equivalence classes. These equivalence classes still capture all required permutations.

Lemma 4.1.10. *Consider a UDP with n inputs. For all $\vec{i} \in I$, $o^p \in \mathbb{T}$, and all permutations $\pi \equiv \pi' \in \Pi_n$ it holds that $[\![\vec{i}, o^p, \pi]\!] = [\![\vec{i}, o^p, \pi']\!]$.*

Proof. Let $\pi \equiv \pi'$. Then $\pi = \pi' \, (a_1 \; a_1{+}1) \cdots (a_q \; a_q{+}1)$ for some $a_1, \ldots, a_q \in \{1, \ldots, n-1\}$ with $\pi_{r-1}(a_r) \diamond \pi_{r-1}(a_r + 1)$ for all $1 \leq r \leq q$, where for every $0 \leq r < q$, $\pi_r = \pi' \, (a_1 \; a_1{+}1) \cdots (a_r \; a_r{+}1)$. Induction on q is performed.

If $q = 0$, then $\pi = \pi'$, which directly shows the desired property.

Otherwise, $\pi = \pi_{q-1} \, (a_q \; a_q{+}1)$. Because of $\pi_{q-1}(a_q) \diamond \pi_{q-1}(a_q{+}1)$, Lemmas 4.1.6 and 4.1.7, can be applied, yielding for $\vec{i'} = \vec{i}[\pi(r)^p := \rho_{\pi(r)}(\vec{i}) \mid 1 \leq r < a_q]$, $o = [\![\vec{i}, o^p, \pi(1) : \ldots : \pi(a_q{-}1)]\!]$, and $\ell = \pi_{q-1}(a_q{+}2) : \ldots : \pi_{q-1}(n)$,

$$\begin{aligned} [\![\vec{i}, o^p, \pi]\!] &= [\![\vec{i}, o^p, \pi_{q-1} \, (a_q \; a_q{+}1)]\!] \\ &= [\![\vec{i'}, o, \pi_{q-1}(a_q{+}1) : \pi_{q-1}(a_q) : \ell]\!] \\ &= [\![\vec{i'}, o, \pi_{q-1}(a_q) : \pi_{q-1}(a_q{+}1) : \ell]\!]. \end{aligned}$$

Furthermore, since $\pi(r) = \pi_{q-1}(r)$ for all $1 \leq r < a_q$, $[\![\vec{i'}, o, \pi_{q-1}(a_q) : \pi_{q-1}(a_q{+}1) : \ell]\!] = [\![\vec{i}, o^p, \pi_{q-1}]\!]$ because of Lemma 4.1.6. Hence, the induction hypothesis can be applied to show that $[\![\vec{i}, o^p, \pi]\!] = [\![\vec{i}, o^p, \pi_{q-1}]\!] = [\![\vec{i}, o^p, \pi']\!]$. $\square$

These equivalence classes are used next to implement the non-deterministic reachability check. It uses only one permutation from each equivalence class, since the above lemma states that all possible behaviors of the UDP are thereby considered.

Non-Deterministic Reachability Analysis

In order to check reachability, the approach of Section 3.2 is followed in encoding the problem as a Boolean Transition System (BTS), which is a transition system with vectors of Booleans as states. However, in contrast to Section 3.2, here all possible behaviors of the UDPs are allowed. For this purpose, the required permutations of the previous section are used and only these are encoded in a (non-deterministic) transition relation. This transition relation is defined as the conjunction of the following formulas for each UDP `udp` with output o (and next output value o') in the cell:

$$\bigvee_{\pi \in \Pi_n / \equiv_{\texttt{udp}}} o' \leftrightarrow [\![\texttt{udp}, \vec{i}, o, \pi]\!]_{\mathbb{B}\times\mathbb{B}}$$

To make these formulas work on Booleans, the dual-rail encoding of the ternary values is used again, where $0 = (\mathsf{true}, \mathsf{false})$, $1 = (\mathsf{false}, \mathsf{true})$, and $\mathsf{X} = (\mathsf{true}, \mathsf{true})$. Furthermore, $(v_L, v_H) \leftrightarrow (w_L, w_H) = (v_L \leftrightarrow w_L) \wedge (v_H \leftrightarrow w_H)$. The dual-rail encoding $[\![\cdot]\!]_{\mathbb{B}\times\mathbb{B}}$ of UDPs is a straight-forward modification of the dual-rail encoding given in Section 3.2, where instead of modeling the simulator order, the order given as extra argument is encoded.

Using such a non-deterministic BTS, the reachability problem can be expressed in the input language of the NuSMV model checker [CCG$^+$02]. The property to be verified is the negation of the counterexample states. This way, a trace is obtained, leading to a counterexample state, in case an order-dependent UDP can exhibit this behavior in an execution.

Therefore, the LTL formula to be checked for a pair j and k of order-dependent inputs is the following, where, by slight abuse of notation, $j \not\simeq_{\texttt{udp}} k$ shall denote the set of all counterexample states for this pair:

$$\mathsf{G} \neg \left(\bigvee_{s \in j \not\simeq_{\texttt{udp}} k} s \right)$$

As an example, the UDP `ff_en` given in Figure 4.1 is extended with an asynchronous reset signal, as shown in Figure 4.3. This UDP `ff_en_rst` behaves like the UDP `ff_en` when `rst` is 0, but forces the output to be 0 when `rst` is 1.

For this UDP, a counterexample is found for the inputs `d` and `rst`. However, this counterexample depends on the previous output value being 1 or X and the input `rst` having the previous value 1. Such a configuration is not reachable, since setting the input `rst` to 1 in some previous state always results in the value 0 for the output. This is verified by the NuSMV model checker, reporting that none of the reachable states is a counterexample state.

Considering inputs `ck` and `en`, a set of counterexample states is found. When applying the encoding and checking reachability, a trace to a counterexample state is

```
primitive ff_en_rst(q, d, ck, en, rst);
  output q; reg q;
  input d, ck, en, rst;

  table
    //  d  ck   en   rst : q : q+
        0 (01)  1     ?  : ? : 0;
        1 (01)  1     0  : ? : 1;
        ? (10)  ?     0  : ? : -;
        *  ?    ?     0  : ? : -;
        ?  ?    0     0  : ? : -;
        ?  ?    *     0  : ? : -;
        ?  ?    ?     1  : ? : 0;
        ?  ?    ?     *  : 0 : 0;
  endtable
endprimitive
```

Figure 4.3: UDP implementing a D Flip-Flop with Enable and Reset

produced, where the previous output is 1, inputs `d` and `rst` are 0, and both inputs `ck` and `en` change from 0 to 1. This indeed may lead to two different outputs of the UDP, since either the output remains unchanged if the enable signal `en` is still 0 while the change in the clock `ck` is processed, or the output takes on the value 0 from the input `d` if the enable signal is first set to 1 and then the rising edge of `ck` is considered.

Experimental Results

Practical applicability of the method was tested on the Nangate Open Cell Library [Nan08]. It contains 12 different cells that instantiate a sequential UDP and that are in the VeriCell subset of Verilog. All experiments were conducted on a Linux PC containing an Intel Pentium 4 3.0 GHz processor and 1 GB RAM.

The results are shown in Table 4.1, where the first column gives the name of the (sequential) cell, the second column gives the number of UDP input pairs, and the third column shows the number of pairs that were found to be order-dependent. For such input pairs, the BTS encoding presented above was used to create input files for the NuSMV model checker [CCG$^+$02], to check whether the found order-dependencies of the UDP are reachable from the initial state of the cell. When allowing the external inputs of the cell to be all possible values from $\mathbb{T}$, then a counterexample state, which exhibits order-dependent behavior, can be reached for all found order-dependencies, as can be seen in the 4th column of Table 4.1. However, quite a few of these counterexamples are due to the value X being allowed as external input, something that cannot happen in a hardware implementation. When restricting the external inputs of the cell to be binary, i.e., either 0 or 1, but still allowing the internal signals to have any value from $\mathbb{T}$, then a number of counterexamples are not reachable anymore, as can be seen in the 5th column of Table 4.1. Thus, when the environment of a cell is restricted to provide binary values, then these order-dependencies will never occur.

Table 4.1: Order-Independence of VeriCell descriptions in the Nangate Open Cell Library

Cell	# Inp. Pairs	# Ord-Dep Prs	# Reach $\mathbb{T}$	# Reach $\mathbb{B}$	Time $\mathbb{T}$ [s]	Time $\mathbb{B}$ [s]
CLKGATE	1	1	1	1	0.10	0.10
CLKGATETST	2	1	1	1	0.12	0.12
DFF	1	1	1	1	0.12	0.05
DFFR	3	3	3	2	0.61	0.47
DFFS	3	3	3	2	0.97	0.28
DFFRS	6	6	6	4	4.21	8.03
DLH	1	1	1	1	0.10	0.11
DLL	1	1	1	1	0.10	0.11
SDFF	1	1	1	1	1.43	0.58
SDFFR	3	3	3	2	4.11	3.00
SDFFS	3	3	3	2	2.63	3.42
SDFFRS	6	6	6	4	15.01	23.77

Table 4.2: Comparing Approaches for Checking Order-Independence of VeriCell descriptions in the Nangate Open Cell Library

Cell	# Inp. Pairs	# Orders	Times [s] Thm. 4.1.8	Times [s] Lem. 4.1.4
CLKGATE	1	1	0.01 / 0.10	0.01 / 0.12
CLKGATETST	2	5	0.01 / 0.12	0.02 / 0.16
DFF	1	1	0.01 / 0.05	0.01 / 0.14
DFFR	3	5	0.01 / 0.47	0.02 / 0.61
DFFS	3	5	0.01 / 0.28	0.02 / 0.59
DFFRS	6	23	0.01 / 8.03	0.34 / 27.06
DLH	1	1	0.01 / 0.11	0.01 / 0.12
DLL	1	1	0.01 / 0.11	0.02 / 0.12
SDFF	1	1	0.01 / 0.58	0.01 / 1.06
SDFFR	3	5	0.01 / 3.00	0.07 / 11.07
SDFFS	3	5	0.01 / 3.42	0.10 / 11.13
SDFFRS	6	23	0.02 / 23.77	0.44 / > 3600

Finally, the 6th and 7th columns of Table 4.1 show the time it took NuSMV to check reachability when the inputs were allowed to take values from $\mathbb{T}$ and when they were allowed to take values from $\mathbb{B} = \{0, 1\}$, respectively. It can be observed that the time taken for the verification is reasonably small, only for the largest cell `SDFFRS` the verification took more than 10 seconds.

To assess the efficiency of the presented method based on Theorem 4.1.8, it is compared to a naive approach based on Lemma 4.1.4, and the time it takes to compute order-dependencies and to model check the reachability of possible counterexample states is measured. For these experiments, the external inputs are restricted to be binary. The results are shown in Table 4.2, where the first column gives the name of the cell, the second the number of input pairs that had to be checked, and the third column shows the number of permutations that had to be compared with the identity permutation. In the fourth column, the first number shows the time it took to compute the order-dependent counterexample states using the commuting diamond property; the second number shows the time it took NuSMV to check reachability of such order-dependent states, which is the same as in Table 4.1. The last column shows first the time it took to compute order-dependent states by comparing all orders with the identity permutation. Second, it displays the time taken by NuSMV to check reachability of such order-dependent states.

A first observation is that checking order-independence of UDPs is extremely fast, regardless of the approach. However, when it comes to model-checking reachability of the candidate counterexample states, the approach based on the commuting diamond property significantly outperforms the naive approach. Particularly for the largest cell `SDFFRS` the model checking timed out after one hour when investigating all possible orders, whereas it was finished within 24 seconds using the commuting diamond property.

These results can be explained by two factors. First, the naive exploration of the state space has to investigate all possible orders that are not behaving in the same way as the identity permutation for the method based on Lemma 4.1.4. Thus, the transition relation is far more complex. Second, for orders that are equivalent to each other but not to the identity permutations, counterexamples are created for every order that all have to be checked. Such equivalences are factored out in the approach based on Theorem 4.1.8, both for the transition relation and for the counterexamples, due to the much more fine-grained commuting diamond property.

4.2 Order-Independence of Transistor Netlists

In the previous section, it was presented how order-dependence, leading to non-determinism of VeriCell descriptions, can be checked for UDPs. The same problem, i.e., different orders of applying input changes leading to different computation results, also exists for transistor netlists. Hence, this section presents a technique to analyze order-independence of transistor netlists.

In transistor netlists there is no initial state in which evaluations start. Instead, any state might be initial. Therefore, the reachability check presented in the previous section, which checks that a found order-dependence can actually occur in the cell, is not applicable for transistor netlist descriptions. But still, also in transistor netlists order-dependencies exist that are not relevant. These order-dependencies only occur in transient initial states, i.e., they can only occur once at the beginning and never thereafter. Thus, the long-run behavior of the transistor netlist should be analyzed. To do so, this section first presents an approximation of the long-run behavior, by requiring a pre-determined number of predecessors for a state from which an order-dependence is observed. This approximates the states reachable from a strongly connected component (SCC) of the transition system, in which all states are reachable from each other and therefore describe the long-run behavior. Analyzing the SCCs further, it is then shown that requiring one predecessor state is exactly the restriction to the long-run behavior of transistor netlists.

Preliminaries

Transistor netlists are abstracted to *vector-based transition systems*, whose main feature is the representation of inputs, considered to be labels of the transitions, as *vectors* containing a fixed number of elements.

Definition 4.2.1. Let S be a finite set of *states* and U be a finite set of *input values*.

A *vector-based transition system* over states S and input domain U with n inputs is defined as a 3-tuple $T = (S, I, \delta)$, where $I = U^n$ and $\delta : S \times I \to \mathcal{P}(S)$.

In the above definition, $\mathcal{P}(S)$ denotes the power-set of S. Note that no initial state is defined, instead an evaluation is allowed to start in any arbitrary state from S. In the rest of this section, only vector-based transition systems will be considered. Hence, they will also be referred to simply as *transition systems*.

Next, a few properties of vector-based transition systems and a graph representation as a relation on the states, labeled with the current input vector, are defined.

Definition 4.2.2. Let $T = (S, I, \delta)$ be a vector-based transition system.

Transition system T is called *deterministic*, iff for all $\vec{i} \in I$ and all $s \in S$, $|\delta(s, \vec{i})| = 1$. Otherwise, it is called *non-deterministic*. The transition system T is called *deadlock-free*, iff for all $\vec{i} \in I$ and all $s \in S$, $|\delta(s, \vec{i})| \geq 1$.

The transition relation $\to \subseteq S \times I \times S$ is defined as $s \xrightarrow{\vec{i}}_T s'$ iff $s' \in \delta(s, \vec{i})$. It is allowed to leave out the subscript T if the transition system is clear from the context.

Composition of transitions is denoted $s_0 \xrightarrow{\vec{i}_1} \circ \xrightarrow{\vec{i}_2} s_2$ for states $s_0, s_2 \in S$ and is defined iff a state $s_1 \in S$ exists such that $s_0 \xrightarrow{\vec{i}_1} s_1 \xrightarrow{\vec{i}_2} s_2$.

Substitutions are defined similar to the substitutions used in Section 4.1, however here no previous values are used. Therefore, a substitution is denoted by $\sigma = [a_1 := w_1, \ldots, a_r := w_r]$ for pairwise disjoint $a_1, \ldots, a_r \in \mathbb{N}$ and $w_1, \ldots, w_r \in U$, and is defined for a vector $\vec{v} = (v_1, \ldots, v_z) \in U^z$ as the mapping $\sigma(\vec{v}) = (v'_1, \ldots, v'_z)$ where $v'_j = w$ if $j := w \in \sigma$ and $v'_j = v_j$ otherwise. Again, application of a substitution is also written as $\vec{v}\sigma = \sigma(\vec{v})$. Projections are defined accordingly as $\rho_j(\vec{v}) = v_j$ for $1 \leq j \leq z$.

The sets $\mathcal{L}_n$ and Π_n are also as defined in Section 4.1, i.e., all lists without duplicates and all permutations (lists π of length $|\pi| = n$) over the numbers $1, \ldots, n$, respectively. Given a list, the function $\mathrm{sort} : \mathcal{L}_n \to \mathcal{L}_n$ returns this list sorted.

Order-Independence Analysis of Vector-Based Transition Systems

As in Section 4.1, the inputs of a transistor netlist are assumed to change one at a time. Thus, not all possible evaluations of a vector-based transition system have to be considered, but only those where this restriction holds.

Definition 4.2.3. A (possibly infinite) trace $s_1 \xrightarrow{\vec{i}_1} s_2 \xrightarrow{\vec{i}_2} s_3 \xrightarrow{\vec{i}_3} \cdots$ of a vector-based transition system $T = (S, I, \delta)$ with $I = U^n$ is called *one-input restricted*, iff for all $j \in \mathbb{N} \setminus \{0\}$ it holds that $\mathrm{d_H}(\vec{i}_j, \vec{i}_{j+1}) \leq 1$, where $\mathrm{d_H} = |\{1 \leq j \leq n \mid \rho_j(\vec{i}_1) \neq \rho_j(\vec{i}_2)\}|$ denotes the *Hamming distance* [Ham50].

A one-input restricted trace therefore comprises successive steps $s_j \xrightarrow{\vec{i}_j} s_{j+1} \xrightarrow{\vec{i}_{j+1}} s_{j+2}$ such that $\vec{i}_{j+1} = \vec{i}_j[a_j := v_j]$ for some $1 \leq a_j \leq n$ and some value $v_j \in U$.

The following example shows that, in general, all possible interleavings of input changes can differ from changing all inputs at the same time. This is because a one-input restricted trace still sees some old input values and gradually updates the state values.

Example 4.2.4. Let $T = (S, I, \delta)$ be the following deterministic transition system, where $S = \mathbb{B} = \{0, 1\}$, $I = \mathbb{B}^2$, and the transition function δ is illustrated below (the transition labeled $\mathbb{B}^2$ is a shorthand for the 4 transitions each with a label in $\mathbb{B}^2$):

$$\begin{tikzcd}
0 \arrow[loop left, "{(0,0)\,,\,(1,1)}"] \arrow[rr, "{(0,1)\,,\,(1,0)}"] & & 1 \arrow[loop right, "\mathbb{B}^2"]
\end{tikzcd}$$

Consider the different traces when transitioning from input vector $(0, 0)$ to $(1, 1)$:

$$\begin{array}{lllllll}
(0) & \xrightarrow{(0,0)} & (0) & \xrightarrow{(1,0)} & (1) & \xrightarrow{(1,1)} & (1) \\
(0) & \xrightarrow{(0,0)} & (0) & \xrightarrow{(0,1)} & (1) & \xrightarrow{(1,1)} & (1) \\
(0) & \xrightarrow{(0,0)} & (0) & \xrightarrow{(1,1)} & (0) & &
\end{array}$$

Here, it can be observed that both of the possible one-input restricted traces lead to the same final state (1), whereas the trace directly applying both of the new values results in a different final state (0).

In the remainder of this section, only the interleaving semantics, represented by one-input restricted traces, will be considered. Hence, for each vector-based transition system T, its interleaving interpretation T^I is defined in such a way that, by construction, at most one input is triggered in each step.

Definition 4.2.5. Let $T = (S, I, \delta)$ be a vector-based transition system with $I = U^n$. The corresponding *one-input restricted vector-based transition system* is defined as $T^I = (S \times I, \{1, \ldots, n\} \times U, \delta^I)$, where $(s, \vec{i}) \in S \times I$ is also denoted by $s;\vec{i}$ and where the transition function $\delta^I : (S \times I) \times \{1, \ldots, n\} \times U \to \mathcal{P}(S \times I)$ is defined as $\delta^I(s;\vec{i}, j, v) = \{s';\vec{i}[j := v] \mid s' \in \delta(s, \vec{i}[j := v])\}$.

Also for T^I a transition relation is defined, where $s;\vec{i} \xrightarrow{j} s';\vec{i'}$ iff $s';\vec{i'} \in \delta^I(s;\vec{i}, j, v)$ with $v = i'_j$ for $\vec{i'} = (i'_1, \ldots, i'_n)$. Note that by looking at the position j in $\vec{i}$ and $\vec{i'}$ the change in the input (second component v of the label) can be recovered. Hence, only the position is shown on the label and the actual value of the changing input is left out.

This definition allows to represent a one-input restricted trace in the form $s_1;\vec{i}_1 \xrightarrow{j_1} s_2;\vec{i}_2 \xrightarrow{j_2} \ldots$, where the numbers $j_1, j_2, \ldots$ indicate the triggered inputs in the corresponding steps. Using the construction given above, any one-input restricted trace of the transition system T can be translated into a trace of T^I and vice versa. Therefore, a transition system T and its corresponding one-input restricted transition system T^I will be used interchangeably. In the following example, the transition system T^I for the transition system T of Example 4.2.4 is presented.

Example 4.2.6. For the transition system $T = (S, I, \delta)$ of Example 4.2.4, $T^I = (S \times I, I, \delta^I)$, where $S \times I = \mathbb{B} \times \mathbb{B}^2$ and δ^I is defined as shown in Figure 4.4 (dashed and bold lines are to be considered just like normal lines; they will be used to illustrate other concepts in subsequent examples). For the sake of brevity, the input vector components are concatenated. Furthermore, it should be mentioned

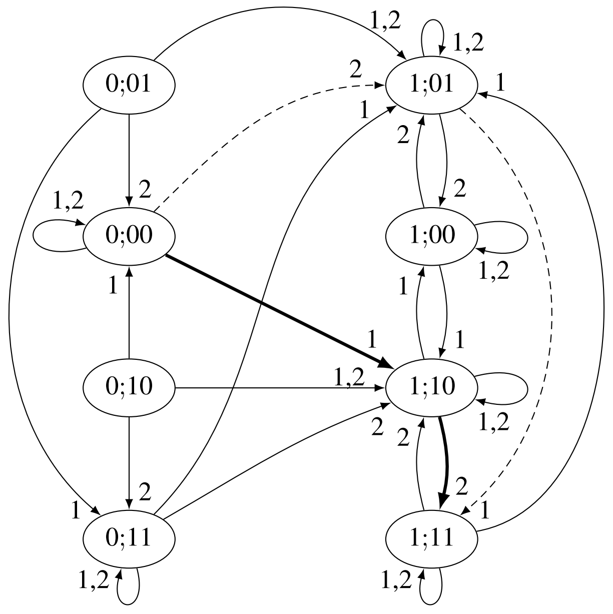


Figure 4.4: Transition System T^I for Example 4.2.4

that a label such as $1, 2$ does not denote a vector, but is a representation of the two separate transitions labeled by 1 and 2, respectively.

To illustrate the construction that leads to the above graph, consider the state 0;00, which represents the state 0 in T that was reached with the input vector 00. When considering the successor states of this state by changing the first input, two possibilities exist, either setting the first input to 0 or to 1. When setting it to 0, one obtains the same input vector 00, and since $0 \xrightarrow{00}_T 0$ one gets the transition $0;00 \xrightarrow{1} 0;00$ in T^I. When setting the first input to 1, input vector 10 must be considered, which implies in T^I the transition $0;00 \xrightarrow{1} 1;10$, due to the transition $0 \xrightarrow{10}_T 1$ in T.

It was already mentioned previously that in the target application of vector-based transition systems created from transistor netlists, transient initial states are common. This is the case since after “boot-up” a transistor netlist may be in any arbitrary state. Such a transient state is one that can only occur once and never be reached again. Since these states are of little practical relevance, the property $k\mathrm{R}$ is defined that determines whether a state has at least k predecessors. Furthermore, this property ensures that input vectors which could not have been used to arrive in a certain state are ruled out.

Definition 4.2.7. Let $T = (S, I, \delta)$ be a vector-based transition system with n inputs. For $k \in \mathbb{N}$, $k\mathrm{R}_T \subseteq S \times I$ is the smallest relation containing all combinations $s;\vec{i} \in S \times I$ of a state and input such that there exist $s_1;\vec{i}_1, \ldots, s_k;\vec{i}_k \in S \times I$ and $j_1, \ldots, j_k \in \{1, \ldots, n\}$ satisfying $s_1;\vec{i}_1 \xrightarrow{j_1} \cdots \xrightarrow{j_{k-1}} s_k;\vec{i}_k \xrightarrow{j_k} s;\vec{i}$.

Instead of $s;\vec{i} \in k\mathrm{R}_T$, also the notation $k\mathrm{R}_T(s;\vec{i})$ will be used in the following, where the subscript T is left out if the transition system is clear from the context. In case $k\mathrm{R}(s;\vec{i})$ holds, then the state $s;\vec{i}$ is said to be *k-step reachable*.

Next, the single-step transitions of the interleaving semantics are combined to obtain traces in which different permutations of applying input are used. For this purpose, the transition relation introduced below is labeled with a permutation of positions to indicate the order in which the inputs are triggered. Furthermore, the relation is restricted to those states that are k-step reachable.

Definition 4.2.8. Let $T = (S, I, \delta)$ be a transition system, $s, s' \in S$, $\vec{i}, \vec{i'} \in I = U^n$, and $\ell \in \Pi_n$ with $\vec{i} = (i_1, \ldots, i_n)$, $\vec{i'} = (i'_1, \ldots, i'_n)$, and $\ell = j_1 : \ldots : j_n$. For the transition system T^I, the transition relation $\twoheadrightarrow_k$ labeled by the permutation ℓ is defined as $s;\vec{i} \xrightarrow{\ell}\!\!\!\twoheadrightarrow_k s';\vec{i'}$ iff $k\mathrm{R}(s;\vec{i})$ and there exist states $s_1;\vec{i}_1, \ldots, s_{n+1};\vec{i}_{n+1} \in S \times I$ such that $s;\vec{i} = s_1;\vec{i}_1 \xrightarrow{j_1} s_2;\vec{i}_2 \xrightarrow{j_2} \ldots \xrightarrow{j_{n-1}} s_n;\vec{i}_n \xrightarrow{j_n} s_{n+1};\vec{i}_{n+1} = s';\vec{i'}$.

This is not the only way to define the combination of considering input events; one could also allow for applying a certain event zero or more times. This will be treated later in this section, where it will be shown that all different variants of the above definition lead to the same conclusion as far as order-independence is concerned.

To illustrate the relation $\twoheadrightarrow_k$, first consider the state 0;01 in the transition system T^I of Example 4.2.6, which is also depicted in Figure 4.4. As can be seen in the figure, there are no incoming arcs for this state. Hence, for any $k > 0$, it holds that $0;01 \not\xrightarrow{\ell}\!\!\!\twoheadrightarrow_k s;\vec{i}$ for any $s;\vec{i} \in S \times I$ and $\ell \in \Pi_2$, since $k\mathrm{R}(0;01)$ does not hold. Next, consider the state 0;00. For any $k \in \mathbb{N}$, the state 0;00 is k-step reachable, for example by repeating $0;00 \xrightarrow{1} 0;00$ k times. Thus, $0;00 \xrightarrow{1:2}\!\!\!\twoheadrightarrow_k 1;11$ holds due to the bold path in Figure 4.4. Furthermore, also $0;00 \xrightarrow{2:1}\!\!\!\twoheadrightarrow_k 1;11$ holds because of the dashed path in Figure 4.4. Hence, in this case it does not matter in which order these two changing input values are evaluated.

In general, this shall be determined for all states and all possible orders of input changes, i.e., the problem is whether the state reached after applying a number of input changes is the same regardless of the chosen order of their application. If this is the case, the transition system is called *order-independent*.

Definition 4.2.9. Given a transition system $T = (S, I, \delta)$ with n inputs, relation $\twoheadrightarrow_k$ is called *order-independent*, iff $\xrightarrow{\ell}\!\!\!\twoheadrightarrow_k = \xrightarrow{\ell'}\!\!\!\twoheadrightarrow_k$ for all $\ell, \ell' \in \Pi_n$.

However, as was the case for UDPs in Lemma 4.1.4, it can easily be seen that not all pairs of permutations have to be considered. One of them can, for example, be fixed to be the identity permutation, which in the considered list representation is equal to the sorted permutation.

Lemma 4.2.10. *For a transition system $T = (S, I, \delta)$ with n inputs, relation $\twoheadrightarrow_k$ is order-independent, iff $\xrightarrow{\ell}\!\!\!\twoheadrightarrow_k = \xrightarrow{\mathrm{sort}(\ell)}\!\!\!\twoheadrightarrow_k$ for all $\ell \in \Pi_n$.*

Proof. Follows from the transitivity of equality. □

Hence, one could check order-independence of $\twoheadrightarrow_k$ by constructing all of these $n!$ relations and comparing them for equality. However, following the basic idea of Section 4.1, this check should be reduced to a quadratic number of comparisons.

The approach relies on the structure of the computation, only treating one input change at a time, and on two properties of the functions computing the next output value when considering a single input as changed: The first (rather modest) property required to check order-independence efficiently is *deadlock freedom*. In the target application area of transistor netlist hardware descriptions, this is very natural since a hardware circuit will always compute some values from its inputs. The second property is that when there is no change in the considered input, then also the output of that evaluation step remains unchanged. In other words, it is required that the twofold application of the same input vector results in the same state as applying the input vector only once. This property, which is similar to Property 4.1.1 required for UDPs, is expressed formally below.

Fixed-Point Property. Let $T = (S, I, \delta)$ be a transition system with n inputs. T has the *fixed-point property* iff for all $s_1, s_2, s_3 \in S$, all $1 \leq j, j' \leq n$, and all $\vec{i}, \vec{i'} \in I$: If $s_1;\vec{i} \xrightarrow{j} s_2;\vec{i'} \xrightarrow{j'} s_3;\vec{i'}$, then $s_3 = s_2$.

For a deadlock-free transition system satisfying the fixed-point property, order-independence is equivalent to checking the equality of two specific relations for all pairs of inputs. This is expressed in the *k-step reachable diamond property*, which is defined next.

Definition 4.2.11. Let $T = (S, I, \delta)$ be a vector-based transition system with n inputs.

Two inputs $1 \leq j, j' \leq n$ with $j \neq j'$ are said to have the *k-step reachable diamond property*, denoted $j \overset{\downarrow k}{\diamond} j'$, iff $s;\vec{i} \xrightarrow{j} \circ \xrightarrow{j'} s';\vec{i'} \iff s;\vec{i} \xrightarrow{j'} \circ \xrightarrow{j} s';\vec{i'}$ for all $s;\vec{i}$, $s';\vec{i'} \in S \times I$ satisfying $k\mathrm{R}(s;\vec{i})$.

The k-step reachable diamond property is similar to the commuting diamond property for UDPs in Definition 4.1.5, except that it also requires k-step reachability of the state that starts the diamond. Thus, this property is also similar to the commuting diamond property known from term rewriting [BN98], but here it is not required to hold globally. Instead, it is only required for a subset of states, namely those having at least k predecessors.

Using the k-step reachable diamond property, order-independence of the relation $\twoheadrightarrow_k$ of a vector-based transition system can be checked. This is proven in the next lemma.

Lemma 4.2.12. *Let $T = (S, I, \delta)$ be a vector-based transition system with n inputs. The transition relation $\twoheadrightarrow_k$ is order-independent, if $j \overset{\downarrow k}{\diamond} j'$ for all $1 \leq j < j' \leq n$.*

Proof. Assume that $j \overset{\downarrow k}{\diamond} j'$ holds for each pair $1 \leq j < j' \leq n$. It will be proven that the property of Lemma 4.2.10 holds, i.e., that $\overset{\ell}{\twoheadrightarrow}_k = \overset{\mathrm{sort}(\ell)}{\twoheadrightarrow}_k$ for every $\ell \in \Pi_n$. To this end, let $\ell = j_1 : \cdots : j_n$. Induction is performed on the number of swaps required in the Bubble-Sort algorithm to sort list ℓ.

If no swaps are required, then $\ell = \mathrm{sort}(\ell)$ and therefore the lemma vacuously holds.

Otherwise, let $\ell' = j_1 : \cdots : j_{r+1} : j_r : \cdots : j_n$ be the list obtained after the first swap performed by the Bubble-Sort algorithm. Using the k-step reachable diamond property, one obtains for all k-step reachable states $s;\vec{i}$ that

$$s;\vec{i} \xrightarrow{j_r} \circ \xrightarrow{j_{r+1}} s';\vec{i'} \iff s;\vec{i} \xrightarrow{j_{r+1}} \circ \xrightarrow{j_r} s';\vec{i'}. \qquad (\diamond)$$

Let $s_1;\vec{i_1} \overset{\ell}{\twoheadrightarrow}_k s_2;\vec{i_2}$ for some $s_1;\vec{i_1}, s_2;\vec{i_2} \in S \times I$. This is by Definition 4.2.8 equivalent to $k\mathrm{R}(s_1;\vec{i_1})$ and $s_1;\vec{i_1} \xrightarrow{j_1} \circ \cdots \circ \xrightarrow{j_n} s_2;\vec{i_2}$. Thus, for every state $s;\vec{i}$ of this trace, $k\mathrm{R}(s;\vec{i})$ holds. This allows to apply ($\diamond$) to this trace, showing the following equivalences:

$$\begin{aligned}
& s_1;\vec{i_1} \overset{\ell}{\twoheadrightarrow}_k s_2;\vec{i_2} \\
\iff\ & s_1;\vec{i_1} \xrightarrow{j_1} \circ \cdots \circ \xrightarrow{j_r} \circ \xrightarrow{j_{r+1}} \circ \cdots \circ \xrightarrow{j_n} s_2;\vec{i_2} \\
\overset{(\diamond)}{\iff}\ & s_1;\vec{i_1} \xrightarrow{j_1} \circ \cdots \circ \xrightarrow{j_{r+1}} \circ \xrightarrow{j_r} \circ \cdots \circ \xrightarrow{j_n} s_2;\vec{i_2} \\
\iff\ & s_1;\vec{i_1} \overset{\ell'}{\twoheadrightarrow}_k s_2;\vec{i_2}
\end{aligned}$$

Therefore, $\overset{\ell}{\twoheadrightarrow}_k = \overset{\ell'}{\twoheadrightarrow}_k$ holds. Applying the induction hypothesis to ℓ' gives $\overset{\ell'}{\twoheadrightarrow}_k = \overset{\mathrm{sort}(\ell')}{\twoheadrightarrow}_k$, and since $\mathrm{sort}(\ell) = \mathrm{sort}(\ell')$ the desired property $\overset{\ell}{\twoheadrightarrow}_k = \overset{\mathrm{sort}(\ell)}{\twoheadrightarrow}_k$ has been proven. □

In the next lemma, it is proven that also the other direction of Lemma 4.2.12 holds. For this, however, the restriction to deadlock-free transition systems having the fixed-point property is required.

Lemma 4.2.13. *Let $T = (S, I, \delta)$ be a deadlock-free vector-based transition system with n inputs having the fixed-point property. Then $j \overset{k}{\diamond} j'$ holds for all $1 \leq j < j' \leq n$ if the transition relation $\twoheadrightarrow_k$ is order-independent.*

Proof. To prove the lemma, assume towards a contradiction that for some $1 \leq j < j' \leq n$, $j \overset{k}{\diamond} j'$ does not hold, i.e., there exist $s;\vec{i}$, $s_1;\vec{i'} \in S \times I$ such that $k\mathrm{R}(s;\vec{i})$, $s;\vec{i} \xrightarrow{j} \circ \xrightarrow{j'} s_1;\vec{i'}$, and not $s;\vec{i} \xrightarrow{j'} \circ \xrightarrow{j} s_1;\vec{i'}$ (or vice versa, but that case is symmetric to the considered one by exchanging the indices j and j').

Define lists $\ell = j : j' : \ell_{\mathrm{tl}}$ and $\ell' = j' : j : \ell_{\mathrm{tl}}$, where $\ell_{\mathrm{tl}} = 1 : \cdots : j{-}1 : j{+}1 : \cdots : j'{-}1 : j'{+}1 : \cdots : n$. Then, both ℓ and ℓ' are permutations by construction. Because the transition system is deadlock-free, there exists a state $s_1';\vec{i'} \in S \times I$ such that $s;\vec{i} \overset{\ell}{\twoheadrightarrow}_k s_1';\vec{i'}$, i.e., $s;\vec{i} \xrightarrow{j} \circ \xrightarrow{j'} s_1;\vec{i'} \xrightarrow{\ell_{\mathrm{tl}}} s_1';\vec{i'}$, where the relation $\xrightarrow{1} \circ \cdots \circ \xrightarrow{j-1} \circ \xrightarrow{j+1} \circ \cdots \circ \xrightarrow{j'-1} \circ \xrightarrow{j'+1} \circ \cdots \circ \xrightarrow{n}$ is abbreviated with $\xrightarrow{\ell_{\mathrm{tl}}}$. The fixed-point property can be applied repeatedly to the steps of $\xrightarrow{\ell_{\mathrm{tl}}}$, which shows that $s_1 = s_1'$. Assume $s;\vec{i} \overset{\ell'}{\twoheadrightarrow}_k s_1;\vec{i'}$. Then there exists a state $s_2;\vec{i'} \in S \times I$ such that $s;\vec{i} \xrightarrow{j'} \circ \xrightarrow{j} s_2;\vec{i'} \xrightarrow{\ell_{\mathrm{tl}}} s_1;\vec{i'}$. Applying the fixed-point property repeatedly to this trace yields $s_2 = s_1$. This however contradicts the assumption that $s;\vec{i} \xrightarrow{j'} \circ \xrightarrow{j} s_1;\vec{i'}$ does not hold, which was to be proven. □

Lemmas 4.2.12 and 4.2.13 are combined into the main theorem of this section, stating that the k-step reachable diamond property is equivalent to order-independence of the relation $\twoheadrightarrow_k$ for a deadlock-free transition system satisfying the fixed-point property. Hence, the global property of order-independence can be checked by only considering a local property, namely the k-step reachable diamond property. Here, local refers to the fact that for the k-step reachable diamond property, only a constant number of steps (k steps to reach an initial state and two steps on each side) have to be considered, whereas for order-independence the number of inputs n linearly affects the number of steps to be considered.

Theorem 4.2.14. *Let $T = (S, I, \delta)$ be a deadlock-free vector-based transition system having the fixed-point property, and let $I = U^n$. Then the transition relation $\twoheadrightarrow_k$ is order-independent, iff $j \overset{\downarrow k}{\diamond} j'$ for all $1 \leq j < j' \leq n$.*

Proof. Soundness follows from Lemmas 4.2.12 and completeness from Lemma 4.2.13. □

Note that the theorem, unlike the corresponding Theorem 4.1.8 in Section 4.1, requires only deadlock-freedom and the fixed-point property; it does not need the extra requirement of Property 4.1.2 that a new input value, for an input that is not currently considered, does not change the computation. This is the case because evaluations of a vector-based transition systems do not store a copy of the new input values, instead the input vector is changed only at the positions of the currently considered pair of inputs in Definition 4.2.11 of the k-step reachable diamond property.

Both deadlock-freedom and the fixed-point property are only requirements in Lemma 4.2.13, i.e., order-independence of a vector-based transition system that does not satisfy either of the two properties can still be proven by Lemma 4.2.12. However, the implication in the other direction does not hold anymore, thus the diamond property is strictly stronger than order-independence for such transition systems. This is demonstrated below, where counter-examples to order-independence are given, witnessing that deadlock-freedom and the fixed-point property cannot be dropped in the "only-if"-direction of Theorem 4.2.14. The first counter-example shows the effect of dropping deadlock-freedom.

Example 4.2.15. Let $T = (\mathbb{B}^2, \mathbb{B}^3, \delta)$ be the transition system whose transition function δ is depicted below, where the components of the state and input vectors are concatenated.

$$\begin{tikzcd}
00 \arrow[r, bend left, "000"] & 01 \arrow[r, bend left, "001"] & 10 \arrow[r, bend left, "011"] & 11
\end{tikzcd}$$

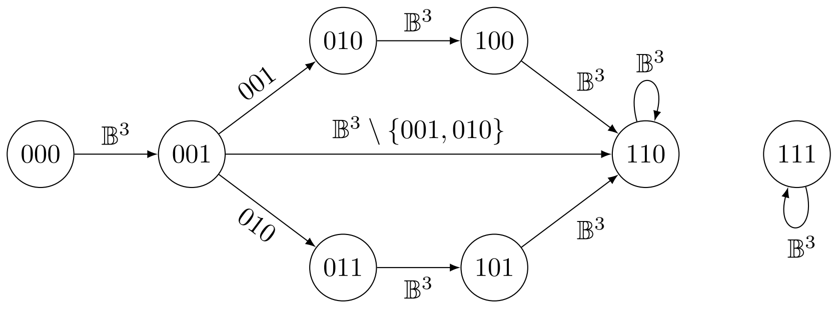
This transition system satisfies the fixed-point property, since in any state an input vector leading to that state cannot be applied again. Furthermore, $\twoheadrightarrow_1$ is order-independent, since there is no path of length 4 and hence any trace starting in a one-step reachable state will deadlock. However, $2 \overset{\downarrow 1}{\diamond} 3$ is not satisfied: For example, in state 01 together with the input vector 000 it holds that $00;001 \xrightarrow{3} 01;000$ (thus, the state is 1-step reachable) and $01;000 \xrightarrow{3} 10;001 \xrightarrow{2} 11;011$. However, no state $s;\vec{i} \in S \times I$ exists such that $01;000 \xrightarrow{2} s;\vec{i}$, hence the requirement of Definition 4.2.11 is not satisfied.

Note that for the above example, it is crucial to have $I = \mathbb{B}^3$ and not $I = \mathbb{B}^2$ by removing the first input component. This is because if $I = \mathbb{B}^2$ was used, Definition 4.2.8 would only concern paths of length 3 (the initial step and two changes of the inputs), hence the above would be a counterexample to order-independence with $k = 1$.

The above example will always eventually deadlock, since every state can be reached at most once. Thus, the behavior after at most 3 steps will be deadlocking, which is always order-independent since there are no successor states which could be different. By increasing k to 2, order-independence of the above example can be proven, as now a counter-example to the commuting diamond property would have to be of length 4 (2-step reachable plus applying the two inputs). It will be shown later that by iteratively increasing k, the approximation of the long-run behavior improves.

The next example shows that also the fixed-point property cannot be removed in the "only-if"-direction of Theorem 4.2.14.

Example 4.2.16. Let $T = (\mathbb{B}^3, \mathbb{B}^3, \delta)$ be the transition system whose transition function δ is defined as illustrated below:



This transition system is deterministic and hence deadlock-free, however it does not satisfy the fixed-point property, since for example $010;001 \xrightarrow{1} 100;001 \xrightarrow{2} 110;001$ and $100 \neq 110$. Furthermore, relation $\twoheadrightarrow_1$ is order-independent:

- For any state $s \in \mathbb{B}^3 \setminus \{000, 111\}$, any input vectors $\vec{\imath}, \vec{\imath}' \in \mathbb{B}^3$, and any permutation $\ell \in \Pi_3$, $s;\vec{\imath} \overset{\ell}{\twoheadrightarrow}_1 110;\vec{\imath}'$.

- For any state $s' \in \mathbb{B}^3$, any input vectors $\vec{\imath}, \vec{\imath}' \in \mathbb{B}^3$, and any permutation $\ell \in \Pi_3$ it holds that $000;\vec{\imath} \overset{\ell}{\not\twoheadrightarrow}_1 s';\vec{\imath}'$, since no $s_0;\vec{\imath}_0 \in \mathbb{B}^3 \times \mathbb{B}^3$ and no $1 \leq j \leq 3$ exist such that $s_0;\vec{\imath}_0 \xrightarrow{j} 000;\vec{\imath}$.

- For any input vectors $\vec{\imath}, \vec{\imath}' \in \mathbb{B}^3$ and any permutation $\ell \in \Pi_3$, $111;\vec{\imath} \overset{\ell}{\twoheadrightarrow}_1 111;\vec{\imath}'$.

However, for the state $001 \in \mathbb{B}^3$ one observes that $000;100 \xrightarrow{1} 001;000$ and the two traces $001;000 \xrightarrow{3} 010;001 \xrightarrow{2} 100;011$ and $001;000 \xrightarrow{2} 011;010 \xrightarrow{3} 101;011$ exist, which shows that $1 \overset{\downarrow 1}{\diamond} 2$ does not hold.

Also for this example, it is sufficient to increase k to 2 to prove order-independence.

Triggering Inputs Multiple Times

Definition 4.2.8 restricts the lists indicating the order of triggering inputs to permutations of the numbers from 1 to n. A natural generalization is to also allow inputs being triggered more than once, or, generalizing even further, to only require those inputs to be triggered at least once, whose values in the initial and the final input vectors are different. Both of these generalizations are formally defined below. To formulate them, the set $\mathcal{L}_n$, containing lists where each of the numbers from 1 to n may occur at most once, is generalized to the set $\overline{\mathcal{L}}_n$, which contains all lists of numbers between 1 and n. Thus, $\mathcal{L}_n \subsetneq \overline{\mathcal{L}}_n$, since for example $1 : 1 : 2 \in \overline{\mathcal{L}}_2$ but $1 : 1 : 2 \notin \mathcal{L}_2$. In the below definitions, to ease presentation, a list is also viewed as the set of elements it contains.

Definition 4.2.17. Let $T = (S, I, \delta)$ be a transition system with $I = U^n$, let $k \in \mathbb{N}$, let $s;\vec{i}$, $s';\vec{i'} \in S \times I$, and let $\ell = j_1 : \cdots : j_z \in \overline{\mathcal{L}}_n$ with $j \in \ell$ for all $1 \leq j \leq n$.

Relation $\twoheadrightarrow_k^{(1)}$ for the transition system T^I is defined as $s;\vec{i} \overset{\ell}{\twoheadrightarrow}_k^{(1)} s';\vec{i'}$ iff $k\mathrm{R}(s;\vec{i})$ and $s;\vec{i} \xrightarrow{j_1} \circ \cdots \circ \xrightarrow{j_z} s';\vec{i'}$.

Relation $\twoheadrightarrow_k^{(1)}$ is called *order-independent*, iff $\overset{\ell}{\twoheadrightarrow}_k^{(1)} = \overset{\ell'}{\twoheadrightarrow}_k^{(1)}$ for all $\ell, \ell' \in \overline{\mathcal{L}}_n$ with $j \in \ell$ for all $1 \leq j \leq n$ and ℓ' being a permutation of ℓ.

Definition 4.2.18. Let $T = (S, I, \delta)$ be a transition system with $I = U^n$, let $k \in \mathbb{N}$, let $s;\vec{i}$, $s';\vec{i'} \in S \times I$ where $\vec{i} = (i_1, \ldots, i_n)$ and $\vec{i'} = (i'_1, \ldots, i'_n)$, and let $\ell = j_1 : \cdots : j_z \in \overline{\mathcal{L}}_n$ with $\{1 \leq j \leq n \mid i_j \neq i'_j\} \subseteq \ell$.

Relation $\twoheadrightarrow_k^{(2)}$ is defined as $s;\vec{i} \overset{\ell}{\twoheadrightarrow}_k^{(2)} s';\vec{i'}$ iff $k\mathrm{R}(s;\vec{i})$ and $s;\vec{i} \xrightarrow{j_1} \circ \cdots \circ \xrightarrow{j_z} s';\vec{i'}$.

The relation $\twoheadrightarrow_k^{(2)}$ is called *order-independent*, iff $\overset{\ell}{\twoheadrightarrow}_k^{(2)} = \overset{\ell'}{\twoheadrightarrow}_k^{(2)}$ for all $\ell, \ell' \in \overline{\mathcal{L}}_n$ with $\{1 \leq j \leq n \mid i_j \neq i'_j\} \subseteq \ell$ and ℓ' being a permutation of ℓ.

It can easily be seen from the above definitions that $\twoheadrightarrow_k^{(0)} \subseteq \twoheadrightarrow_k^{(1)} \subseteq \twoheadrightarrow_k^{(2)}$, where $\twoheadrightarrow_k$ is also denoted by $\twoheadrightarrow_k^{(0)}$. To illustrate the two more general relations, it holds for instance in Example 4.2.6 that 0;00 $\overset{1:2:2}{\twoheadrightarrow}_1^{(1)}$ 1;11 and 0;00 $\overset{1:2:2}{\twoheadrightarrow}_1^{(2)}$ 1;11, whereas 0;00 $\overset{1:2:2}{\not\twoheadrightarrow}_1$ 1;11 since $|1 : 2 : 2| = 3 \neq 2$. Furthermore, 0;00 $\overset{1}{\twoheadrightarrow}_1^{(2)}$ 1;10 but 0;00 $\overset{1:\mathsf{nil}}{\not\twoheadrightarrow}_1^{(1)}$ 1;10, since $|1 : \mathsf{nil}| = 1 < 2$.

Also for $\twoheadrightarrow_k^{(1)}$ and $\twoheadrightarrow_k^{(2)}$, only a single list has to be compared to its corresponding sorted list, as was already observed for the relation $\twoheadrightarrow_k$ in Lemma 4.2.10.

Lemma 4.2.19. *Let T be a transition system with n inputs.*

Relation $\twoheadrightarrow_k^{(1)}$ is order-independent, iff $\overset{\ell}{\twoheadrightarrow}_k^{(1)} = \overset{\mathrm{sort}(\ell)}{\twoheadrightarrow}_k^{(1)}$ for all $\ell \in \overline{\mathcal{L}}_n$ with $j \in \ell$ for all $1 \leq j \leq n$.

Relation $\twoheadrightarrow_k^{(2)}$ is order-independent, iff $\overset{\ell}{\twoheadrightarrow}_k^{(2)} = \overset{\mathrm{sort}(\ell)}{\twoheadrightarrow}_k^{(2)}$ for all $\ell \in \overline{\mathcal{L}}_n$ with $\{1 \leq j \leq n \mid i_j \neq i'_j\} \subseteq \ell$.

Proof. Follows from transitivity of equality. □

Again, the k-step reachable diamond property given in Definition 4.2.11 shall be used to check whether these two generalized relations are order-independent or not. Since $\twoheadrightarrow_k^{(0)} \subseteq \twoheadrightarrow_k^{(1)} \subseteq \twoheadrightarrow_k^{(2)}$, the "only-if" direction in the proof of Theorem 4.2.14 holds directly. Furthermore, the "if" direction of that proof, i.e., the proof of Lemma 4.2.12, does not make use of the restriction to permutations, hence it also holds for the relations $\twoheadrightarrow_k^{(1)}$ and $\twoheadrightarrow_k^{(2)}$. This allows to conclude that if one of the transition relations is order-independent, then all are, provided the transition system is deadlock-free and satisfies the the fixed-point property. This is formally expressed in the lemma below.

Lemma 4.2.20. *For a deadlock-free transition system $T = (S, I, \delta)$ with n inputs that satisfies the fixed-point property, relation $\twoheadrightarrow_k^{(a)}$ with $0 \leq a \leq 2$ is order-independent, iff a $0 \leq b \leq 2$ exists such that $\twoheadrightarrow_k^{(b)}$ is order-independent.*

Proof. The "only-if" direction holds trivially. For the "if" direction, assume $\twoheadrightarrow_k^{(b)}$ is order-independent. Then $j \overset{\downarrow k}{\diamond} k$ holds for all $1 \leq j < j' \leq m$, otherwise such a pair would constitute a counterexample to order-independence of $\twoheadrightarrow_k$ due to the "only-if" direction of Theorem 4.2.14. Such a counterexample would also be a counterexample to order-independence of $\twoheadrightarrow_k^{(1)}$ and $\twoheadrightarrow_k^{(2)}$, since $\overset{\ell}{\twoheadrightarrow}_k = \overset{\ell}{\twoheadrightarrow}_k^{(1)} = \overset{\ell}{\twoheadrightarrow}_k^{(2)}$ for all $\ell \in \Pi_n$. Hence, since the proof of Lemma 4.2.12 does not make use of the requirements imposed onto list ℓ, order-independence of $\twoheadrightarrow_k^{(a)}$ has been proven. $\square$

Example 4.2.16 showed that the fixed-point property cannot be dropped for order-independence of $\twoheadrightarrow_k$. This example still applies to $\twoheadrightarrow_k^{(1)}$. For relation $\twoheadrightarrow_k^{(2)}$ however, this is not a valid counterexample, since it cannot be assumed that a trace has a certain (minimal) length, hence the traces showing that the one-step reachable diamond property is violated in Example 4.2.16 is also a counterexample to order-independence of $\twoheadrightarrow_k^{(2)}$. Indeed, the following lemma shows that order-independence of $\twoheadrightarrow_k^{(2)}$ only requires deadlock-freedom and the k-step reachable diamond property, i.e., the fixed-point property is not required.

Lemma 4.2.21. *For a deadlock-free transition system $T = (S, I, \delta)$ with n inputs the relation $\twoheadrightarrow_k^{(2)}$ is order-independent, iff $j \overset{\downarrow k}{\diamond} j'$ holds for all $1 \le j < j' \le n$.*

Proof. The "if" direction follows from Lemma 4.2.12. To show the "only-if" direction, assume $s;\vec{i} \overset{j:j'}{\twoheadrightarrow}_k^{(2)} s_1;\vec{i'}$ and not $s;\vec{i} \overset{j':j}{\twoheadrightarrow}_k^{(2)} s_1;\vec{i'}$ for some $s, s_1 \in S$, $\vec{i} = (i_1, \ldots, i_n), \vec{i'} = (i'_1, \ldots, i'_n) \in I$. By Definition 4.2.17 and Definition 4.2.5, $\vec{i'} = \vec{i}[j := i'_j, j' := i'_{j'}]$ holds and therefore $\{1 \le j \le n \mid i_j \ne i'_j\} = \{j, j'\}$. Hence, this provides a counterexample to order-independence of $\twoheadrightarrow_k^{(2)}$. $\square$

However, in case the transition system is deadlock-free and satisfies the fixed-point property, then the relations $\twoheadrightarrow_k^{(a)}$ with $0 \le a \le 2$ are all equivalent to the relation $\overset{1:\cdots:n}{\twoheadrightarrow}_k$ as will be shown in the theorem below.

Theorem 4.2.22. *Let $T = (S, I, \delta)$ be a deadlock-free transition system with $I = U^n$ satisfying the fixed-point property.*

If $\twoheadrightarrow_k^{(b)}$ is order-independent for some $0 \le b \le 2$, then $\overset{\ell}{\twoheadrightarrow}_k^{(a)} = \overset{1:\cdots:n}{\twoheadrightarrow}_k$ for all $0 \le a \le 2$ and all lists $\ell \in \overline{\mathcal{L}}_n$ satisfying the requirements of $\overset{\ell}{\twoheadrightarrow}_k^{(a)}$.

Proof. Let $\twoheadrightarrow_k^{(b)}$ be order-independent for some $0 \le b \le 2$. Due to Lemma 4.2.20, all relations $\twoheadrightarrow_k^{(a)}$ are order-independent, i.e., $\overset{\ell}{\twoheadrightarrow}_k^{(a)} = \overset{\mathrm{sort}(\ell)}{\twoheadrightarrow}_k^{(a)}$ for all lists $\ell \in \overline{\mathcal{L}}_n$ that satisfy the requirements of $\twoheadrightarrow_k^{(a)}$. Hence, for $a = 0$ the theorem holds trivially.

For the remaining cases, let $\ell = j_1 : \cdots : j_{|\ell|}$ be an arbitrary list satisfying the requirements of $\twoheadrightarrow_k^{(a)}$. Define $\ell' = j_1 : \cdots : j_{|\ell|} : j_{|\ell|+1} : \cdots : j_{|\ell|+z}$ such that $j \in \ell'$ for all $1 \le j \le n$ and assume that the trace is starting with input vector $\vec{i} = (i_1, \ldots, i_n)$ and ending with input vector $\vec{i'} = (i'_1, \ldots, i'_n)$. By requirement on the list ℓ, $\{1 \le j \le n \mid i_j \ne i'_j\} \subseteq \ell$, thus $i_{j_{|\ell|+r}} = i'_{j_{|\ell|+r}}$ for all $1 \le r \le z$. Furthermore, for any $s';\vec{i'} \in S \times I$, a $s'';\vec{i'} \in S \times I$ exists due to deadlock-freedom such that $s';\vec{i'} \xrightarrow{j_{|\ell|+r}} s'';\vec{i'}$ for all $1 \le r \le z$. Hence, because $i_{j_{|\ell|+r}} = i'_{j_{|\ell|+r}}$ and the state s' was reachable with input vector $\vec{i'}$, the fixed-point property can be applied, yielding $s'' = s'$. Repeating this for all $1 \le r \le z$ shows that $\overset{\ell'}{\twoheadrightarrow}_k^{(a)} = \overset{\ell}{\twoheadrightarrow}_k^{(a)}$.

Because $\twoheadrightarrow_k^{(a)}$ is order-independent, $\overset{\ell'}{\twoheadrightarrow}_k^{(a)} = \overset{\mathrm{sort}(\ell')}{\twoheadrightarrow}_k^{(a)}$. Note that ℓ', and therefore also $\mathrm{sort}(\ell')$, might contain duplicates. However, for any computation sequence of the form $s_0;\vec{i}_0 \xrightarrow{j} s_1;\vec{i}_1 \xrightarrow{j} s_2;\vec{i}_2$ that occurs as part of $\xrightarrow{\mathrm{sort}(\ell')}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k$, it holds that $\vec{i}_1 = \vec{i}_2$, thus the fixed-point property can be applied again, showing $s_1 = s_2$. Thereby, all duplicates from $\mathrm{sort}(\ell')$ can be removed, which results in the list $1 : \cdots : n \in \mathcal{L}_n$. This proves the theorem, since $\xrightarrow{\ell}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k = \xrightarrow{\ell'}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k = \xrightarrow{\mathrm{sort}(\ell')}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k = \xrightarrow{1:\cdots:n}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k = \xrightarrow{1:\cdots:n}\mathrel{\mkern-14mu}\rightarrow_k$. □

To perform a macro-step of a deadlock-free transition system satisfying the fixed-point property with an order-independent transition relation $\twoheadrightarrow^{(a)}_k$ for $a \in \{0, 1, 2\}$ and possibly changing lists ℓ, it therefore suffices to only consider the single relation $\xrightarrow{1:\cdots:n}\mathrel{\mkern-14mu}\rightarrow_k$. This especially allows to reduce evaluations with lists of arbitrary length to evaluation with the fixed length n. If the relation is also allowed to depend on the input values, then it even suffices to only consider the changed inputs once, as the unchanged inputs do not affect the final state.

Corollary 4.2.23. *Let $T = (S, I, \delta)$ be a deadlock-free transition system with n inputs which satisfies the fixed-point property, let $s, s' \in S$, $\vec{i} = (i_1, \ldots, i_n)$, $\vec{i'} = (i'_1, \ldots, i'_n) \in I$, and $0 \leq a \leq 2$ such that $\twoheadrightarrow^{(a)}_k$ is order-independent. Define $\ell_c = j_1 : \cdots : j_z$, where $\{j_1, \ldots, j_z\} = \{1 \leq j \leq n \mid i_j \neq i'_j\}$.*

Then for all lists $\ell \in \overline{\mathcal{L}}_n$ satisfying the requirements of $\xrightarrow{\ell}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k$, $s;\vec{i} \xrightarrow{\ell}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k s';\vec{i'}$ iff $s;\vec{i} \xrightarrow{\ell_c}\mathrel{\mkern-14mu}\rightarrow^{(2)}_k s';\vec{i'}$.

Proof. Follows from Theorem 4.2.22, since $\xrightarrow{\ell}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k = \xrightarrow{1:\cdots:n}\mathrel{\mkern-14mu}\rightarrow_k = \xrightarrow{\ell_c}\mathrel{\mkern-14mu}\rightarrow^{(2)}_k$ for all lists $\ell \in \overline{\mathcal{L}}_n$ that satisfy the requirements of $\xrightarrow{\ell}\mathrel{\mkern-14mu}\rightarrow^{(a)}_k$. □

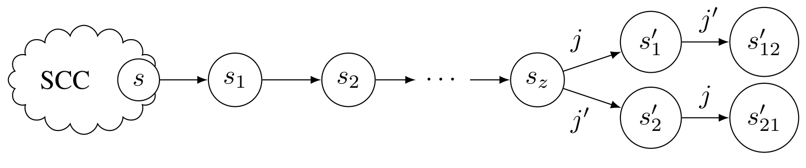
Restricting to Long-Run Behavior

As already stated previously, only the long-run behavior of a transition system is interesting in the given context of hardware. Thus, states that only occur during the initial phase should not be considered relevant for order-independence.

In the previous section, the parameter k was used to indicate the required length of a path leading to a relevant counter-example state. Such a counter-example state is the state from which two traces of applying inputs in different orders lead to two different final states. In Examples 4.2.15 and 4.2.16 it was already shown that by increasing this parameter, the previously found counter-examples could be ruled out, thus proving the transition systems to be order-independent in the long-run.

In general, the long-run behavior mainly consists of states contained in the non-trivial strongly connected components (SCCs) of the transition system, i.e., those states which can occur infinitely often in a trace. An SCC is called *non-trivial* if it contains at least one transition. Thereby, SCCs consisting of a single state are disregarded except for those with a self-loop. In this setting, the notion of order-independence has to be adapted. Previously, equal computations of length n (the number of inputs) were required. However, when restricting to non-trivial SCCs, using the same requirement gives rise to unwanted behavior, since a non-trivial SCC might be left after an arbitrary number of steps, transitioning to states (not belonging to any non-trivial SCC) that are eventually order-dependent. This problem is demonstrated in the below example.

Example 4.2.24. Consider a vector-based transition system $T = (S, I, \delta)$ with n inputs having the shape sketched below.



It is assumed that $z \geq n$, i.e., the path sticking out to the right is branching after more than n steps when starting in the state s. This latter state s is assumed to be deterministic, i.e., its successors are uniquely defined by the input vector applied. Furthermore, s is contained in the non-trivial SCC on the left and the transition system, when restricted to the states of that SCC, shall be order-independent.

If order-independence is considered to be the property that computations of length n starting in a state of a non-trivial SCC always agree on the final state, then also the full transition system is order-independent. This holds, because the branching in state s_z can only be reached after more than n steps by construction.

The above example is however undesired, as in the long-run behavior the SCC may be traversed an unbounded number of times, until eventually the path through s_1 is taken, which might eventually give rise to order-dependent behavior. Thus, all states reachable from some state contained in a non-trivial SCC should be order-independent in the sense of Definition 4.2.9. This is expressed by the relation $\xrightarrow{\ell}\!\!\!\!\rightarrow_{\text{SCC}}$, which is defined as $\xrightarrow{\ell}\!\!\!\!\rightarrow_{\text{SCC}} = \{(s;\vec{i}, s';\vec{i}') \mid s;\vec{i} \xrightarrow{\ell}\!\!\!\!\rightarrow_0 s';\vec{i}' \text{ and } s_0;\vec{i}_0 \rightarrow^* s;\vec{i} \text{ for some } s_0;\vec{i}_0 \in S' \text{ and non-trivial SCC } S' \text{ of } T^I\}$. The order-independence property can then again be expressed as $\xrightarrow{\ell}\!\!\!\!\rightarrow_{\text{SCC}} = \xrightarrow{\text{sort}(\ell)}\!\!\!\!\rightarrow_{\text{SCC}}$. Applied to the above Example 4.2.24, the desired order-dependence holds, since state s_{z-n+2} is reachable from state s contained in a non-trivial SCC and gives rise to two order-dependent evaluations ending in states s'_{12} and s'_{21}, respectively.

To approach the relation $\twoheadrightarrow_{\text{SCC}}$, it is proved next that for a state $s;\vec{i} \in S \times I$ the property $k\text{R}(s;\vec{i})$ is implied by the property $k{+}1\text{R}(s;\vec{i})$. Thus, the parameter k can be increased iteratively to check for order-independence.

Lemma 4.2.25. *Let $T = (S, I, \delta)$ be a transition system with n inputs.*

The relation $\twoheadrightarrow_k$ is order-independent, iff $\twoheadrightarrow_{k'}$ is order-independent for some $k' \leq k$.

Proof. The "only-if" direction obviously holds by choosing $k' = k$.

To prove the "if" direction, let $\twoheadrightarrow_{k'}$ be order-independent for $k' \leq k$. Thus, $s;\vec{i} \xrightarrow{\ell}\!\!\!\!\rightarrow_{k'} s';\vec{i}' \iff s;\vec{i} \xrightarrow{\text{sort}(\ell)}\!\!\!\!\rightarrow_{k'} s';\vec{i}'$ for all $s;\vec{i}$, $s';\vec{i}' \in S \times I$ such that $k'\text{R}(s;\vec{i})$. Let $s;\vec{i} \in S \times I$ such that $k\text{R}(s;\vec{i})$. Then, there exist k predecessor states of $s;\vec{i}$, hence also k' ones for any $k' \leq k$. Therefore, it also holds that $s;\vec{i} \xrightarrow{\ell}\!\!\!\!\rightarrow_k s';\vec{i}' \iff s;\vec{i} \xrightarrow{\text{sort}(\ell)}\!\!\!\!\rightarrow_k s';\vec{i}'$, which proves the lemma. □

Since the set S of states is finite, a decision procedure can be devised that determines whether the relation $\twoheadrightarrow_{\text{SCC}}$ of a vector-based transition system is order-independent or not. Let $m = |S \times I|$ be the number of states in T^I. Then, a state $s;\vec{i} \in S \times I$ is reachable from a state in a non-trivial SCC iff $m\text{R}(s;\vec{i})$ holds. In case order-independence does not hold for such a state then it constitutes a counter-example to order-independence of $\twoheadrightarrow_{\mathrm{SCC}}$. Using the above lemma, an iterative approach can be used, which stops as soon as order-independence for some $k \leq m$ has been proven or a counter-example has been found.

Note however that only Lemma 4.2.12 may be used to conclude order-independence, i.e., one must not use Lemma 4.2.13 to conclude order-dependence in case $j \overset{\downarrow k}{\diamond} j'$ does not hold for some $1 \leq j < j' \leq n$. This is the case, since Lemma 4.2.13 additionally requires deadlock-freedom and the fixed-point property. When also considering these requirements, then any one-step reachable state has a self-loop, as shown next.

Lemma 4.2.26. *Let $T = (S, I, \delta)$ be a deadlock-free vector-based transition system with n inputs having the fixed-point property.*

If for some $s_0;\vec{\imath}_0, s;\vec{\imath} \in S \times I$ and $1 \leq j \leq n$, $s_0;\vec{\imath}_0 \xrightarrow{j} s;\vec{\imath}$, then $s;\vec{\imath} \xrightarrow{j'} s;\vec{\imath}$ for all $1 \leq j' \leq n$.

Proof. Let $s_0;\vec{\imath}_0 \xrightarrow{j} s;\vec{\imath}$. Since T is deadlock-free, there exists for every $1 \leq j' \leq n$ an $s' \in S$ such that $s;\vec{\imath} \xrightarrow{j'} s';\vec{\imath}$. Hence, by the fixed-point property it holds that $s = s'$. □

Thus, in a vector-based transition system satisfying these two additional properties, every state is contained in a non-trivial SCC and only one-step reachability has to be considered. This is expressed formally in the below theorem.

Theorem 4.2.27. *For a deadlock-free vector-based transition system $T = (S, I, \delta)$ with n inputs satisfying the fixed-point property the relation $\twoheadrightarrow_{\mathrm{SCC}}$ is order-independent, iff $j \overset{\downarrow 1}{\diamond} j'$ for all $1 \leq j < j' \leq n$.*

Proof. The "if" direction follows from Lemma 4.2.12.

The "only-if" direction is proved indirectly. To this end, assume $j \overset{\downarrow 1}{\diamond} j'$ does not hold for some $1 \leq j < j' \leq n$. Thus, by Lemma 4.2.13, $\twoheadrightarrow_1$ is not order-independent, i.e., there exist states $s;\vec{\imath}$, $s';\vec{\imath}' \in S \times I$ and permutations $\ell, \ell' \in \Pi_n$ such that $s;\vec{\imath} \overset{\ell}{\twoheadrightarrow}_1 s';\vec{\imath}'$ and not $s;\vec{\imath} \overset{\ell'}{\twoheadrightarrow}_1 s';\vec{\imath}'$. Furthermore, $s;\vec{\imath}$ is 1-step reachable, i.e., there exists a predecessor $s_0;\vec{\imath}_0 \in S \times I$ and $1 \leq j \leq n$ such that $s_0;\vec{\imath}_0 \xrightarrow{j} s;\vec{\imath}$. Hence, Lemma 4.2.26 can be applied to obtain that $s;\vec{\imath} \xrightarrow{j'} s;\vec{\imath}$ for all $1 \leq j' \leq n$, which implies that $s;\vec{\imath}$ must be contained in a non-trivial SCC of T^I. Therefore, $\twoheadrightarrow_{\mathrm{SCC}}$ is not order-independent, which proves the theorem. □

Similarly, also Theorem 4.2.14 can always be restricted to the one-step reachable commuting diamond property.

Corollary 4.2.28. *For a deadlock-free vector-based transition system $T = (S, I, \delta)$ with n inputs satisfying the fixed-point property and $k \in \mathbb{N}$, the relation $\twoheadrightarrow_k$ is order-independent, iff $j \overset{\downarrow 1}{\diamond} j'$ for all $1 \leq j < j' \leq n$.*

Proof. The "if" direction follows from Lemmas 4.2.12 and 4.2.25. The "only-if" direction holds, because any state starting a counter-example to the one-step reachable commuting diamond property is reachable from an SCC due to Theorem 4.2.27, thus it is k-step reachable for any $k \in \mathbb{N}$. □

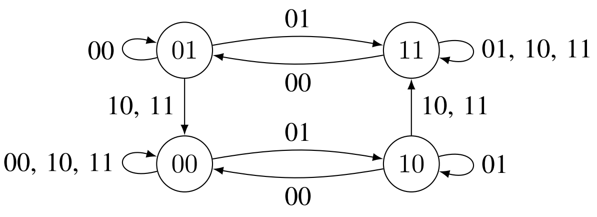


Figure 4.5: D flip-flop as a transition system

Application to Transistor Netlists

To show that the method presented in this section is indeed applicable to transistor netlist representations of hardware cells, order-independence of the transistor netlist descriptions contained in the Nangate Open Cell Library [Nan08] is investigated. For this purpose, a netlist is represented as a set of *fixed-point equations*. These are the result when applying the method of [Bry87], cf. Section 2.2.

To give a formal description of fixed-point equations, let V_S and V_I be two disjoint sets of variables, whose values are in some domain U (usually the Booleans $\mathbb{B}$ or the ternary values $\mathbb{T}$). Furthermore, let $m, n \in \mathbb{N}$, $\vec{s}^v = (s_1^v, \dots, s_m^v) \in V_S^m$ with all s_j^v pairwise disjoint, and $\vec{i}^v = (i_1^v, \dots, i_n^v) \in V_I^n$ with all i_j^v pairwise disjoint. Then a set $\mathcal{E} = \{s_1^v \equiv f_1(\vec{i}^v, \vec{s}^v), \dots, s_m^v \equiv f_m(\vec{i}^v, \vec{s}^v)\}$, with functions $f_j : U^n \times U^m \to U$ for $1 \le j \le m$ is called a set of *fixed-point equations*, iff all of these functions satisfy the following *local fixed-point property*, requiring for all $1 \le j \le m$, all $\vec{i} \in U^n$, and all $\vec{s} \in U^m$ that

$$f_j(\vec{i}, \vec{s}) = f_j(\vec{i}, (f_1(\vec{i}, \vec{s}), \dots, f_m(\vec{i}, \vec{s}))).$$

Such a set of fixed-point equations is interpreted as a vector-based transition system $T(\mathcal{E}) = (U^m, U^n, \delta)$, where $\vec{s} \xrightarrow{\vec{i}}_{T(\mathcal{E})} \vec{s'}$, with $\vec{s'} = (s'_1, \dots, s'_m)$, iff $s'_j = f_j(\vec{i}, \vec{s})$ for all $1 \le j \le m$. Again, the subscript $T(\mathcal{E})$ is left out if the set of fixed-point equations is clear from the context. Note that $T(\mathcal{E})$ is deterministic, i.e., for every $\vec{s}$ and every $\vec{i}$ there exists exactly one $\vec{s'}$ such that $\vec{s} \xrightarrow{\vec{i}} \vec{s'}$.

The following example presents the set of fixed-point equations extracted from the transistor netlist of a D flip-flop.

Example 4.2.29. Consider the following set of fixed-point equations modeling a D flip-flop, where $V_S = \{\mathrm{iq}, \mathrm{q}\}$, $V_I = \{\mathrm{ck}, \mathrm{d}\}$, and the domain of the values of these variables are the Booleans $\mathbb{B}$.

$$\begin{array}{rcl} \mathrm{iq} & \equiv & \neg\mathrm{ck} \wedge \mathrm{d} \ \vee \ \ \mathrm{ck} \wedge \mathrm{iq} \\ \mathrm{q} & \equiv & \mathrm{ck} \wedge \mathrm{iq} \ \vee \ \neg\mathrm{ck} \wedge \mathrm{q} \end{array}$$

These equations describe the transition system depicted in Figure 4.5, where the state variables are concatenated in the order iq, q and the inputs in the order ck, d.

This transition system is deterministic, hence deadlock-free, and satisfies the fixed-point property. Furthermore, it is order-dependent: for example, in state 00 it holds that $00;10 \xrightarrow{1} 00;00$ and $00;00 \xrightarrow{2} 10;01 \xrightarrow{1} 11;11$, whereas $00;00 \xrightarrow{1} 00;10 \xrightarrow{2} 00;11$. This shows that it matters for a flip-flop whether first the data input d changes and then the clock ck, which corresponds to the first trace and sets the output q to the new value of input d, or vice versa, which corresponds to the second trace and sets the output q to the old value of input d.

For the special case of a transition system that stems from a set of fixed-point equations, the required global fixed-point property always holds, as will be shown next.

Lemma 4.2.30. *Every set of fixed-point equations has the fixed-point property.*

Proof. Let $\mathcal{E} = \{s_1^v \equiv f_1(\vec{i^v}, \vec{s^v}), \dots, s_m^v \equiv f_m(\vec{i^v}, \vec{s^v})\}$ be a set of fixed-point equations and let $\vec{s}_1;\vec{i^p} \xrightarrow{j} \vec{s}_2;\vec{i} \xrightarrow{j'} \vec{s}_3;\vec{i}$. Assume that $\vec{s}_2 = (s_{2,1}, \dots, s_{2,m}) \neq (s_{3,1}, \dots, s_{3,m}) = \vec{s}_3$. Then $1 \leq j \leq m$ exists such that $s_{2,j} \neq s_{3,j}$.

By definition, $\vec{s}_2 = (f_1(\vec{i}, \vec{s}_1), \dots, f_m(\vec{i}, \vec{s}_1))$. Since $\mathcal{E}$ is a set of fixed-point equations, the following furthermore holds for the j-th component:

$$\begin{aligned} s_{2,j} &= f_j(\vec{i}, \vec{s}_1) \\ &= f_j(\vec{i}, (f_1(\vec{i}, \vec{s}_1), \dots, f_m(\vec{i}, \vec{s}_1))) = f_j(\vec{i}, \vec{s}_2) \end{aligned}$$

Also by definition, $s_{3,j} = f_j(\vec{i}, \vec{s}_2)$ and hence $s_{3,j} = s_{2,j}$. This is a contradiction to the initial assumption, which proves the lemma. □

To check order-independence of such a set of fixed-point equations, the domain U, which is usually either the set of Booleans $\mathbb{B}$ or the ternary values $\mathbb{T}$, is encoded as Boolean vectors. Then for each pair of inputs, a pair of BDDs is constructed, representing the two sides of the one-step reachable diamond property in Definition 4.2.11, with $k = 1$. Thus, for every two input coordinates $1 \leq j < j' \leq n$ and state coordinate $1 \leq p \leq m$, a BDD representing the function $f_p(\vec{i}[j := v, j' := w], f(\vec{i}[j := v], f(\vec{i}, \vec{s})))$ and a BDD representing $f_p(\vec{i}[j := v, j' := w], f(\vec{i}[j' := w], f(\vec{i}, \vec{s})))$ are constructed, for arbitrary values v and w. Here, the abbreviation $f(\vec{i}, \vec{s}) = \big(f_1(\vec{i}, \vec{s}), \dots, f_n(\vec{i}, \vec{s})\big)$ is used and it is assumed for simplicity that the domain U is the set of Booleans (otherwise, two vectors of BDDs are constructed, representing the encoding of the domain U).

If all such pairs of BDDs are equal, then order-independence has been proven due to Theorem 4.2.14, since then $j \overset{1}{\diamond} j'$ holds for all $j \neq j'$. Furthermore, by Theorem 4.2.27, the transition system is order-independent for any state reachable from a non-trivial SCC, which represent the long-run behavior. Otherwise, a set of counterexample states is determined, which can be obtained by computing the xor of the unequal BDDs. Particularly, for this application it was found that including the one-step reachability into the requirement, i.e., restricting to the long-run behavior, removes many spurious counterexamples, which were due to certain dependencies of the internal signals on the input signals. This corresponds to a stabilization of the netlist before applying the first input vector, i.e., all transistors are evaluated w.r.t. the previous input vector until there are no more changes. Restricting to one-step reachable states is incorporated into the above BDD construction by starting with the state $f(\vec{i}, \vec{s})$, which is a stable state due to Lemma 4.2.26.

Experimental Results

The presented method has been applied to the transistor netlists of the 12 sequential cells in the Nangate Open Cell Library [Nan08], whose corresponding functional

Table 4.3: Order-Independence of Transistor Netlists in the Nangate Open Cell Library

Cell	**# State Vars**	**# Inp. Pairs**	**# Ord-Dep Prs**	**Time $\overset{\downarrow 1}{\Diamond}$ [s]**	**Time all orders [s]**
CLKGATE	2	1	1	0.02	0.07
CLKGATETST	2	3	2	0.02	0.23
DFF	4	1	1	0.01	0.19
DFFR	4	3	2	0.02	0.26
DFFS	4	3	2	0.02	0.35
DFFRS	4	6	4	0.08	1.43
DLH	2	1	1	0.01	0.07
DLL	2	1	1	0.01	0.04
SDFF	4	6	3	0.08	1.70
SDFFR	4	10	4	0.16	18.17
SDFFS	4	10	4	0.14	18.05
SDFFRS	4	15	6	0.21	359.66

descriptions were used in the experiments presented in Section 4.1. All external inputs to the cells were considered to be binary, i.e., to be either 0 or 1.

Table 4.3 shows the results of these experiments. In its first column, the name of the cell is listed. The second column gives the number of state variables that resulted from encoding the transistor netlist as vector-based transition system. Note that for implementation reasons outputs are required to be a variable, i.e., they must not be a combinational function of some state variables. This increases the number of state variables in some cases. The third column of Table 4.3 presents the number of input pairs that had to be checked for each state variable, and the fourth column gives the number of pairs that were found to be order-dependent for some state variable. The fifth column gives the overall time taken for the order-dependency analysis based on comparing pairs of inputs and checking the one-step reachable commuting diamond property. These times are contrasted with the last column, showing the time taken to check order-independence by checking the property of Lemma 4.2.10, which enumerates all possible orders.

It should be remarked that the number of input pairs considered is different from the numbers that had to be considered in the VeriCell experiments of Section 4.1, which were presented in Table 4.1. This is because in VeriCell, the UDPs can be identified as the state holding elements, therefore only their inputs need to be considered. In a transistor netlist, it is not straightforward to identify such elements, hence all inputs are considered. Furthermore, this directly takes into account the functions computing inputs to state holding elements, something that was only considered during the reachability analysis for VeriCell descriptions.

As can be seen from the results, order-independence of transistor netlists can be analyzed in less than a quarter of a second for every cell contained in an industrial cell library, using the one-step reachable commuting diamond property. Therefore, this analysis can for example be used as a fast preprocessing step to compute the independence relation needed for partial order reduction [Pel98], which allows to reduce the state space that has to be explored when checking other properties.

When comparing these times with those it took for a naive approach to determine order-independence, it can clearly be seen that the method based on the one-step reachable commuting diamond property is far more scalable. For small cells, the times for the naive approach, listed in the last column of Table 4.3, are still acceptable. But for larger cells, these times quickly increase. Especially for the largest cell in the Nangate Open Cell Library, the cell `SDFFRS`, it took the naive approach almost 6 minutes to analyze order-dependencies, whereas the improved approach only took a fraction of a second.

4.3 Using Non-Determinism to Reduce Power Consumption

Non-determinism in cell libraries was considered undesired in the previous two sections, since it can lead to different computation results. This section presents a technique that makes use of the non-determinism contained in cell libraries, by allowing a more efficient analysis of power consumption and by choosing among functionally equivalent orders those that minimize the power consumption. This analysis focuses on the transistor netlist descriptions of cells, since these are closest to the final implementation. The technique could also be used for Verilog descriptions, however there it is unclear what a good abstract measure of power consumption could be. This is the case since most of the logic implemented by a cell is described as UDP, which is not easily mapped to an implementation using transistors, which will be the elements consuming power in the final chip.

Preliminaries

As in the previous Section 4.2, transistor netlists are represented as vector-based transition systems. In such a transition system, only the one-input restricted traces are of interest, i.e., only one input may change at a time, as was defined in Definition 4.2.3. Thus, also in this section the one-input restricted vector-based transition system representation T^I of a vector-based transition system T, which was defined in Definition 4.2.5, is re-used.

The order of evaluating changed inputs is given by permutations from Π_n, which is again seen as a subset of the lists $\mathcal{L}_n$ containing lists in which each of the numbers occurs at most once. Given such a list $\ell = j_1 : \ldots j_z \in \mathcal{L}_n$, a *sublist* of ℓ is defined as $\ell[a \mathinner{..} b] = j_a : \ldots j_b$ for $1 \leq a \leq b \leq z$, and $\ell[a] = \ell[a \mathinner{..} a] = j_a$.

Reducing Input Vector Orders for Power Analysis

An improvement in power analysis is achieved by grouping, in equivalence classes, those orders that have the same power characteristics. For such orders, it is sufficient to only consider one member of the equivalence class. Important for such equivalence classes is to result in the same state in order not to affect the functionality of the netlist. This was already dealt with in the order-independence analysis of vector-based transition systems, presented in Section 4.2. To also take the dynamic power consumption into account, a *power-extended vector-based transition system* is defined.

Definition 4.3.1. The *power-extended vector-based transition system* $T_p = (S_p, I, \to_p)$ of a vector-based transition system $T = (S, I, \delta)$ is defined $S_p = \mathbb{R} \times S$ and $w;s \xrightarrow{\vec{i}}_p w{+}p(s, s');s'$ for $s \xrightarrow{\vec{i}} s'$. The function $p : S \times S \to \mathbb{R}$ computes a weighted number of wire chargings given the source and target states.

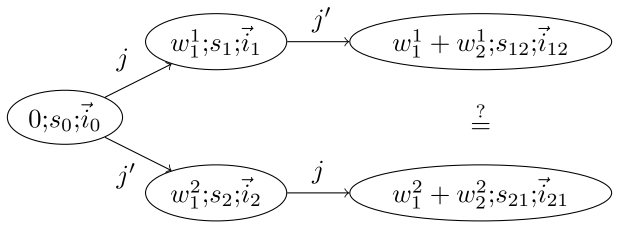


Figure 4.6: Evaluation of two input coordinates j and j'

The added first component of the states, a number, is used to sum the weights of the charged wires (which are interpreted as the consumed power) during an evaluation. The definition allows for weighted wire charging in order to cater for node capacitance, by appropriately instantiating the function p. However, throughout the rest of this section, the weights of all wire chargings are assumed to be equal. Hence, in the remainder the sum of weights denotes the number of wire chargings. Note that a vector-based transition system does not assume a particular shape of the states, so a power-extended vector-based transition system is still a vector-based transition system.

To determine the weights, initially the number in the newly added first component is set to 0, indicating that no chargings have taken place yet. Next, two inputs are selected (which are identified by their position in the input vector, as in the transition system T^I) and are applied in both possible orders. Finally, it is checked whether the resulting states for the two orders are equal or not. Thus, for a state s_0 that was reached with an input vector $\vec{i}_0$, the evaluation starts in the state $0;s_0;\vec{i}_0$. Then, the two input changes, here denoted with substitutions $[j := v_1']$ and $[j' := v_2']$ where $j \neq j'$, are applied in both possible orders, leading to the two evaluations depicted in Figure 4.6.

As indicated in Figure 4.6, it is then to be checked whether the two states $w_1^1 + w_2^1;s_{12};\vec{i}_{12}$ and $w_1^2 + w_2^2;s_{21};\vec{i}_{21}$ are equal or not. First, it is noted that the input vectors $\vec{i}_{12}$ and $\vec{i}_{21}$ are equal, as they are constructed by updating positions j and j' in $\vec{i}_0$ with the same values. Formally, this holds because for $j \neq j'$, $\vec{i}_{12} = \vec{i}_0[j := v_1'][j' := v_2'] = \vec{i}_0[j' := v_2'][j := v_1'] = \vec{i}_{21}$. Thus, only the remaining parts of the states have to be compared. Checking that the states s_{12} and s_{21} are equal is the same as order-independence, i.e., checking that the order of these two inputs does not affect the functionality. By requiring that $w_1^1 + w_2^1 = w_1^2 + w_2^2$, it is additionally required that the order of the two inputs also does not cause different power consumptions.

In the above check, only the order of two inputs was considered. However, by extending the result of Section 4.2, it can be shown that checking order-independence for two inputs is both necessary and sufficient to establish order-independence for traces of full input length in transistor netlists. To denote such traces of full input length, the relation $\twoheadrightarrow_k$ given in Definition 4.2.8 is used again. In that definition, it is required that the initial state is k-step reachable, to rule out transient initial states that can only occur at boot-up and will never be reached again. However, as was shown in Section 4.2, for transistor netlists it is sufficient to consider the case $k = 1$.

A vector-based transition system T with n inputs is called *order-independent*, iff for all $\pi, \pi' \in \Pi_n$ it holds that $\overset{\pi}{\twoheadrightarrow}_k = \overset{\pi'}{\twoheadrightarrow}_k$, cf. Definition 4.2.9. To check order-independence, the relation $\overset{\downarrow k}{\diamond}$ was introduced in Definition 4.2.11, called the *k-step reachable commuting diamond property*. It relates two input positions $1 \leq j \neq j' \leq n$, if state $s_0;\vec{i}_0$ is k-step reachable and $s_0;\vec{i}_0 \xrightarrow{j} \circ \xrightarrow{j'} s_{12};\vec{i}_{12}$ iff $s_0;\vec{i}_0 \xrightarrow{j'} \circ \xrightarrow{j} s_{12};\vec{i}_{12}$. This is similar to the situation depicted in Figure 4.6, where, in the general setting, the first component summing the power consumption is removed, i.e., it is required that the initial state $s_0;\vec{i}_0$ is k-step reachable and that the reached states s_{12} and s_{21} are equal.

It was shown in Theorem 4.2.14 that the k-step reachable commuting diamond property is necessary and sufficient for order-independence of deadlock-free vector-based transition systems that also have the fixed-point property. Deadlock-freedom requires that for every current state and every possible input transition, a next state can be computed. This is also the case for transistor netlists represented as power-extended vector-based transition systems, as they always compute a next state for any input vector. The second requirement, the fixed-point property, demands that a reached state is stable; applying the same input vector twice does not result in a different state as when the input vector is only applied once. Vector-based transition systems constructed from a transistor netlist using the algorithm of [Bry87] always have the fixed-point property, as was shown in Section 4.2. Since an unchanged state means that also the number of wire chargings does not change, this also holds in for power-extended vector-based transition systems. Therefore, in the remainder these two requirements are assumed to always hold, without mentioning this explicitly.

To summarize, also for power-extended vector-based transition systems only pairs of inputs have to be analyzed, instead of complete sequences, to determine order-independence of the power-extended vector-based transition system. This entails equivalent power consumption due to the construction of the states. Thereby, the number of required checks is reduced drastically from $n!$, i.e., the number of all permutations, to $\frac{n^2-n}{2}$, the number of all pairs of inputs. Furthermore, since the transition systems considered in this section are always deadlock-free and have the fixed-point property, it is sufficient to only consider one-step reachability instead of k-step reachability.

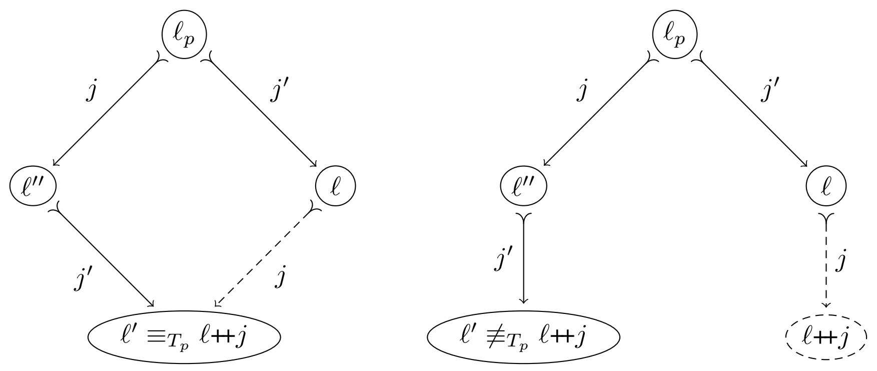
Power-Equivalence Relation on Orders

Full order-independence of a power-extended vector-based transition system would mean that all orders always have the same power consumption. Of course, this is neither expected in any useful transistor netlist, nor is it of much practical relevance. Therefore, an equivalence relation on orders is to be defined that groups together those subsets of orders having the same number of wire chargings. This relation, formally defined below, is called *power independence*.

Definition 4.3.2. Let $T = (S, I, \delta)$ be a vector-based transition system with n inputs.

The relation $\leftrightarrow_T$ on $\mathcal{L}_n$ is defined as $\ell \leftrightarrow_T \ell'$ iff the lists ℓ and ℓ' are equal except for swapped positions $\ell'[j+1] = \ell[j]$ and $\ell'[j] = \ell[j+1]$, for which the one-step reachable commuting diamond property holds (i.e., $\ell[j] \overset{\downarrow 1}{\diamond} \ell[j+1]$).

Using relation $\leftrightarrow_T$, the equivalence relation $\equiv_T$ on $\mathcal{L}_n$ is defined as the reflexive transitive closure of $\leftrightarrow_T$. If $\ell \equiv_T \ell'$, then ℓ and ℓ' are also called *(power-)independent*.



(a) $j \overset{\downarrow 1}{\diamond} j'$ (b) $j \overset{\downarrow 1}{\not\diamond} j'$

Figure 4.7: Construction of the power-independence DAG

In the above definition, a general vector-based transition system is used. If this is a power-extended one, then the relation is called power-independence, otherwise it is only called independence of lists. For the power-independence relation the following result holds, showing that it indeed groups together those orders that have equal (functional and power consumption) behavior.

Lemma 4.3.3. *Let $T_p = (S_p, I, \rightarrow_p)$ be a power-extended vector-based transition system with n inputs and let $\pi, \pi' \in \Pi_n$ be power-independent.*

Then for traces $0;s_0;\vec{i}_0 \overset{\pi}{\twoheadrightarrow}_1 w_1;s_1;\vec{i}$ and $0;s_0;\vec{i}_0 \overset{\pi'}{\twoheadrightarrow}_1 w_2;s_2;\vec{i}$ of T_p it holds that $w_1 = w_2$ and $s_1 = s_2$.

Proof. Follows by an induction on the number of swapped input coordinates to reach π' from π. □

Thus, to characterize a cell, one only has to choose one representative from each power-independent equivalence class and measure the power consumption for this order. All other orders in this equivalence class will have the same power consumption and hence do not have to be considered.

To obtain the different orders that have to be considered, the so-called *power-independence DAG* (directed acyclic graph) is constructed. It enumerates all equivalence classes of the power-independence relation $\equiv_{T_p}$.

Definition 4.3.4. Let $T_p = (S_p, I, \rightarrow_p)$ be a power-extended vector-based transition system with n inputs.

The *power-independence DAG* $G_i = (V_i, \rightarrowtail_i)$ of T_p is defined as $V_i \subseteq \mathcal{L}_n$ with root $\mathsf{nil} \in V_i$ and for $1 \leq j \leq n$ with $j \notin \ell$, $\ell \rightarrowtail_i \ell'$ for some unique $\ell' \equiv_{T_p} \ell +\!\!+ j$.

Note that every edge $\ell \rightarrowtail_i \ell'$ with $\ell' \equiv_{T_p} \ell +\!\!+ j$ can be regarded as labeled by input j. Hence, they will also be denoted $\ell \overset{j}{\rightarrowtail}_i \ell'$. However, these labels need not be explicitly added, since this is always the single input by which the two lists of the start and the end node of the edge differ.

To construct the power-independence DAG G_i, one starts with the single root of this DAG, nil, indicating that initially no inputs have been considered yet. The construction of the DAG then proceeds in a breadth-first fashion: For every *leaf node* ℓ (which is a node without outgoing edges) and every input j that has not yet been considered (i.e., which is not contained in ℓ), an edge is added to that leaf node. The target node of this edge is determined by looking at the parent nodes of the currently considered leaf. If there exist a parent node ℓ_p reaching the current node ℓ with input j', a node ℓ' reachable from the parent node ℓ_p with list $j{+\!\!+}j'$, and the inputs j and j' are exchangeable, i.e., $j \overset{\downarrow 1}{\diamond} j'$, then the edge is drawn to the existing node ℓ'. This is depicted in Figure 4.7 (a), where the dashed edge is added. Otherwise, if one of the above conditions is violated (i.e., either the inputs cannot be exchanged or the node ℓ' has not been generated yet), a new node $\ell{+\!\!+}j$ is created and an edge drawn there. As an example, the case where for all $\ell_p \overset{j:j'}{\rightarrowtail}{}_i^* \ell'$ it holds that $j \overset{\downarrow 1}{\not\diamond} j'$ is depicted in Figure 4.7 (b). There, the dashed edge and the dashed node are added to the DAG. This process finishes at leaves for which the list of considered inputs contains every input exactly once. It can furthermore be shown that the above construction always yields the power-independence DAG.

Next, it is proven that this DAG exactly distinguishes between the equivalence classes of the power-independence relation $\equiv_{T_p}$.

Theorem 4.3.5. *Let T_p be a power-extended vector-based transition system with n inputs and let $G_i = (V_i, \rightarrowtail_i)$ be its power-independence DAG.*

Then, for all orders $\pi_1, \pi_2 \in \Pi_n$, π_1 and π_2 are power-independent, iff there exist paths $\mathsf{nil} \overset{\pi_1}{\rightarrowtail}{}_i^* \pi$ *and* $\mathsf{nil} \overset{\pi_2}{\rightarrowtail}{}_i^* \pi$ *in G_i for some order $\pi \in \Pi_n$.*

Proof. To prove the "if" direction, one observes that due to the definition of the power-independence DAG in Definition 4.3.4, all nodes on a path, when appending the remaining considered inputs, are power-independent. This directly entails $\pi_1 \equiv_{T_p} \pi \equiv_{T_p} \pi_2$.

The "only-if" direction is proved by an induction over the number of swappings needed to reach π_1 from π_2. If there are none, then $\pi_1 = \pi_2$ and the theorem trivially holds. Otherwise, the induction hypothesis can be applied to π_2 and the order π' resulting from π_1 by undoing the last swapping of j and $j{+}1$. This gives two paths $\mathsf{nil} \overset{\pi'}{\rightarrowtail}{}_i^* \pi$ and $\mathsf{nil} \overset{\pi_2}{\rightarrowtail}{}_i^* \pi$ in the graph. Since the two swapped positions j and $j+1$ are power-independent, and the rest of the orders π_1 and π' are the same, the two paths induced by these two orders must have the diamond shape due to the requirement in Definition 4.3.4, proving that also a path $\mathsf{nil} \overset{\pi_1}{\rightarrowtail}{}_i^* \pi$ exists. □

Thus, to determine the power-independent orders of a given power-extended vector-based transition system, its power-independence DAG is constructed. Due to the above theorem, the lists contained in the leaves of the power-independence DAG are representatives of the different equivalence classes of orders that have to be considered for power characterization, i.e., only one of these orders has to be measured to obtain the real power consumption of all equivalent orders. Therefore, the number of leaves compared to the number of all possible orders is a measure for the reduction obtained by this method.

Selecting Orders to Minimize Power Consumption

Contrary to the goal above, where orders were identified that always have the same dynamic power consumption, now orders shall be identified that are functionally independent, i.e., they do not influence the computation of a next state, but may have different power consumption. Then, by taking the order (one representative order among the functionally equivalent orders) that consumes the least amount of power, the dynamic power consumption of computing the next state can be reduced.

For this purpose, another DAG structure is defined to describe the different possible orders, but now the nodes represent lists that are computing the same next state, i.e., the inputs leading to such a shared node only need to have the diamond property regarding the functionality and not necessarily regarding the power consumption. Furthermore, each node is equipped with a back-pointer that determines which input leads to less power consumption. Then, by traversing the DAG from some leaf to the root following these back-pointers, one can construct the order that computes the same next state but uses minimal power. Below, this intuition is formalized.

Definition 4.3.6. Given a power-extended vector-based transition system $T_p = (\mathbb{R} \times S, I, \rightarrow_p)$ with n inputs, the *power-sum DAG* is defined as $G_s = (V_s, \rightarrowtail_s, \rightsquigarrow_s)$, where $V_s \subseteq \mathcal{L}_n$, $\rightarrowtail_s \subseteq V_s \times V_s$, and $\rightsquigarrow_s \subseteq V_s \times (S \times I \times I) \times V_s$. The root is defined to be $\mathsf{nil} \in V_s$.

The *transition relation* $\rightarrowtail_s$ is defined for every $\ell \in V_s$ and every $1 \leq j \leq n$ as $\ell \rightarrowtail_s \ell'$ for some unique ℓ' such that $\ell{+\!\!+}j \equiv_T \ell'$.

The *back-pointer relation* $\rightsquigarrow_s$ is defined for every $\ell \in V_s$, $s;\vec{i} \in S \times I$, and $\vec{i'} \in I$ as $\mathsf{nil} \overset{s;\vec{i};\vec{i'}}{\not\rightsquigarrow}_s \ell$ and, if $\ell \neq \mathsf{nil}$, $\ell \overset{s;\vec{i};\vec{i'}}{\rightsquigarrow}_s \ell'$ for some unique $\ell' \in V_s$ with $\ell' \overset{j'}{\rightarrowtail}_s \ell$, $\ell' \equiv_T j_1 : \cdots : j_h$, and $0;s;\vec{i} \xrightarrow{j_1}_p \ldots \xrightarrow{j_h}_p \circ \xrightarrow{j'}_p w;s';\vec{i'}$ for which $w \in \mathbb{R}$ is minimal.

Note that the definition of the transition relation of DAG G_s uses independence based on equal states, not the extended power-independence which also checks for equal power consumption. Again, the transition relation can be understood as labeled by an input position $1 \leq j \leq n$, indicating the added input coordinate that has been considered. This was already made use of in the definition of the back-pointer relation, but this position can again be recovered as the single input coordinate by which the two lists differ.

The construction of the power-sum DAG works similarly to the construction of the power-independence DAG. For it, the auxiliary function $w_{\min}$ is used, which assigns to every node and state and input transition the minimal weight that the resulting state can be reached with, i.e., for $\ell \in V_s$ and $s;\vec{i};\vec{i'} \in S \times I \times I$, $w_{\min}(\ell, s;\vec{i};\vec{i'}) = w$ if $0;s;\vec{i} \overset{\ell'}{\twoheadrightarrow}_1 w;s';\vec{i'}$ and w is minimal among all $\ell' \equiv_T \ell$. This can be efficiently read from the back-pointer relation. To complete the function, it is defined as $w_{\min}(\ell, s;\vec{i};\vec{i'}) = \infty$ if no $\ell' \in V_s$ exists such that $\ell \overset{s;\vec{i};\vec{i'}}{\rightsquigarrow}_s \ell'$.

The construction starts with the root node nil and adds nodes in a breadth-first fashion. At each step, for each leaf ℓ of the DAG, an edge is added for every input position $1 \leq j \leq n$ that is not yet contained in ℓ. If there exists a node ℓ' such that $\ell_p \overset{j'}{\rightarrowtail}_s \ell$, $\ell_p \overset{j{+\!\!+}j'}{\rightarrowtail}_s \ell'$, and $j \overset{\downarrow 1}{\diamond}_T j'$, then the edge $\ell \overset{j}{\rightarrowtail}_s \ell'$ is added. Otherwise, a new node $\ell{+\!\!+}j$ is added to the DAG, and the edge $\ell \overset{j}{\rightarrowtail} \ell{+\!\!+}j$ is added. This is

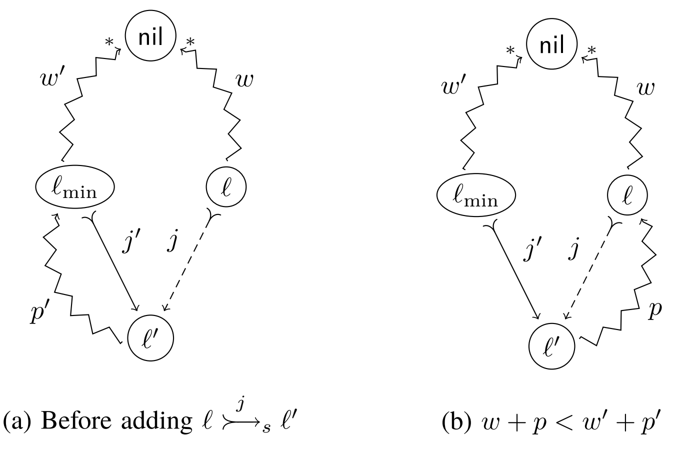


Figure 4.8: Construction of the back-pointer relation in the power-sum DAG for some state and input vector transition $s;\vec{\imath};\vec{\imath}' \in S \times I \times I$

the same construction that was illustrated for the power-independence DAG G_i in Figure 4.7, only here the power consumption is not considered.

A sub-path of a path with minimal weight also is of minimal weight, otherwise such a sub-path could be replaced by one with less weight, giving a contradiction to the initial path having minimal weight. This is used during the construction of the back-pointer relation. When an edge $\ell \overset{j}{\rightarrowtail}_s \ell'$ is added to the DAG, then $\ell' \overset{s;\vec{\imath};\vec{\imath}'}{\rightsquigarrow}_s \ell$ if $s;\vec{\imath} \overset{\ell}{\twoheadrightarrow}_1 s_0;\vec{\imath}_0 \overset{j}{\rightarrow} s';\vec{\imath}'$ and $w_{\min}(\ell, s;\vec{\imath}) + p(s_0, s') < w_{\min}(\ell', s;\vec{\imath})$. Otherwise, $\rightsquigarrow_s$ is left unchanged. Note that if ℓ' is a new node, then the first case always applies, since the sum is always smaller than ∞.

An illustration of the back-pointer construction is shown in Figure 4.8, where the dashed edge $\ell \overset{j}{\rightarrowtail}_s \ell'$ is to be added. Initially, it is assumed that already a node $\ell_{\min}$ exists such that $\ell' \overset{s;\vec{\imath};\vec{\imath}'}{\rightsquigarrow}_s \ell_{\min}$, i.e., the power consumption is minimal if taking the minimal path from the root nil to node $\ell_{\min}$, which is assumed to have weight w', and then extending it by considering coordinate j', whose power consumption is assumed to be p'. This situation is depicted in Figure 4.8 (a). Next, the node ℓ is considered. It should be noted that $\ell' \equiv_T \ell{+\!\!+}j \equiv_T \ell_{\min}{+\!\!+}j'$, as otherwise the edge $\ell \overset{j}{\rightarrowtail}_s \ell'$ would not be drawn. The weight of the minimal path from the root nil to the node ℓ is denoted w and the power consumption of the step from ℓ to ℓ' is denoted p. In case $w + p < w' + p'$, then a new minimal path for ℓ' has been found via ℓ, thus the back-pointer relation is updated as shown in Figure 4.8 (b). Otherwise, the previous path of the back-pointers is still giving the minimal path even after adding the edge $\ell \overset{j}{\rightarrowtail}_s \ell'$, so in that case the back-pointer relation remains as depicted in Figure 4.8 (a).

It can be shown that the above construction yields exactly the power-sum DAG G_s of a power-extended vector-based transition system. The following theorem shows that this DAG identifies all orders that lead to the same state and constructs the order consuming the minimal amount of power.

Theorem 4.3.7. *Let $T_p = (\mathbb{R} \times S, I, \rightarrow_p)$ be a power-extended vector-based transition system with n inputs and power-sum DAG $G_s = (V_s, \rightarrowtail_s, \rightsquigarrow_s)$. Furthermore, let $\pi, \pi' \in \Pi_n$ be some orders and $s;\vec{i};\vec{i'} \in S \times I \times I$ be some state together with previous and next input vectors.*

If $\mathsf{nil} \overset{\pi}{\rightarrowtail}{}^*_s \pi'$, *then a path* $\pi' = \ell_n \overset{s;\vec{i};\vec{i'}}{\rightsquigarrow}_s \dots \overset{s;\vec{i};\vec{i'}}{\rightsquigarrow}_s \ell_0 = \mathsf{nil}$ *exists and* $\pi \equiv_T \pi' \equiv_T \pi''$ *for* $\pi'' = j_1 : \cdots : j_n \in \Pi_n$ *defined by* $\ell_r = \ell_{r-1} +\!\!+ j_r$ *for all* $1 \leq r \leq n$ *such that* $0;s;\vec{i} \overset{\pi''}{\twoheadrightarrow}_1 w;s';\vec{i'}$ *and* w *is minimal.*

Proof. Existence of the back-pointer path and hence of π'' is guaranteed by the (unique) existence of a successor w.r.t. $\rightsquigarrow_s$ for every node that is not the root and since $\ell_r \neq \mathsf{nil}$ for every $1 \leq r \leq n$. The property $\pi \equiv_T \pi' \equiv_T \pi''$ directly follows from the definition of the transition relation of G_s. Finally, minimality of w follows from the definition of the back-pointer relation of G_s. $\square$

Given a cell and its power-sum DAG G_s, one can obtain the order consuming the least amount of power for a given state, input vector transition, and order π in which the inputs are to be changed. This works by first traversing the DAG G_s according to the order π, which will result in a leaf π' of the DAG, satisfying $\pi' \equiv_T \pi$. From that leaf, the back pointer relation is followed upwards to the root, giving (in reverse order, by taking the difference between the nodes along the back-pointer path) another order π'' with $\pi'' \equiv_T \pi' \equiv_T \pi$ which consumes the least amount of power, as was shown in Theorem 4.3.7. Enforcing this order π'' can for example be done by adding delays, which is also proposed in [RDJ96].

Implementation

The above techniques for grouping together orders that have equal power consumption behavior and for determining functionally equivalent orders were implemented in a prototype tool. This tool first parses a SPICE netlist and builds a symbolic vector-based transition system from it using the algorithm of [Bry87], where states consist of a vector of formulas, computing values from the set $\{\mathsf{0}, \mathsf{1}, \mathsf{Z}\}$. The values $\mathsf{0}$ and $\mathsf{1}$ correspond to the logic values false and true, respectively, and represent an active path from a wire in the netlist to the low and high voltage rails, respectively. The third value, Z, represents a floating wire that has neither a path to the low nor to the high voltage rail. As initial state of the netlist, arbitrary values are allowed for all of the wires. The inputs are restricted to the binary values $\mathsf{0}$ and $\mathsf{1}$.

The power consumption of a transition is computed by the function p in Definition 4.3.1. In the implementation, this function is defined as $p(\vec{s}, \vec{s'}) = \left|\{1 \leq j \leq m \mid \rho_j(\vec{s}) = \mathsf{0}, \rho_j(\vec{s'}) = \mathsf{1}\}\right|$ for a netlist consisting of m wires, i.e., it counts the number of wires that transition from $\mathsf{0}$ to $\mathsf{1}$.

Building the power-independence DAG is then performed by first computing the diamond relation for all pairs of inputs (also taking power consumption into account). This is done symbolically using BDDs, requiring a total of $O(m \cdot n^2)$ BDD comparisons for n inputs and m state variables: For every of the $O(m^2)$ pairs of input variables and every of the n state variables, two BDDs are constructed. The first computes the next state after applying the two inputs in one order, the second BDD computes the next state after applying the inputs in the other order. The currently considered pair of inputs has the power-extended diamond relation, iff these pairs of BDDs are equal for all state variables and the total number of wire chargings is the same. From this information, the power-independence DAG can finally be constructed, using the previously presented algorithm.

If the power-sum DAG is to be constructed, then the functional independence relation has to be computed first for all pairs of inputs. This uses the method presented in Section 4.2 and also requires $O(m \cdot n^2)$ BDD comparisons. To construct the power-sum DAG, one needs to keep track of the state to which a list of input coordinate changes leads, to be able to construct its back-pointer relation. For this purpose, the symbolic transition relation is unrolled, i.e., a new transition relation is created that computes, given a starting state and input vector, the state and input vector after changing the inputs in the order of the currently considered node. This is used to create a symbolic formula computing the number of wires charged when adding another input to the list. Finally, a symbolic minimum is computed among these formulas that indicates which parent node leads to minimal power consumption.

Experimental Results

The technique to reduce the number of considered orders and the technique to select an equivalent order that consumes less power were applied to the open-source Nangate Open Cell Library [Nan08]. For each of the contained netlists, the SPICE source was parsed, a transition system created, and the power-independence DAG or power-sum DAG built and traversed to enumerate all equivalence classes. All of these experiments were conducted on a commodity PC equipped with an Intel Pentium 4 3.0 GHz processor and 1 GB RAM running Linux.

Reducing Input Vector Orders

The results for reducing the number of considered orders with different functional or power consumption behavior are presented in Table 4.4, where the first column gives the name of the cell, the second column the number of inputs and wires, the third column the number of all possible orders, and the fourth column shows the number of different equivalence classes in the power-independence DAG together with the time it took to construct it. Finally, the last column demonstrates the achievable power reduction, to be explained below.

For combinational cells, marked with “(c)” in Table 4.4, the results show that by constructing the power-independence DAG the number of orders that have to be considered for power characterization of these cells cannot be reduced. This is usually due to situations in which wires are in one order first discharged only to be finally charged, whereas evaluating them in another order keeps the wires charged during the whole evaluation. Thus, all possible orders have to be considered during power characterization of these cells.

For sequential cells however, some larger savings can be observed, especially for the larger cells. For example, in case of the largest cell in the library, the cell `SDFFRS`, the number of orders to consider could be reduced from 720 to only 288, which is a reduction by 60 %. Especially for sequential cells these savings have an effect, since for these cells the characterization not only has to take the possible input combinations into account, but also the current internal state. Overall, when summing up the absolute number of orders that have to be considered for the sequential cells, a reduction by more than 47 % is obtained. This is especially advantageous for the large cells, as witnessed by the average of the reduction rates of sequential cells, which is only slightly above 16 %. So especially for large sequential cells with lots

Table 4.4: Results for the Nangate Open Cell Library

Cell	**#I / W**	**#Π$_m$**	**|G$_i$| / t [s]**	**|G$_s$| : Avg / t [s]**
AND2 (c)	2 / 3	2	2 / 0.37	1 : 2.5 / 0.37
AND3 (c)	3 / 4	6	6 / 0.46	1 : 3.5 / 0.47
AND4 (c)	4 / 5	24	24 / 0.60	1 : 4.5 / 0.66
AOI211 (c)	4 / 4	24	24 / 0.62	1 : 4.0 / 0.64
AOI21 (c)	3 / 3	6	6 / 0.44	1 : 2.5 / 0.45
AOI221 (c)	5 / 5	120	120 / 0.97	1 : 7.0 / 1.14
AOI222 (c)	6 / 6	720	720 / 1.79	1 : 9.5 / 5.57
AOI22 (c)	4 / 4	24	24 / 0.68	1 : 5.0 / 0.66
BUF (c)	1 / 2	1	1 / 0.33	1 : 0.0 / 0.27
CLKBUF (c)	1 / 2	1	1 / 0.25	1 : 0.0 / 0.29
CLKGATETST	3 / 13	6	6 / 0.68	4 : 3.7 / 0.71
CLKGATE	2 / 11	2	2 / 0.54	2 : 0.0 / 0.54
DFFRS	4 / 24	24	24 / 1.20	12 : 1.1 / 1.82
DFFR	3 / 19	6	4 / 0.78	4 : 0.0 / 0.90
DFFS	3 / 19	6	6 / 0.82	4 : 1.0 / 0.90
DFF	2 / 16	2	2 / 0.63	2 : 0.0 / 0.68
DLH	2 / 9	2	2 / 0.50	2 : 0.0 / 0.52
DLL	2 / 9	2	2 / 0.53	2 : 0.0 / 0.52
FA (c)	3 / 14	6	6 / 0.71	1 : 3.0 / 0.76
HA (c)	2 / 8	2	2 / 0.46	1 : 1.0 / 0.48
INV (c)	1 / 1	1	1 / 0.24	1 : 0.0 / 0.25
MUX2 (c)	3 / 6	6	6 / 0.52	1 : 4.0 / 0.53
NAND2 (c)	2 / 2	2	2 / 0.35	1 : 1.5 / 0.35
NAND3 (c)	3 / 3	6	6 / 0.44	1 : 2.5 / 0.44
NAND4 (c)	4 / 4	24	24 / 0.58	1 : 3.5 / 0.61
NOR2 (c)	2 / 2	2	2 / 0.35	1 : 1.5 / 0.36
NOR3 (c)	3 / 3	6	6 / 0.47	1 : 2.5 / 0.45
NOR4 (c)	4 / 4	24	24 / 0.58	1 : 3.5 / 0.63
OAI211 (c)	4 / 4	24	24 / 0.59	1 : 4.0 / 0.64
OAI21 (c)	3 / 3	6	6 / 0.47	1 : 2.5 / 0.45
OAI221 (c)	5 / 5	120	120 / 1.07	1 : 7.0 / 1.22
OAI222 (c)	6 / 6	720	720 / 1.80	1 : 9.5 / 5.61
OAI22 (c)	4 / 4	24	24 / 0.62	1 : 5.0 / 0.66
OAI33 (c)	6 / 6	720	720 / 1.77	1 : 8.0 / 4.79
OR2 (c)	2 / 3	2	2 / 0.37	1 : 2.5 / 0.38
OR3 (c)	3 / 4	6	6 / 0.46	1 : 3.5 / 0.47
OR4 (c)	4 / 5	24	24 / 0.59	1 : 4.5 / 0.66
SDFFRS	6 / 30	720	288 / 2.99	48 : 6.4 / 13.01
SDFFR	5 / 25	120	96 / 1.61	16 : 6.4 / 3.07
SDFFS	5 / 25	120	36 / 1.49	16 : 6.0 / 2.77
SDFF	4 / 22	24	18 / 1.04	8 : 6.0 / 1.33
TBUF (c)	2 / 5	2	2 / 0.38	1 : 3.0 / 0.39
TINV (c)	2 / 4	2	2 / 0.39	1 : 3.0 / 0.38
TLAT	3 / 12	6	6 / 0.63	2 : 3.0 / 0.67
XNOR2 (c)	2 / 5	2	2 / 0.41	1 : 2.0 / 0.41
XOR2 (c)	2 / 5	2	2 / 0.44	1 : 2.0 / 0.40

of possible orders the approach can reduce the number of orders that have to be considered significantly.

Selecting Input Vector Orders

The technique to compute the functionally equivalent orders and a path back that uses the minimal amount of power, by constructing the power-sum DAG, was also evaluated on the open-source Nangate Open Cell Library [Nan08]. The results are shown in the last column of Table 4.4, giving the number of equivalence classes w.r.t. $\equiv_T$, the average number of maximal differences in wire chargings taken over all equivalence classes, and the amount of time until the power-sum DAG was constructed and the average computed from its leaves.

These results show that for combinational cells all orders lead to the same final state, which is expected as the state is completely determined by the new input values. For the sequential cells not all orders lead to the same final state, witnessed by multiple leaves in the power-sum DAG. This happens because the computation can depend on internally stored values, which might have different values when applying the input changes in different orders, cf. Section 4.2.

Below, the selection of orders shall be illustrated by means of an example. For this purpose, the scan logic of the cell `SDFFRS` is studied (which is also the same in the cells beginning with `SDFF`). This logic is implemented as a mux that selects, based on the value of the scan enable signal, between the data input and the scan input. In case the scan enable signal changes from 0 to 1 and the data input changes, then the corresponding back-pointer path in the power-sum DAG shows that it is better to first change the scan enable signal and then change the data input, than vice versa. This can be explained intuitively by the fact that while the scan enable signal is `0`, the mux is transparent to changes in the data input, so also wires connected to transistors controlled by the mux output are affected. This is not the case anymore if first the scan enable is changed to 1, so that the change in the data input cannot be observed at the output of the mux. In case of the cell `SDFFRS`, choosing the first order can cause 7 more wires to be charged than when using the second order.

Note that some correlation exists between the size of an equivalence class and the achievable power reduction: The more possible orders there are the more likely it is that another equivalent order with less power consumption exists. This can also be observed in results of Table 4.4, where the largest differences occur for combinational cells, which always have exactly one equivalence class.

4.4 Summary

This chapter presented techniques to efficiently analyze non-deterministic behavior, that is still present in the functional descriptions of cells. Such non-determinism can be an issue if it leads to different possible computation results. Both for VeriCell and for transistor netlists, the analysis runs in a very short amount of time and is not only able to prove presence or absence of different computation outcomes, but it also provides a counterexample in case order-dependent behavior exists. Then, additional constraints need to be imposed to still be able to use the cell in a larger environment. These constraints will take the form of timing checks, and will be presented in Section 5.1 of the next chapter.

Non-determinism that does not affect the functional behavior can be exploited to optimize other design goals, such as the measurement and the reduction of power consumption. For this purpose, the non-determinism analysis was extended to be power aware, by adding an abstract measure of the power consumed by a single wire charging. This allows to create a partitioning of all orders for which the power consumption has to be measured only for one element of each equivalence class. Furthermore, functionally equivalent orders can be re-ordered in such a way that the least amount of power is consumed.

A technique related to the presented reduction of executions of a hardware definition language (in this case, the VeriCell subset of Verilog) is [HMMCM06], which describes an application of dynamic partial-order reduction techniques to efficiently explore all possible execution runs of a test-suite for parallel SystemC processes. To this end, the code of parallel SystemC processes is analyzed and non-commutative transitions are detected. Subsequently, all possible permutations of non-commutative actions are considered in order to generate all schedules that may possibly lead to different final states. The technique reported in [HMMCM06] is comparable to the presented confluence-detection and -reduction techniques, however it is used in [HMMCM06] for the purpose of testing instead of exhaustive model-checking, as is the case here. The input language considered in [HMMCM06] is very rich and hence requires some manual code instrumentation to cater for the dynamic communication structure of parallel processes, which may be a restrictive factor in industrial cases. In [KGG08], the approach of [HMMCM06] is enhanced with slicing techniques and combined with static partial order reduction techniques. Neither of the approaches reported in [HMMCM06, KGG08] claim the minimality of the generated schedules. The non-determinism analysis in this thesis, however, guarantees that each two generated schedules do produce different output from some initial state. Thus, there is a formal justification for including both.

Order-independence analysis can be seen as computing the independence relation that is sufficient to perform partial order reduction [Pel98]; but it goes beyond ordinary independence relations by proving that, for the particular setting of UDPs and transistor netlists (and, in general, for vector-based transition systems that are deadlock-free and satisfy the fixed-point property), the presented criteria are also necessary for partial order reduction, i.e., violating them results in order dependent behavior leading to different states.

The analysis shares some basic ideas with the analysis of confluence in the setting of term rewrite systems, see for example [BN98, Ter03] for an introduction. Generally speaking, a system is confluent if any two computations can be joined again after an arbitrary number of steps. For transition systems, this relation has already been observed in [Kel75], where sufficient conditions for confluence of general transition systems are given. In contrast to the presented work, [Kel75] requires the transition system to be deterministic, whereas in Section 4.2 also non-determinism is allowed, i.e., a state may have multiple successor states that are labeled with the same input pattern. However, deadlock-freedom is required in Section 4.2 for every state and every input pattern, which is not globally the case in [Kel75]. Furthermore, one should note that the notion of order-independence is stronger than confluence; confluence only requires the existence of a state in which two computations can be re-joined, for order-independence it is required that the state reached after any of the two computations is the same.

Power consumed by cells is also determined in [HKC00]. The authors present an empirical algorithm, which also groups together different input vector transitions. However, they group together different values of inputs, whereas the approach in Section 4.3 groups together different orders of applying the same input vector.

Furthermore, the grouping in [HKC00] is made manually and afterwards all remaining input vectors and orders are enumerated explicitly, as opposed to the symbolic approach used in this thesis. Another approach that also uses a transition system model of circuits is presented in [SLL97]. This approach builds an explicit representation of the transition system, and hence has to combat the size of these transition systems by simplifying the netlist, something which is not required in a symbolic representation as is used here. A symbolic representation of cells for the purpose of power analysis is also used in [BBR96]. There, the symbolic representation is used during simulation of cells to determine the charge for each wire. The analysis of power-aware non-determinism can be seen as a preprocessing step, as the number of orders is reduced that later have to be simulated. Already in [RDJ96] it was observed that superfluous transitions (called *glitches*) of signals increase the power consumption. There however, glitches are detected by simulations, and are only considered at cell outputs. The authors of [RDJ96] propose a number of techniques to reduce glitches. One of these is the addition of delays to enforce a certain order of events, which is also what is proposed in Section 4.3 to select a low power evaluation.

Chapter 5

Relating Functional and Timing Behavior

Consistency of the different descriptions in a cell library is a prerequisite for obtaining correct chip designs. This does not only apply to the functional descriptions, whose equivalence was investigated in Chapter 3, but also to the consistency between functional and non-functional descriptions. For example, if the timing information is not consistent with the functional description, then it does not reflect the actual behavior of a cell, which could lead to a non-functional chip in the end.

In this chapter, two forms of timing information are considered. The first are *timing checks*. These checks describe certain assumptions about the environment of a cell. For example, a `$setup` timing check describes the amount of time a data signal is required to be stable before a clock edge may occur. The complementary timing check is the `$hold` timing check, that describes the amount of time a data signal is required to be stable after a clock edge. If a timing check is violated, an error is raised during simulation and often also an erroneous behavior is exhibited. In the rest of this chapter it is therefore assumed that the timing checks are never violated. Thereby, the timing checks constrain the set of possibly simultaneously changing inputs. Hence, timing checks are related to the order-independence considered in Sections 4.1 and 4.2 of the previous chapter, where all possible simultaneous input changes were considered. Section 5.1 therefore presents a way to incorporate the restrictions imposed by timing checks into the non-determinism analysis. This section is based on [RMR+09] and [RMZ10, RMZ11].

The second form of timing information that is considered in this chapter are *module paths*, also known as *timing arcs* or *delay arcs*. In the remainder of this chapter, the term *module path* is used, as is the case in the Verilog standard [IEE06, Clause 14]. Such a module path assigns a time delay to a change that propagates through a cell, i.e., a certain change in some input of the cell can cause a certain output of the cell to change its value after the given amount of time. Hence, such a module path does not only describe timing information, it also describes functional behavior, namely the change in the output to occur. Therefore, also this functional behavior should be consistent with the actual implementation. There are two possibilities for module paths to be inconsistent: Either, the module path describes a change propagation that can never occur, or a change can propagate through the cell for which no delay is assigned. The first case leads to a so-called *false path* and could, during timing closure, reject a correct design by imposing these incorrect constraints.

Table 5.1: Timing Checks based on a stability window

`$setup`	`$hold`	`$setuphold`
`$recovery`	`$removal`	`$recrem`

In the second case, a path is forgotten. Simulators would treat this situation by assigning no delay. This can cause further flaws in the subsequent design steps since a simulation of such a circuit does not reflect its actual behavior.

To analyze the module paths of a cell, two techniques are presented in Section 5.2, which is based on [RMS10]. The first considers all module paths specified, and verifies that they actually describe behavior that can be exhibited by the cell's functional description. Otherwise, a false path was found. The second technique enumerates all module paths of a cell. Thereby, one can check that all possible delay behaviors of that cell have been considered.

5.1 Timing Checks

Timing checks provide a means to describe constraints that the environment of a cell must satisfy in order to guarantee correct functioning of the cell. They are a part of the Verilog language, described in [IEE06, Clause 15], and can be divided into two classes. The first class contains timing checks that make use of a *stability window*, which describes a span of time around a given reference event in which a specified data event must not occur. An *event* is a specified transition of some signal in the cell. The other class of timing checks describes timing assumptions by providing limits on time differences between certain events. In this section, only timing checks using a stability window are considered, since those based on time differences are usually only specified for a single control signal, thus they only provide certain implementation constraints but they do not affect the functional description.

In total, there are 6 timing checks in the Verilog language that are based on a stability window. These are listed in Table 5.1. The two timing checks in the last column however, `$setuphold` and `$recrem`, are just a combination of the respective timing checks in the previous two columns. Thus, only the timing checks in the first two columns will be explained below. Such a timing check has the following general form:

$$\texttt{\$}\text{timing\_check}(\text{reference\_event}, \text{data\_event}, \text{timing\_check\_limit}[, [\text{notifier}]])$$

It should be remarked that due to historical reasons, the order of reference_event and data_event is reversed for the `$setup` timing check. The notifier is optional and specifies a signal that should be toggled in case a timing violation has been detected. The reference_event specifies an event that enables the timing check, i.e., only when this event is found, the check could report a timing violation. The data_event specifies another event that must not occur in a certain time interval around the time when the reference_event occurred. The particular time interval is specific to the concrete timing check and its size is given by the timing_check_limit.

For the `$setup` timing check, a timing violation is found whenever the data_event occurred less than timing_check_limit time units before the reference_event. Formally, if the reference_event occurs at time t, there must not be a data_event in the open interval $(t - \text{timing\_check\_limit}, t)$. For example, the timing check

`$setup(data, posedge clk, 10)` requires that the signal `data` is stable at least 10 time units before every positive edge of the signal `clk`, otherwise an error is raised.

The `$hold` timing check is the dual of the `$setup` timing check. It reports a timing violation in case the data_event occurs at most timing_check_limit time units after the reference_event. Formally, if the reference_event occurs at time t, the data_event must not occur in the half-open interval $[t, t + \text{timing\_check\_limit})$. An example of such a timing check is `$hold(posedge clk, data, 5)`, requiring the signal `data` to be stable every time the signal `clk` exhibits a positive edge, and at least 5 time units afterwards.

The timing checks `$removal` and `$recovery` behave like the `$setup` and `$hold` timing checks, respectively. Their purpose is purely syntactical; while `$setup` and `$hold` timing checks are generally used with a clock signal as reference_event and a data signal as data_event, the reference_event of the `$removal` and `$recovery` timing checks is usually a control signal like clear, reset, and set, while the data_event is usually a clock signal. Thus, for example the timing check `$recovery(negedge rst, posedge clk, 12)` is equivalent to the timing check `$hold(negedge rst, posedge clk, 12)`. For this reason, only `$setup` and `$hold` timing checks will be considered in the remainder; however, all statements apply to `$removal` and `$recovery` timing checks as well.

An event (either reference_event or data_event) has an optional event control, determining which transitions are considered as an event. By default, if no event control is specified, any change in value is considered an event. Possible restrictions are **posedge**, **negedge**, and restrictions of the form **edge**[v_1w_1, v_2w_2, ...]. The latter lists as arguments the transitions that should be matched, where $v_i \neq w_i \in \{\mathsf{X}, 0, 1\}$ (the value Z is also allowed, but is treated in the same way as X). Then, the restriction **posedge** is equivalent to **edge**`[01, 0x, x1]` and **negedge** is equivalent to **edge**`[10, x0, 1x]`. Mandatory for an event specification is the name of an input signal to the cell, for which the specified transitions cause this event to occur. Finally, an optional condition (indicated by `&&&`) can be given that further restricts when the event is considered to have occurred. Thus, the event **posedge** `clk` occurs for positive edges of the clock, whereas `d &&& cond` occurs for any transition of the signal `d`, provided the condition `cond` holds. Such a condition can be either a single signal (or its negation), or an equality or disequality expression. Conditions are divided into two classes, deterministic and non-deterministic conditions. A deterministic condition, which is of the form `s`, `~s`, `s === `v, or `s !== `v with `s` being a signal name and $v \in \{0, 1\}$, holds when the condition evaluates to 1, i.e., it does not hold if the signal `s` has value X. Non-deterministic conditions are of the form `s == `v or `s != `v, with `s` being a signal name and $v \in \{0, 1\}$ and they hold when the condition evaluates to 1 or X. Thereby, for example, both the events `d &&& ~rst` and `d &&& rst === 0` occur whenever the signal `d` changes and signal `rst` has value 0, whereas the event `d &&& rst == 0` occurs whenever signal `d` changes and signal `rst` has either value 0 or value X.

Constraints Imposed by Timing Checks on Order-Independence Analysis

As stated above, timing checks are added to assert a certain behavior of the system. Otherwise, if this behavior is not encountered, an error is triggered. Hence, timing checks can be regarded as describing illegal behavior.

When analyzing order-independence of a VeriCell description, described in Section 4.1, or of a transistor netlist, described in Section 4.2, one is only interested in whether two inputs might change simultaneously. Thus, neither the actual time limits nor the notifier variable are relevant for this purpose, as long as the time window is non-empty. Hence, for analyzing order-independence, formally only the restriction that the events of a `$hold` timing check may not occur simultaneously in any execution is used. In practice however, a `$hold` timing check is usually accompanied by a corresponding `$setup` timing check, thereby defining a stability window extending to both sides of the reference event.

Timing Checks and VeriCell Order-Independence

In a VeriCell description, all sequential behavior is encoded in the UDPs, thus only order-independence of UDPs has to be considered by the method presented in Section 4.1. However, timing checks are specified for inputs, thus information about the inputs of UDPs has to be inferred from these constraints.

For the combinational logic driving the inputs of a currently considered UDP, it is required that it does not contain loops and it is assumed that it computes its value instantaneously. Under these assumptions, functions in the external inputs and the outputs of other UDPs (which are assumed to be external inputs) can be created. These functions are then used as inputs of the UDP when checking the commuting diamond property using the technique presented in Section 4.1. Thereby, behavior that cannot occur due to functional dependencies of the UDP inputs is removed, and furthermore the counterexample states, in which two different orders of evaluation exist that lead to different final states, are expressed as formulas in these external inputs to the cell.

If such counterexample states are found, all those states are removed from them that violate one of the constraints imposed by the `$hold` timing checks. Then, if for a certain pair of input signals no counterexample states exist anymore, the UDP is order-independent in all of the allowed executions of the module. It should be noted that this order-independence does not solely depend on the UDP anymore, but also on the combinational logic and the timing checks present in the module that instantiates the UDP. Thus, the cell that is described by the module computes its state independent of the concrete order used to evaluate the UDP, provided the constraints of the timing checks are satisfied.

As an example, the UDP from Figure 4.1 and an enclosing module defining a cell `dff_enb` is given in Figure 5.1. This VeriCell description defines a D flip-flop that is enabled when the input `enb` is low, and that is disabled when the input `enb` is high. This is implemented by first negating, in line 21, the input `enb` to produce an internal signal `en` which is then used in line 22 as input for the active-high UDP implementation in `ff_en`.

It was already observed in Section 4.1 that the UDP `ff_en` has an order-dependency between inputs `d` and `ck`. Intuitively, this holds because on a positive edge of the clock either the clock changes first, thus the flip-flop still sees and stores the old value of the data input `d`, or the data changes before the positive edge of the clock, which makes the flip-flop store the new value of the data input `d`. However, this situation is usually considered to be illegal for a D flip-flop, hence a designer is likely to add, among others, the following timing checks:

```
$hold(posedge ck, negedge d, t1);
$hold(posedge ck, posedge d, t2);
```

```
primitive ff_en(q, d, ck, en);
  output q; reg q;
  input d, ck, en;

  table
    //  d  ck   en : q : q+
        0  (01) 1  : ? : 0;
        1  (01) 1  : ? : 1;
        ?  (10) ?  : ? : -;
        *   ?   ?  : ? : -;
        ?   ?   0  : ? : -;
        ?   ?   *  : ? : -;
  endtable
endprimitive


module dff_enb(q, d, ck, enb);
  output q;
  input d, ck, enb;

  not(en, enb);
  ff_en(q, d, ck, en);
endmodule
```

Figure 5.1: D Flip-Flop with Active-Low Enable

These timing checks rule out the behavior leading to the order-dependent counterexample that was described above, since the data input is not allowed to change simultaneously with the clock input anymore. When removing all counterexample states in which the clock input `ck` exhibits a positive edge simultaneously to a change of the data input `d`, no counterexamples remain. Therefore, the cell `dff_enb` has no order-dependency for these two inputs under the constraints defined above. Note that the two `$hold` timing checks could also be written as a single timing check `$hold(`**`posedge`**` ck, d, t)`, however this does not allow to specify different time limits, something that might be useful when determining the minimal time windows required by the implementation.

As explained above, the timing checks are incorporated into the order-dependence analysis for VeriCell by removing all those counterexamples in which a `$hold` timing check is violated. This way, illegal behavior of the environment is removed and only legal behavior that respects the timing checks is considered. Still, reachability of the counterexample states has to be determined, as was already done in Section 4.1, since it is not guaranteed that from the initial state a counterexample state is reachable and because outputs of other UDPs than the currently considered one are abstracted into new external inputs.

However, this reachability analysis can be improved, by also taking the timing checks into account. First, the non-deterministic next-state functions can be improved, since they can be reduced to only those cases where the changing inputs are order-independent under the added constraints. For example, in the cell shown in

Figure 5.1, together with the timing checks given above, the inputs `d` and `ck` are order-independent; the problematic cases must not be used as an input to the cell. This enlarges the equivalence classes of orders exhibiting the same behavior when restricting to legal behavior, thus decreasing the amount of non-determinism that has to be considered in the reachability check. Second, a trace reaching a counterexample state should be legal, i.e., it should respect all the timing checks. Hence, the LTL property to be checked for all counterexample states of an order-dependent pair $j \not\diamond_{udp}^{module} j'$ (which includes the restrictions imposed by the timing checks) becomes the following:

$$\mathsf{G}\,\neg\left(hold\_constraints \wedge \bigvee_{s \in j \not\diamond_{udp}^{module} j'} s\right)$$

In the above formula, the newly added state variable *hold_constraints* represents that no state of the currently considered trace has violated any of the timing checks. Thereby, the number of traces that have to be considered during reachability checking is reduced to the legal traces.

Timing Checks and Transistor Netlist Order-Independence

Also for transistor netlists, some order-dependencies are expected and should not be considered erroneous. For example, the dependency between the data input and the clock input of a transistor netlist implementing a flip-flop, which was discussed above for UDPs and was already found in the transistor netlist description considered in Example 4.2.29, should not be considered an error. Thus, also for transistor netlists timing checks should be considered.

As presented in Section 4.2, a vector-based transition system is created from a transistor netlist and order-independence is checked there. To incorporate the timing checks into this analysis, the conjunction of a BDD describing counterexample states and a BDD describing all legal input combinations is taken, to rule out those input combinations that are forbidden by the timing checks. This way, only those counterexamples remain where the initial state is one-step reachable and the counterexample is not contradicting the timing checks. Note that reachability of this state does not have to be checked, since the transistor netlist is allowed to start up in any arbitrary state. Furthermore, as a one-step reachable state is stable due to Lemma 4.2.26, a possibly remaining counterexample state is also one-step reachable when considering the timing checks; it is still possible to reach the state without changing the value of any input (thus, no event can occur and therefore no timing check can be violated).

Experiments

The restriction imposed by the timing checks, which describe illegal behavior, was added to the analysis of order-independence of VeriCell descriptions, presented in Section 4.1, and to the analysis of order-independence of transistor netlist descriptions, which was presented in Section 4.2, using the approaches presented above. This extended order-independence analysis was also evaluated on the Nangate Open Cell Library [Nan08], which contains 12 sequential cells whose Verilog descriptions are in the VeriCell subset. All of the experiments were conducted on a computer with an Intel Pentium 4 processor with 3.0 GHz and 1 GB memory running Linux.

Table 5.2: Reachability checking of order-independence of VeriCell descriptions in the Nangate Open Cell Library considering timing checks

Cell	**# Inp. Pairs**	**# Ord-Dep Prs**	**# Reach $\mathbb{B}$**	**Time [s] Thm. 4.1.8**	**Time [s] Lem. 4.1.4**
CLKGATE	1	0	–	–	–
CLKGATETST	2	0	–	–	–
DFF	1	0	–	–	–
DFFR	3	2	0	0.31	1.13
DFFS	3	2	0	0.33	0.67
DFFRS	6	5	1	6.87	24.20
DLH	1	0	–	–	–
DLL	1	0	–	–	–
SDFF	1	0	–	–	–
SDFFR	3	2	0	4.17	11.77
SDFFS	3	2	0	4.77	10.71
SDFFRS	6	5	1	49.72	2541.60

VeriCell

The encoding of cells into Boolean Transition Systems, which was presented in Section 4.1, was extended by the previously presented inclusion of timing checks. For every cell, an input file for the NuSMV model checker [CCG$^+$02] was created. Still, the model checker found a reachable order-dependent state for all of the cells. However, these counterexamples were due to the value X being allowed as an input of the cell, something that is not possible in a hardware implementation. Hence, the external inputs were restricted to be binary, i.e., to be either $\mathsf{0}$ or $\mathsf{1}$. The internal signals were not restricted and were allowed to take any ternary value from $\mathbb{T} = \{\mathsf{0}, \mathsf{1}, \mathsf{X}\}$. The results for these experiments are shown in Table 5.2, where the first column shows the name of the cell, the second column gives the number of UDP input pairs, and the third column shows the number of pairs that were found to be order-dependent. It can be observed from the table that only for 6 cells states exist that can cause an order-dependency. For 4 of these cells none of the counterexample states can be reached, hence the UDPs used in these cells with binary inputs are order-independent. This can be seen in the fourth column of Table 5.2, which shows the number of reachable counterexample states. Finally, the fifth and sixth columns of the table present the time it took NuSMV to check reachability, where for the times shown in the fifth column the approach of Theorem 4.1.8 was used, whereas in the other case the naive approach of Lemma 4.1.4 was used.

For the last 2 cells, which are the cells `DFFRS` and `SDFFRS` implementing a flip-flop (with scan logic) that can be set and reset, a counterexample state can still be reached. The inputs that cause this behavior are in both cases the set and reset inputs. When switching both from active to inactive, the order of this deactivation determines the output of the cell. When deactivating the set signal first, then the reset is still active, forcing the output to be $\mathsf{0}$. Otherwise, when first deactivating the reset signal, the activated set signal will set the output to be $\mathsf{1}$. Looking at the Verilog implementation, it seems that for this combination of inputs a `$hold` check was forgotten, since a `$setup` check has been specified. This demonstrates that

Table 5.3: Order-Independence of Transistor Netlists in the Nangate Open Cell Library considering timing checks

Cell	**# State Vars**	**# Inp. Pairs**	**# Ord-Dep Prs**	**Time $\overset{\downarrow 1}{\diamond}$ [s]**	**Time all orders [s]**
CLKGATE	2	1	0	0.01	0.04
CLKGATETST	2	3	0	0.01	0.12
DFF	4	1	0	0.01	0.11
DFFR	4	3	0	0.04	0.31
DFFS	4	3	0	0.02	0.24
DFFRS	4	6	1	0.07	1.23
DLH	2	1	0	0.02	0.04
DLL	2	1	0	0.01	0.04
SDFF	4	6	0	0.03	1.05
SDFFR	4	10	0	0.07	7.38
SDFFS	4	10	0	0.07	7.51
SDFFRS	4	15	1	0.17	197.54

formal verification of these timing checks is needed and that the presented method is able to indicate what timing checks might be missing.

Moreover, a proprietary cell library provided by a customer to Fenix Design Automation was verified. This cell library was already suspected of containing an issue related to non-determinism. Indeed, the order-independence analysis found a reachable state from which two possible executions exist that lead to different behavior. This counterexample is rather complex in nature and cannot be traced back to or even be solved by adding timing checks. This shows that although timing checks and order-dependence are related, the timing checks are not powerful enough to rule out all possible order-dependent behavior.

The order-independence analysis based on the commuting diamond property and the naive approach of Lemma 4.1.4, which enumerates all possible orders, were extended by the presented inclusion of timing checks and were compared for the 12 sequential cells in the Nangate Open Cell Library [Nan08]. Again, the times it took NuSMV to model check reachability of possible counterexample states was measured for both versions. The experiments showed that the approach based on the diamond property was consistently faster. Particularly for the largest cell `SDFFRS` the model checking time could be reduced from more than 40 minutes to less than 50 seconds. For this cell, also NuSMV's memory consumption was measured, which decreased from more than 880 MB to ca. 110 MB.

Transistor Netlists

The order-independence analysis of Section 4.2, extended with the presented consideration of timing checks, was also applied to the transistor netlists of the 12 sequential cells in the Nangate Open Cell Library [Nan08], whose corresponding functional descriptions were used in the above experiments for VeriCell descriptions. For each of the cells the timing checks given in the corresponding Verilog module were used. The results are shown in Table 5.3, where the first column shows the name of the cell. In the second and third column, the number of state variables and the number of input pairs are presented. Finally, the fourth and fifth column give the time it took to check order-independence of the transistor netlist, using the approach based on the one-step reachable commuting diamond property and the naive approach enumerating all orders, respectively. It can be observed that analyzing order-independence was still possible in less than 0.25 seconds for every cell in the library when using the diamond property, whereas the naive approach still took more time, especially for the larger cells.

With the timing checks ruling out illegal behavior, ten cells were proven to be order-independent, when considering binary inputs. For two cells however, namely the cells `DFFRS` and `SDFFRS`, a counterexample was found. This counterexample is the same as the one found for the Verilog implementation above: there is no timing check specified for the deactivation of the set and reset inputs, hence when deactivating both at almost the same time the output value depends on whether the set is deactivated first, leaving the reset still active, or whether the reset is deactivated first, leaving the set still active. This problem might therefore really cause non-deterministic behavior, which is undesired. The non-determinism can be resolved by adding a timing check that disallows simultaneous disabling of both the set and reset inputs. Then, the presented technique does not report any further order-dependencies.

5.2 Module Paths

This section analyzes the *module paths* of cells and presents techniques to guarantee that they are consistent with the cell's functional behavior. A module path assigns a delay for a change of an input to propagate to an output, as defined in the Verilog standard [IEE06, Clause 14]. The actual time values put on module paths are irrelevant for the analyses presented in this section. Only necessity and realizability of module paths is considered, i.e., the need and use of their presence or absence. When considering all of the specified paths of a cell, one might get *false paths* during the timing closure of a larger design composed of multiple cells, which might reject correct designs. Therefore, the given module paths are checked for being feasible within the functional description. Furthermore, a technique is presented to enumerate all possible module paths of a cell. This forms the basis for any subsequent timing analysis, i.e., one has to first determine whether a module path can actually occur or not before the actual delays specified by module paths can be determined.

Since module paths are given in the Verilog description of a cell, this section will focus on functional descriptions in the VeriCell subset which was introduced in Section 2.3. However, since the functional description is kept rather abstract, the techniques could also be extended to other functional descriptions. As equivalence of the different functional descriptions is usually checked anyway, for example by the method presented in Chapter 3, considering only VeriCell descriptions is not a restriction.

Preliminaries

Consider the cell depicted in Figure 5.2, which shows the functional description of a D flip-flop with active low reset in the VeriCell subset of Verilog and the module paths of its timing specification. This cell is constructed from two instances of the built-in primitives **`buf`** and **`not`**. Furthermore, it contains two instances of the User Defined Primitive (UDP) `latch`, whose definition is given at the top of Figure 5.2. This UDP implements a storage latch that is transparent when its `ck` input is 1 and

```
primitive latch(q, d, ck, rb);
  output q; reg q; input ck, d, rb;
  table
    // d  ck  rb  : q : q'
       *   0   ?  : ? : - ;
       ?   0   1  : ? : - ;
       0   1   ?  : ? : 0 ;
       1   1   1  : ? : 1 ;
       ?   ?   0  : ? : 0 ;
  endtable
endprimitive

module dffr(q, ck, d, rb);
  output q; input ck, d, rb;
  buf(q, qint);
  latch(qint, iq, ck, rb);
  latch(iq, d, ckb, rb);
  not(ckb, ck);

  specify
    (negedge rb => (q +: 0)) = t_rst;
    if (rb==1) (posedge ck => (q +: d)) = t_ck;
  endspecify
endmodule
```

Figure 5.2: Verilog Source of a Resettable D Flip-Flop

that stores its current value if the `ck` input is 0. Additionally, it can be reset by setting input `rb` to 0.

In the timing specification of the example cell `dffr`, given between the keywords **`specify`** and **`endspecify`**, there are two module paths which are both *edge-sensitive* paths, i.e., they are active whenever a certain change in an input is exhibited. The first module path describes that when the reset has a falling edge (i.e., an edge towards 0), then the output `q` changes its value after `t_rst` time units. The *data source expression* `+: 0` describes that the output will change its value to 0. Similarly, the second module path defines that on a positive edge of the clock, the output `q` will change its value to the value of `d`. However, this module path is a *state-dependent* module path, since it only applies when the condition `rb == 1` is true, as specified in the **`if`** preceding the module path.

Such a cell with k inputs $i_1, \ldots, i_k$, m outputs $q_1, \ldots, q_m$, and n sequential variables $s_1, \ldots, s_n$ is interpreted as a set of Boolean equations, using the semantics presented in Chapter 3. All of these variables can take values from the ternary values $\mathbb{T} = \{0, 1, \mathsf{X}\}$, where 0 and 1 correspond to the Boolean values false and true, respectively, whereas X can be understood as representing an unknown value. Let I denote the space of all possible inputs $I = \mathbb{T}^k$, O denote all possible outputs $O = \mathbb{T}^m$, and S denote all possible states $S = \mathbb{T}^n \times \mathbb{T}^k$, in which also the previous values of the inputs are stored, denoted $i_1^p, \ldots, i_k^p$, in order to detect transitions. For $z \in \mathbb{N}$ and a vector $\vec{v} = (v_1, \ldots, v_z) \in \mathbb{T}^z$, let $\rho_j(\vec{v}) = v_j$ again denote the projection to the j-th component of that vector, for all $1 \leq j \leq z$. This definition is lifted to sets of vectors by defining $\rho_j(V) = \bigcup_{\vec{v} \in V}\{\rho_j(\vec{v})\}$ for $V \subseteq \mathbb{T}^z$ and $1 \leq j \leq z$. Instead of $(\vec{s}, \vec{i}) \in S$ the notation $\vec{s};\vec{i} \in S$ will be used.

Because VeriCell programs might be non-deterministic, as already observed in Chapter 3, the computation of stable next states is represented by the function $\delta : I \times S \rightarrow \mathcal{P}(S)$, with $\mathcal{P}(S)$ being the power set of states, computing for a given input vector $\vec{i} \in I$ and state $\vec{s};\vec{i^p} \in S$ the set of all possible next states $\delta(\vec{i}, \vec{s};\vec{i^p})$. Given a state $\vec{s};\vec{i^p} \in S$, the function $\lambda : S \rightarrow O$ computes the values $\lambda(\vec{s};\vec{i^p})$ of the outputs in that state. Extending δ and λ to sets of states is done in the natural way: $\delta(\vec{i}, S') = \bigcup_{s' \in S'} \delta(\vec{i}, s')$ and $\lambda(S') = \bigcup_{s' \in S'} \lambda(s')$ for every $S' \subseteq S$. As in Chapter 3 and as stated in the Verilog standard [IEE06], the initial state s_0 of a cell is the state in which all variables have the value X.

A state $\vec{s};\vec{i} \in S$ is called *reachable* if there exist states $\vec{s}_1;\vec{i}_1, \ldots, \vec{s}_r;\vec{i}_r \in S$ and inputs $\vec{i}_0, \ldots, \vec{i}_{r-1} \in I$ such that $\vec{s}_r;\vec{i}_r = \vec{s};\vec{i}$ and for all $0 \leq j < r$ it holds that $\delta(\vec{i}_j, \vec{s}_j;\vec{i}_j) \ni \vec{s}_{j+1};\vec{i}_{j+1}$. The *constrained evaluation* of δ for an input vector $\vec{i} \in I$, state $\vec{s};\vec{i^p} \in S$, and constraint values $\vec{c} \in \mathbb{T}^k$ is defined as $\delta(\vec{i} \wedge \vec{c}, \vec{s};\vec{i^p}) = \delta(\vec{i}, \vec{s};\vec{i^p})$ if for all $1 \leq j \leq k$ either $i_j = c_j$ or $c_j = \mathsf{X}$. Otherwise, $\delta(\vec{i} \wedge \vec{c}, \vec{s};\vec{i^p}) = \emptyset$.

In the following, an *edge specification* edge $\in \{$**posedge**, **negedge**, `""`$\}$ is interpreted as the set of transitions that it matches. Here, as required in the Verilog standard [IEE06, Clause 14], also transitions to and from X are matched. Thus, **posedge** $= \{(0,1), (0,\mathsf{X}), (\mathsf{X},1)\}$, **negedge** $= \{(1,0), (\mathsf{X},0), (1,\mathsf{X})\}$, and `""` (the empty string) $= \star = \{(u, v) \mid u \neq v\} =$ **posedge** $\cup$ **negedge**.

Checking Module Paths

A *module path* describes a delay between an input and an output of the cell. There are two basic types of module paths: *simple paths* and *edge-sensitive paths*. An example of a simple path is `(ck => q) = 10`, expressing that a change of input `ck` influences output `q` after a delay of 10 time units. An example of an edge-sensitive path is `(`**`posedge`**` ck => (q +: d)) = 12`, expressing that a positive edge of input `ck` affects the output `q`, which takes the value `d` ("+" indicates that the value of `d` is passed in non-inverted form), after a delay of 12 time units.

Basic paths can be used to construct the *state-dependent module paths*, which are module paths equipped with a condition. An example of a state-dependent module path is **`if`**` (rb == 1) (`**`posedge`**` ck => (q +: d)) = 12`, which specifies the same delay as the previous edge-sensitive path example, but this delay only occurs if the condition `rb == 1` holds. A condition is defined to hold if it does not evaluate to 0, i.e., if it evaluates to either 1 or X.

Only state-dependent module paths will be considered in the following, since the others can be expressed as such paths by simply adding "**`if`**` (`i`==X)`" for some input i, as a comparison with the value X will always make the equality evaluate to X.

Requirements for Simple Module Paths

A simple module path, as its name indicates, is the simplest form of specifying that an input influences the value of an output. However, it is desirable that such an effect does actually take place. For example, one could add the simple module path `(d => q) = 1;` to the example in Figure 5.2, stating that a change of input `d` affects the value of output `q` one time unit after the change of input `d`. However, the output of a flip-flop will never change as a result of only changing the data input; for the output to change a positive edge of the clock is required. Hence this is a module path that never occurs in practice, which might result in too severe restrictions in the timing analysis such that a circuit using this flip-flop with the above module path could fail to meet its timing requirements, although an implementation would never suffer from this problem.

A formal definition of the requirements for a module path to be *consistent* with the functional description shall be given next. For this purpose, a state-dependent simple module path of the form `if ( (`i_1`==`c_1`) &&...&& (`i_k`==`c_k`) ) (`i_j p`=>` q_l`)` is considered, where p is called the *polarity* of the module path, with $p \in \text{Pol} = \{\texttt{+}, \texttt{-}, \texttt{""}\}$ (where `""` represents the empty string). The polarity expresses the direction of the output change: For positive polarity ($p = +$), if the output changes, then the change is into the same direction as the input. For negative polarity ($p = -$) the output changes into the opposite direction of the input, if it does change. For no polarity ($p = \texttt{""}$) the output is free to change into any direction. Note that any (state-dependent) simple module path can be written in the above format, by inserting X for c_j when input i_j is not constrained in the original module path.

The semantics of module paths imposes two constraints on the transition system of cells. First, a state is required to be reachable in which the specified output will change as a result of only changing the specified input. If this were not the case, then the output would never change as a result of the changing input, hence this module path would never be active and could be removed. The second constraint deals with the polarity: If it is either + or −, then in case the output changes, it is required to change into the direction specified by the polarity. Formally, these two constraints are expressed as follows:

1. There exists a reachable state $\vec{s};\vec{i^p} \in S$, a value $v \in \mathbb{T}$, and an output value $\lambda' \in \rho_l(\lambda(\delta(\vec{i^p}[j := v] \wedge \vec{c}, \vec{s};\vec{i^p})))$ such that $\lambda' \neq \rho_l(\lambda(\vec{s};\vec{i^p}))$.

2. If $p \in \{+, -\}$, then for all reachable states $\vec{s};\vec{i^p} \in S$, all values $v \in \mathbb{T}$, $\lambda = \rho_l(\lambda(\vec{s};\vec{i^p}))$, and every $\lambda' \in \rho_l(\lambda(\delta(\vec{i^p}[j := v] \wedge \vec{c}, \vec{s};\vec{i^p})))$ the following holds:

   - If $p = +$ then
     - If $(i^p_j, v) \in$ **posedge**, then $\lambda = \lambda'$ or $(\lambda, \lambda') \in$ **posedge**
     - If $(i^p_j, v) \in$ **negedge**, then $\lambda = \lambda'$ or $(\lambda, \lambda') \in$ **negedge**
   - If $p = -$ then
     - If $(i^p_j, v) \in$ **posedge**, then $\lambda = \lambda'$ or $(\lambda, \lambda') \in$ **negedge**
     - If $(i^p_j, v) \in$ **negedge**, then $\lambda = \lambda'$ or $(\lambda, \lambda') \in$ **posedge**

Requirements for Edge-Sensitive Module Paths

An edge-sensitive module path specifies that a certain change of the input influences the output. Hence, it is again required that this effect on the output does exist when the input exhibits one of the specified transitions, like for simple module paths. Furthermore, an edge-sensitive module path specifies the value of the output after the change by means of the data source expression. It should also be verified that this expression reflects the actual computation of the output value in the functional description.

Again, two constraints have to be satisfied for a state-dependent edge-sensitive module path, which can always be written in the form

$\texttt{\textbf{if}}$ `((`i_1`==`c_1`) &&...&& (`i_k`==`c_k`)) (edge `i_j` => (`q_l` `p`: `d`)),`

with polarity $p \in \mathrm{Pol} = \{+,- , \texttt{""}\}$ and edge $\in \{\texttt{\textbf{posedge}}, \texttt{\textbf{negedge}}, \texttt{""}\}$. The first constraint is the same as constraint (1) for simple module paths, except that now the input change must be one of the transitions contained in the specified edge. The second constraint deals with the polarity and the data source expression specified in an edge-sensitive module path. It is required that for positive or negative polarity p, the output has the value of the data source expression or its negation, respectively, whenever the input makes one of the specified transitions. Hence, the module path must be realizable in some reachable state and its data source expression must correspond to the output value in all reachable states triggering the module path.

Formally, constraint (1) is required to hold for one of the given input transitions in edge and second, if $p \in \{+, -\}$, then for all reachable states $\vec{s};\vec{i^p}$ and values $v \in \mathbb{T}$ with edge $\ni (i^p_j\ v)$, it is required that $\rho_l(\lambda(\delta(\vec{i^p}[j := v] \wedge \vec{c}, \vec{s};\vec{i^p}))) = \{q'_l\}$, where $q'_l = d$ if $p = +$ and $q'_l = \neg d$ if $p = -$.

Reachability Checking of Module Paths

With the formal requirements of module paths as given above, realizability and correctness can be checked in the functional implementation of the cell. For this purpose, the VeriCell semantics of Chapter 3 is used to obtain a symbolic representation of the cell's functional description. A fixed-point construction is applied to the presented semantics to create a function δ that computes, given a current stable state and some new input vector, a next stable state. By definition, all variables in Verilog initially have the value X. Starting from this state, a reachable state is searched that satisfies condition (1). This search uses an implementation of a symbolic simulator for symbolic transition systems described in terms of Boolean equations. If no such state exists, then the input of the module path never affects the output, hence this module path is infeasible and is reported to the user. Otherwise, another reachability check is performed. This time, it checks whether from the initial state another state can be reached in which a counterexample to the specified behavior occurs. For simple module paths with specified polarity, such a counterexample is a state where the output changes in the opposite direction of the polarity. In case of an edge-sensitive module path with specified polarity, a reachable state is being searched for in which one of the specified input transitions has been applied, but the output does not have the value given by the data source expression. If such a counterexample state is reachable, then the module path is flagged as incorrect. Otherwise, the reachability check established that for all reachable states the data source expression represents the value of the output, hence the module path correctly reflects the functional behavior of the cell.

However, with the above notion of inconsistent module paths, usually only a very small number of module paths are correct. This is usually due to the value X. Since digital circuits only operate on two logical values (the Boolean values false and true), this value is commonly interpreted as unknown in the two-valued logic. Then, for example, a transition from 0 to X could either be a transition from 0 to 1 or no transition at all. But the definition of the value X in the Verilog standard [IEE06] defines it to be a third value unrelated to both 0 and 1 (see [Tur03] for an in-depth discussion of the problems associated with the value X). In order to cope with practical examples, and because in actual implementations the values will always be either true or false, the module paths may be *strengthened*, which interprets the value X as the set $\{0, 1\}$ and then filters out all transitions that do not express a change in values. As an example, the edge specification **posedge** $= \{(0, 1), (0, X), (X, 1)\}$ shall be strengthened. Replacing all occurrences of X by both 0 and 1 gives the set $\{(0, 1), (0, 0), (1, 1)\}$. When filtering out the non-changing transition $(0, 0)$ and $(1, 1)$, only the binary transition $(0, 1)$ remains.

Furthermore, also conditions can be strengthened: Previously, and as required by the Verilog standard [IEE06], a condition was active if it evaluates to either 1 or to X. In the strengthened condition, it is required that both sides of an equation are equal. For comparisons with Boolean values, this again amounts to viewing X as either 0 or 1.

These two strengthenings can be enabled by two options passed to the reachability checking procedure. Then, the requirements as described above are checked for the strengthened module paths. If they are consistent with the functional description, then they faithfully specify the behavior of the functional description, under the condition that the strengthenings hold.

In case of the example cell `dffr` shown in Figure 5.2, both non-strengthened module paths are inconsistent with the functional description. The first module path `(`**`negedge`**` rb => (q +: 0))` specifies that the output `q` changes to the value 0 if the input `rb` exhibits one of the transitions that are considered a **negedge**. Looking at the table of the UDP `latch` in Figure 5.2 one can observe that the output is set to 0 when input `rb` is 0. However, as defined in the preliminaries of this section, also the transition $(1, X)$ is contained in **negedge**, hence the value of input `rb` after this transition is X. For this transition, the value of output `q` will be set to X, since none of the rows of UDP `latch` match in this case. Similarly for the second module path **`if`**` (rb==1) (`**`posedge`**` ck => (q +: d)):` The data value is only latched if the clock has value 1, but the **posedge** also matches if it changes from 0 to X. Furthermore, for this module path the condition `rb == 1` is true if input `rb` is either 1 or X, since $X == v$ is true for all values v. If however the input `rb` has the value X, then the data input will not be latched since it is unclear whether the reset is active or not. Hence, in this case the data source expression does not always correspond to the value of the output `q`, even when restricting to binary values for input `ck`. Strengthening the condition, which requires that both of its sides have equal values, activates this module only when the reset signal `rb` has value 1. With these two strengthenings, the data source expression does express the value of the output `q`, as intended.

Deriving Module Paths

When given a cell library, all possible module paths have to be specified, otherwise the simulation will use no delay and therefore might not faithfully model the real implementation. Hence, a method is desirable to automatically extract all possible module paths from a functional description.

To determine module paths for a given cell, it is first described as a set of Boolean equations, using the VeriCell semantics introduced in Chapter 3. This gives for every signal in the cell an equation computing its next output value from the current values of the inputs to the primitive driving the wire. Such a primitive might either be a built-in primitive or a combinational UDP, which then results in a combinational equation, or a sequential UDP, for which also the current output value is considered as an input, i.e., this is modeled as a feedback loop. Then, again a fixed-point is constructed to obtain equations that describe the stable next values.

Using these equations, each triple of **posedge** or **negedge**, input, and output of the given cell is considered. Given such a triple, a formula is constructed describing those states for which the specified input makes a transition matched by the edge specification and for which the output changes its value. Furthermore, the formula requires the remaining inputs to remain unchanged, since the influence of the single input on the output is to be determined.

Formally, for a triple (edge, j, l), reachability of a state $\vec{s};\vec{i^p}$ is checked, for which $v \in \mathbb{T}$, $(i^p_j, v) \in \mathsf{edge}$, and $\lambda' \in \rho_l(\lambda(\delta(\vec{i^p}[j := v], \vec{s};\vec{i^p})))$ with $\lambda' \neq \rho_l(\lambda(\vec{s};\vec{i^p}))$. This reachability check is performed using a symbolic simulator. If no such state is reachable, then the considered input transitions never have an effect on the currently considered output, and hence there should not be a module path. Otherwise, if such a reachable state can be found, then the considered transition of the input can change the output value and therefore a module path should exist for this configuration. In that case a symbolic simulation of the specified transitions in the edge specification is performed for the currently considered output. This approximates the data source expression for the module path. It is only an approximation, since only two input vectors are simulated: One where the currently considered input has the previous value of its transition and one where the currently considered input has the new value after the transition. Such a transition might occur at any time, therefore the state in which the symbolic simulation starts is allowed to be arbitrary. This however includes unreachable states, i.e., states which are never reached during the operation of the cell. To simplify the data source expression, unreachable states are removed. For this purpose, the found data source expression is converted into its sum-of-products representation. Then, reachability is checked for every product (representing a state). If one of these states is unreachable, then it can never contribute to the value of the output, and hence it can be removed. The final data source expression is therefore constructed from the reachable products and the found module path is output to the user.

Experimental Results

The presented methods for checking feasibility of module paths and for deriving module paths from functional descriptions were applied to cells taken from the Nangate Open Cell Library [Nan08] and to cells from proprietary cell libraries provided by the industrial partner Fenix Design Automation. In the remainder of this section, the main focus is on the results obtained for the Nangate Open Cell Library, due to its public availability, and only brief comments are made on the observations for proprietary cells.

Checking Module Paths

The results of checking the module paths contained in the Nangate Open Cell Library are shown in Table 5.4. Columns “Cell” and “# MPs” give the name (marked by “(c)” for combinational cells) and the number of module paths in that cell, respectively; “Direct” specifies how many module paths hold directly without any strengthening, “Edge” is the number of consistent module paths when strengthening the edge, “Cond” is the number of consistent module paths when strengthening the condition, and “Cond & Edge” is the number of consistent module paths when strengthening both

Table 5.4: Checking module paths in the Nangate Open Cell Library

Cell	# MPs	Direct	Edge	Cond	Cond & Edge	Time [s]
AND2 (c)	2	2	2	2	2	0.11
AND3 (c)	3	3	3	3	3	0.12
AND4 (c)	4	4	4	4	4	0.12
AOI21 (c)	5	5	5	5	5	0.13
AOI211 (c)	8	8	8	8	8	0.13
AOI22 (c)	12	12	12	12	12	0.14
AOI221 (c)	15	15	15	15	15	0.16
AOI222 (c)	21	21	21	21	21	0.25
BUF (c)	1	1	1	1	1	0.08
CLKBUF (c)	1	1	1	1	1	0.10
CLKGATE	1	1	1	1	1	0.17
CLKGATETST	1	1	1	1	1	0.19
DFF	2	0	2	0	2	0.20
DFFR	6	0	4	0	6	0.24
DFFS	6	0	4	0	6	0.24
DFFRS	14	4	6	4	14	0.41
DLH	2	1	2	1	2	0.20
DLL	2	1	2	1	2	0.20
FA (c)	18	18	18	18	18	0.14
HA (c)	6	6	6	6	6	0.12
INV (c)	1	1	1	1	1	0.08
MUX2 (c)	6	6	6	6	6	0.13
NAND2 (c)	2	2	2	2	2	0.12
NAND3 (c)	3	3	3	3	3	0.12
NAND4 (c)	4	4	4	4	4	0.12
NOR2 (c)	2	2	2	2	2	0.11
NOR3 (c)	3	3	3	3	3	0.12
NOR4 (c)	4	4	4	4	4	0.13
OAI21 (c)	5	5	5	5	5	0.12
OAI211 (c)	8	8	8	8	8	0.13
OAI22 (c)	12	12	12	12	12	0.14
OAI221 (c)	15	15	15	15	15	0.16
OAI222 (c)	22	22	22	22	22	0.26
OAI33 (c)	19	19	19	19	19	0.26
OR2 (c)	2	2	2	2	2	0.11
OR3 (c)	3	3	3	3	3	0.12
OR4 (c)	4	4	4	4	4	0.13
SDFF	2	0	0	0	2	0.21
SDFFR	6	0	4	0	6	0.35
SDFFS	6	0	4	0	6	0.32
SDFFRS	14	4	6	4	14	0.87
XNOR2 (c)	4	4	4	4	4	0.12
XOR2 (c)	4	4	4	4	4	0.12

the condition and the edge. Note that after strengthening either the edge or the condition, module paths that are readily consistent remain consistent. Also, module paths that are consistent after strengthening either the edge or the condition remain consistent after strengthening both. The final column "Time [s]" gives the time in seconds required for checking all module paths of that cell.

The strengthenings were tried in the order of the columns, i.e., if a module path was found to be inconsistent with the functional description, first the edge specification was replaced by an edge that does not contain any X values, and if this module path was still inconsistent the condition was strengthened, first with the general edge specification, then with the strengthened edge.

As it can be seen in the results, none of the module paths in the library were found to be inconsistent with the functional description of the cell when both strengthenings are used, since the numbers in the column "Cond & Edge" are the same as the total number of module paths in column "# MPs". For the combinational cells, marked with "(c)" in the table, all module paths were directly realizable. In case of the sequential cells, it was never sufficient to only strengthen the condition. Module paths that still were inconsistent after strengthening the edge required strengthening of the condition and of the edge. This can be seen in the results, as the numbers in the column "Direct" are always equal to the numbers in the column "Cond". It should be stressed again that these strengthenings reflect a misconception between the semantics implied by the Verilog standard and the common perception of Verilog designers.

A final observation is that the time it took to check all module paths in a cell is negligible, as shown in the last column of Table 5.4. Hence, this check can easily be done before doing timing analysis of larger circuits using such a cell library.

For the cells from the proprietary cell libraries, it was also observed that the time taken to check all module paths was usually small. Only for 2 cells, each having 224 module paths, checking the module paths took up to 19 seconds. However, for the proprietary cells quite a number of module paths were inconsistent with the functional description, even after strengthening. For example, in a cell containing a scanable flip-flop, the scan logic was forgotten in the data source expression of a module path, i.e., the scan-enable input was assumed to be always 0. Another reason for module path inconsistency was the assumption that all inputs to a cell are binary, i.e., either 0 or 1. An example of this is the module path `(`**`posedge`**` CK => (Q +: !SE&D | SE&SI))`, which was specified for another scanable flip-flop with clock input `CK`, data input `D`, scan-enable `SE` and scan-input `SI`. The idea is that when `SE` is 0 then the value of `D` will become visible at the output `Q`, whereas the output value will be the value of `SI` if the input `SE` is 1. However, in this case a non-pessimistic multiplexer was used to select between `D` and `SI`. This multiplexer also outputs 1 when both `D` and `SI` are 1, even when `SE` is X. Then however, the data source expression does not describe the behavior of the circuit, since it evaluates to X whereas the flip-flop outputs a 1. Hence, such a module path is only consistent with the functional description if one strengthens even more to allow only binary input values, which can also be done in the implementation. Another possibility to make the module path consistent is the description of this non-pessimistic behavior of the multiplexer, by using the data source expression `!SE&D | SE&SI | D&SI`.

Deriving Module Paths

Also the method for the derivation of module paths was applied to the Nangate Open Cell Library and to proprietary cell libraries provided by the industrial partner Fenix Design Automation. Here, the inputs were always restricted to be binary, i.e., either 0 or 1. For the combinational cells, it found the expected dependency of all inputs on the outputs. More interesting are the results for the sequential cells, for which, as a representative, the output for the cell `SDFFRS` from the Nangate Open Cell Library shall be given, describing a scanable flip-flop with active-low reset `RN` and active-low set `SN` (where the data source expressions were modified slightly to be more readable):

```
(posedge CK => (Q +: RN & (!SN | !SE&D | SE&SI) ));
(posedge CK => (QN +: SN & (!RN | !SE&!D | SE&!SI) ));
(negedge RN => (Q +: 0));
(posedge RN => (Q +: !SN));
(negedge RN => (QN +: SN));
(negedge SN => (Q +: RN));
(negedge SN => (QN +: 0));
(posedge SN => (QN +: !RN));
```

The above output gives all module paths that were found in the cell. In that output, the module paths on a positive edge of the clock `CK` for the output `Q` and the inverted output `QN` are contained, as expected. Also the module paths for negative edges of inputs `RN` and `SN` on the two outputs are expected, which describe the reset and set functionality, respectively. However, what at first seems surprising are the two module paths (**`posedge`** `RN => (Q +: !SN))` and (**`posedge`** `SN => (QN +: !RN))`, which correspond to the deactivation of the reset and set signals, respectively. When looking at the functional description of this cell, one sees that it sets both outputs `Q` and `QN` to 0 if both reset and set are active (i.e., when both `RN` and `SN` are 0). When either set or reset is deactivated, then the respective other signal is still active, and for just one active signal the outputs `Q` and `QN` are the inverses of each other. Hence, these module paths do exist in the cell, something that could easily be overlooked.

5.3 Summary

This chapter presented techniques to check that also the timing specification of a cell, given as timing checks and module paths, are consistent with the functional behavior of the cell.

Section 5.1 considered timing checks, which enforce certain restrictions on the environment of a cell. It was shown that they have a relation with the order-independence analysis presented in Chapter 4, since they rule out input changes occurring simultaneously. Thus, both the order-independence analysis of Verilog simulation descriptions and that of transistor netlists were extended to only investigate situations where the timing checks are respected. These extended order-independence analyses were applied on industrial cell libraries and were able to prove order-independence for most of them. Therefore, after analyzing order-independence, the equivalence check presented in Chapter 3 only needs to encode the simulator order, since it is ensured that non-deterministic behavior does not lead to different computation results when respecting the constraints imposed by the timing checks.

Furthermore, the experiments showed that an order-dependency can often be removed by a timing check, i.e., an order-dependency that remains after restricting to the allowed behavior is a sign that the environment of a cell is not restricted sufficiently. However, it was also observed that order-dependencies exist that cannot be solved by adding any number of timing checks.

Section 5.2 described a method for checking module paths against the functional description of cells contained in a cell library. This method is useful to achieve timing closure by identifying infeasible module paths. Furthermore, it showed a technique to extract module paths from a functional description of a cell. This method complements the first one, which allows to remove module paths, by identifying module paths that have not been considered before and therefore would lead to no delay being used during simulation. Both these methods were implemented and it was shown that they are applicable to industrial cell libraries.

Using symbolic execution for detecting paths has been studied in various domains, see for example [DKMW94, DB95]. The main goal of these studies has been to make the calculation of delays or worst case execution times more precise. For example, in [DKMW94] an accurate timing analysis for combinational circuits is introduced. There, timing information is modeled as *distributed delays*, i.e., a delay is assigned to every gate computing a logic function. This is different from the module paths, which only provide a delay value for a complete path from some input to an output, possibly traversing multiple logic gates. The goal of [DKMW94] is to compute the maximal delay that can occur in the combinational logic which is not a false path, i.e., that is functionally realizable.

In [DB95], symbolic simulation is used to verify the timing of FSM models of sequential circuits. For every such FSM, setup and hold constraints are imposed on the input signals and distributed delays are specified for the output signals. The goal of this work is to detect timing violations, which occur when the distributed delays cause transitions to occur during the setup/hold times, for some fixed clocking scheme. This requires some extra assumptions on the structure of circuits to make the timing verification feasible. Using these assumptions, the work in [DB95] then traces transitions through the circuit, which is similar to the presented derivation of existing module paths in a cell.

The work of [PBE$^+$09] also observed that module paths / timing arcs describe both functional and timing behavior. This property of module paths is used there to investigate the problem of crosstalk, i.e., physically close wires affecting each other. The crosstalk analysis assigns Boolean variables to all module paths indicating whether they are active or not. Then, Boolean constraints are added that describe functional dependencies between different module paths, e.g., a module path can only be active if a module path in the preceding cell is active. Finally, these constraints are solved by a combination of a SAT solver and a timing solver. As can already be observed from the above summary, this crosstalk analysis crucially depends on all possible module paths of a cell to be listed and to accurately reflect the functionality of the cell. Otherwise, if module paths are missing or incorrect, the analysis of [PBE$^+$09] does not consider the actual behavior. Hence, the correspondence of the module paths and the functional behavior, as was investigated in Section 5.2, is a precondition for such a technique to work.

Chapter 6

Productivity Analysis by Context-Sensitive Termination

Nowadays, computations are not performed in batch anymore, where one would first provide all required input values from which then some output values are computed eventually. Instead, working with digital devices has become much more interactive, by steadily providing new inputs from which output is computed over and over again. In the setting where all input values are provided once at the beginning, termination is a desired property. It guarantees that, regardless of the input values, some output values are computed after a finite amount of time. However, in the latter setting, where sequences of input values are provided, termination, deadlock, and livelock are considered harmful, as any of them would mean that no further values can be computed and the computation is stuck. Instead, it should be the case that always eventually a next output value is computed. This property is known as *productivity* and shall be studied in this chapter.

Productivity is the property that a given set of computation rules computes a desired infinite object, such as a *stream* of output values. A stream can be seen as a mapping from the natural numbers to some value domain, which can also be understood as describing an infinite list. However, also for other structures infinite objects are of interest, such as for example the *Stern-Brocot Tree* [Ste58, Bro61] (see [Hay00] for a detailed description of its history), which is an infinite search tree containing all positive rational numbers. Also, mixtures of finite and infinite structures can occur, a prominent example being lists in the programming language Haskell [Pey03], which can be finite (by ending with a sentinel "[ ]") or can go on forever. For this reason, the techniques for proving productivity presented in this chapter are not restricted to streams, instead they are also applicable to these more general structures.

Also for hardware cells it is desired that always a next stable value is eventually computed. Thus, techniques proving productivity can be used to assert this, which was implicitly assumed in the previous chapters. However, the input values of cells are often not specified and are left up to the concrete environment in which a cell is being used. To overcome this problem, the concrete sequences of input values are abstracted away by allowing all possible sequences. Thereby, one can guarantee that the cell computes stable values, regardless of the environment in which it is being used.

In Section 6.1 the problem of productivity is stated formally. For this purpose, *Term Rewrite Systems (TRS)* [BN98, Ter03] are being used, which describe rules to perform computations. Certain restrictions on the term rewrite systems have to be imposed to make the analysis of productivity feasible. These restrictions are combined in the definition of *proper specifications*. In the literature, only *orthogonal* specifications have been considered. This however conflicts with the idea of allowing all possible input sequences, since orthogonality requires the computations to be deterministic. Therefore, the restriction to orthogonal specifications is ultimately removed. Section 6.2, which is based on [ZR10a], first considers orthogonal specifications and presents a technique to prove their productivity by context-sensitive termination. Then in Section 6.3, which is based on [Raf11], the previous results are extended by removing the restriction to orthogonal, i.e., deterministic, specifications. This allows to prove stabilization of hardware cells, as illustrated in Section 6.4.

6.1 Term Rewriting, Specifications, and Productivity

Term Rewriting [BN98, Ter03] is a theoretical model of computation, where steps of a computation are made by successively applying rules. These rules are given as pairs of *terms*, which are syntactic constructions consisting of constants and function symbols, together with variables that can be substituted by other terms. Even though it is a rather simple model, it is both Turing complete and still easily understandable, whereas for Turing machines [Tur36] the latter is arguably not the case.

To describe a *Term Rewrite System (TRS)*, first the notion of (finite) *term* is formalized. A term is either a *variable* x from some countably infinite set $\mathcal{V}$, or it is inductively constructed using a *function symbol* f from some given set Σ (called *signature*). For every function symbol f, the function ar assigns an *arity* to it (denoted $\mathsf{ar}(f) = n$, which informally speaking is the number of arguments that f requires). Then, a term is constructed by applying f to $\mathsf{ar}(f) = n$ terms $t_1, \dots, t_n$ to form the term $f(t_1, \dots, t_n)$. Note that n can also be 0, in that case f is also called a *constant*. The set $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the smallest set such that it contains all terms constructed according to the above rules. Given a non-variable term $t = f(t_1, \dots, t_n)$, the *root* symbol of that term is denoted $\mathrm{root}(t) = f$.

Terms can be interpreted as trees, where the nodes are labeled with either a variable or a function symbol and the children are the arguments of the function symbol. A *subterm* of a term is identified by a word from $\mathbb{N}^*$ that indicates the path traversed in the tree to arrive at the subterm. Not all words of $\mathbb{N}^*$ identify a subterm, since function symbols have fixed arities and also variables do not have any arguments. Hence, the set $\mathrm{Pos}(t) \subseteq \mathbb{N}^*$ denotes the set of all *positions* occurring in a term t. This set is defined as $\mathrm{Pos}(x) = \{\epsilon\}$ for all $x \in \mathcal{V}$ and $\mathrm{Pos}(f(t_1, \dots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \le i \le \mathsf{ar}(f), p \in \mathrm{Pos}(t_i)\}$ for a term $f(t_1, \dots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$. Here, ϵ denotes the empty word. The subterm $t|_p$ identified by some position $p \in \mathrm{Pos}(t)$ is defined as $t|_\epsilon = t$ and $f(t_1, \dots, t_n)|_{i.p} = t_i|_p$. Positions can be partially ordered, where for $p, p' \in \mathrm{Pos}(t)$, $p \le p'$ if $p' = p.\hat{p}$ for some $\hat{p} \in \mathrm{Pos}(t|_p)$. Otherwise, if neither $p \le p'$ nor $p' \le p$, p and p' are *independent*, denoted $p \parallel p'$. Subtraction of positions is defined for $p, p' \in \mathrm{Pos}(t)$ with $p \le p'$ as $p' - p = \hat{p}$ if $p' = p.\hat{p}$. Replacing a subterm $t|_p$ by another term t' is denoted $t[t']_p$ and is defined as $t[t']_\epsilon = t'$ and $f(t_1, \dots, t_n)[t']_{i.p} = f(t_1, \dots, t_i[t']_p, \dots, t_n)$. Finally, a *ground term* is a term that does not contain any variables, which is a term $t \in \mathcal{T}(\Sigma, \emptyset)$, also denoted as $t \in \mathcal{T}(\Sigma)$.

To replace variables by other terms, *substitutions* are used. A substitution is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$, which assigns every variable a term that it is replaced with. Often, a variable is replaced by itself, i.e., no effective change is made to a variable. If the *domain* $\mathrm{dom}(\sigma) = \{v \in \mathcal{V} \mid \sigma(v) \neq v\}$ of a substitution σ is finite, it will be denoted $\sigma = \{x_1 := t_1, \ldots, x_k := t_k\}$ with pairwise disjoint variables $x_i \in \mathcal{V}$ and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$ for $i = 1, \ldots, k$, which defines the substitution $\sigma(x_i) = t_i$ and $\sigma(y) = y$ for all $y \notin \{x_1, \ldots, x_k\}$. Application of a substitution σ to a term t is denoted $t\sigma$ and defined as $x\sigma = \sigma(x)$ for all $x \in \mathcal{V}$ and $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ for all $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$. A term t' is *matched* by another term t if a substitution σ exists such that $t\sigma = t'$.

Terms can also be *sorted*. In that case, every function symbol is not assigned an arity, but a *type*. Let $S = \{s_1, \ldots, s_k\}$ be a set of sorts and let $\mathcal{V} = \biguplus_{j=1}^{k} \mathcal{V}_{s_j}$ with all $\mathcal{V}_{s_i}$, $\mathcal{V}_{s_j}$ pairwise disjoint. Then, $\mathsf{ty}(f) = s_1^{n_1} \times \cdots \times s_k^{n_k} \to s_j$ is the function assigning the type to every function symbol $f \in \Sigma$. This means that f requires n_1 argument terms of sort s_1, n_2 arguments of sort s_2, etc. A term is of sort s_j, if it either is a variable $x \in \mathcal{V}_{s_j}$, or it is a term $f(t_1, \ldots, t_n)$ with $\mathsf{ty}(f) = s_1^{n_1} \times \cdots \times s_k^{n_k} \to s_j$. In the latter case, the function symbol f is also said to be of sort s_j. Having sorts also allows to partition the signature Σ into pairwise disjoint sets Σ_{s_j}, where $f \in \Sigma_{s_j}$ iff f is of sort s_j. Furthermore, a function symbol f with type $\mathsf{ty}(f) = s_1^{n_1} \times \cdots \times s_k^{n_k} \to s_j$ is assigned arities $\mathsf{ar}_{s_j}(f) = n_j$ for all $1 \leq j \leq k$. The unsorted arity $\mathsf{ar}(f)$ is defined as $\mathsf{ar}(f) = \sum_{j=1}^{k} \mathsf{ar}_{s_j}(f)$.

A *term rewrite system* (TRS) is a collection $\mathcal{R} \subseteq \mathcal{T}(\Sigma, \mathcal{V})^2$ of pairs of terms. Instead of $(\ell, r) \in \mathcal{R}$, the notation $\ell \to r \in \mathcal{R}$ is used. This already indicates the direction in which such rules are applied. It is required that for a rule $\ell \to r \in \mathcal{R}$, the term ℓ is not a variable and all variables occurring in r are also occurring in ℓ. A term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ can be *rewritten* to a term t' with rule $\ell \to r \in \mathcal{R}$ at position $p \in \mathrm{Pos}(t)$, denoted $t \to_{\ell \to r, p} t'$, if $t|_p = \ell\sigma$ for some substitution σ and $t' = t[r\sigma]_p$. The term $t|_p$ is also called a *redex*. Note that the subscript $\ell \to r$ can be replaced by $\mathcal{R}$ if the concrete rule is not of importance and that subscripts can be left out if the rule or the position are not relevant. If a term t cannot be rewritten, then t is said to be in *normal form*, which is also denoted $t \not\to$.

A term rewrite system $\mathcal{R}$ is called *left-linear* if for all $\ell \to r \in \mathcal{R}$ the term ℓ is *linear*, i.e., no variable $x \in \mathcal{V}$ occurs more than once in ℓ. Formally, this can be expressed by requiring that for all positions $p, p' \in \mathrm{Pos}(\ell)$, $\ell|_p = \ell|_{p'} \in \mathcal{V} \implies p = p'$. The TRS $\mathcal{R}$ is called *overlapping* if for some rules $\ell_1 \to r_1, \ell_2 \to r_2 \in \mathcal{R}$, position $p_1 \in \mathrm{Pos}(\ell_1)$ such that $\ell_1|_{p_1} \notin \mathcal{V}$, and substitutions σ_1, σ_2 it holds that $\ell_1|_{p_1}\sigma_1 = \ell_2\sigma_2$ and $p_1 \neq \epsilon$ if $\ell_1 \to r_1 = \ell_2 \to r_2$. Otherwise, $\mathcal{R}$ is called *non-overlapping*. A TRS $\mathcal{R}$ is called *orthogonal* if it is both left-linear and non-overlapping, otherwise $\mathcal{R}$ is called *non-orthogonal*. Finally, $\mathcal{R}$ is called *terminating* if no infinite reduction sequence $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$ exists.

The set of *constructors* $\mathcal{C}$ of a TRS $\mathcal{R}$ is the set of symbols that do not occur at the root of any rule's left-hand side, i.e., $\mathcal{C} = \Sigma \setminus \{\mathrm{root}(\ell) \mid \ell \to r \in \mathcal{R}\}$. The set of *defined symbols* Σ^{def} is the set of all symbols occurring as root of left-hand sides, i.e., $\Sigma^{\mathrm{def}} = \{\mathrm{root}(\ell) \mid \ell \to r \in \mathcal{R}\} = \Sigma \setminus \mathcal{C}$.

Context-Sensitive Rewriting

The variant of rewriting with the restriction that rewriting inside certain arguments of certain symbols is disallowed is called *context-sensitive rewriting* [Luc98, Luc02].

In context-sensitive rewriting, for every symbol f the set $\mu(f)$ of arguments of f is specified inside which rewriting is allowed. More precisely, μ-rewriting $\xrightarrow{\mu}_{\mathcal{R}}$ with respect to a TRS $\mathcal{R}$ is defined inductively by

- if $\ell \to r \in \mathcal{R}$ and σ is a substitution, then $\ell\sigma \xrightarrow{\mu}_{\mathcal{R}} r\sigma$;
- if $i \in \mu(f)$ and $t_i \xrightarrow{\mu}_{\mathcal{R}} t'_i$ and $t'_j = t_j$ for all $j \neq i$, then $f(t_1, \dots, t_n) \xrightarrow{\mu}_{\mathcal{R}} f(t'_1, \dots, t'_n)$.

Such a *replacement map* μ can also be used to partition the set of positions of a term into the *allowed* and *blocked* positions. For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, the allowed positions of t are defined as $\mathrm{Pos}_\mu(t) = \{\epsilon\} \cup \{i.p \mid i \in \mu(\mathrm{root}(t)), p \in \mathrm{Pos}_\mu(t|_i)\}$ and the set of *blocked* positions of t as $\mathrm{blocked}_\mu(t) = \mathrm{Pos}(t) \setminus \mathrm{Pos}_\mu(t)$. Context-sensitive rewriting then is the restriction of the rewrite relation to those redexes on positions from Pos_μ, i.e., $t \xrightarrow{\mu}_{\ell \to r, p} t'$ iff $t \to_{\ell \to r, p} t'$ and $p \in \mathrm{Pos}_\mu(t)$. A TRS $\mathcal{R}$ over a signature Σ, together with some replacement map μ, is called *μ-terminating* iff no infinite $\xrightarrow{\mu}_{\mathcal{R}}$-chain exists. Proving μ-termination has received quite some attention in the recent past and strong tools, such as for example AProVE [GSKT06] and μ-Term [L$^+$], exist for it. These tools combine approaches to transform μ-termination into standard termination problems [GM04, Luc06] and approaches that adapt their techniques (such as dependency pairs [AG00]) by taking μ-rewriting into account [AGL06, AEF$^+$08].

Infinite Terms

So far, only finite terms have been considered. However, in an infinite computation, infinite terms can occur in the limit. Intuitively, a term (both finite and infinite) is defined by saying which symbol is at which position. Here, a *position* $p \in \mathbb{N}^*$ is again a finite sequence of natural numbers. In order to be a proper term, some requirements have to be satisfied as indicated in the following definition. This definition is an extension of the definition in [Zan08] to sorted terms, which includes the unsorted case by just considering a single sort. In the below definition, the symbol $\perp$ represents undefined.

Definition 6.1.1. A (possibly infinite) *term* over a sorted signature $\Sigma = \biguplus_{j=1}^{k} \Sigma_{s_j}$ is defined to be a map $t : \mathbb{N}^* \to \Sigma \cup \{\perp\}$ such that

- the root $t(\epsilon)$ of the term t is a function symbol from Σ, so $t(\epsilon) \in \Sigma$, and
- for all $p \in \mathbb{N}^*$ and all $i \in \mathbb{N}$,

$$\begin{array}{rcl} t(p.i) \in \Sigma_{s_j} & \iff & t(p) \in \Sigma \wedge \mathsf{ty}(t(p)) = s_1^{n_1} \times \cdots \times s_k^{n_k} \to s_o \\ & & \wedge \ \sum_{r=1}^{j-1} n_r < i \leq \sum_{r=1}^{j} n_r. \end{array}$$

So $t(p.i) = \perp$ for all p, i not covered by the above two cases.

The set of all ground terms over Σ is denoted $\mathcal{T}^\infty(\Sigma)$.

An alternative equivalent definition of $\mathcal{T}^\infty(\Sigma)$ can be given based on co-algebra. Another alternative uses metric completion, where infinite terms are limits of finite terms. However, for the results in this thesis these alternatives are not needed.

A position $p \in \mathbb{N}^*$ satisfying $t(p) \in \Sigma_{s_j}$ is called a *position of* t of sort s_j. The *depth* of a position $p \in \mathbb{N}^*$ is the length of p considered as a string. The usual notion of finite term coincides with a term in this setting having finitely many positions, that is, $t(p) = \bot$ for all but finitely many p.

Also for a symbol $f \in \Sigma$ with $\mathsf{ty}(f) = s_1^{n_1} \times \cdots \times s_k^{n_k} \to s_o$ and terms $t_{1,1}, \ldots, t_{1,n_1}, t_{2,1} \ldots, t_{k,n_k} \in \mathcal{T}^\infty(\Sigma)$ where $t_{i,j}$ is of sort s_i, the term t with $t(\epsilon) = f$, $t(i.p) = t_j(p)$ for $1 \leq j \leq k$ such that $\sum_{r=1}^{j-1} n_r < i \leq \sum_{r=1}^{j} n_r$, and $t(i.p) = \bot$ otherwise is denoted $t = f(t_{1,1}, \ldots, t_{1,n_1}, t_{2,1}, \ldots, t_{k,n_k})$.

In this thesis, only infinite terms constructed over a set $\mathcal{C}$ of constructors and a set $\mathcal{D}$ of data (disjoint from $\mathcal{C}$) will be considered. Hence, those terms will be two-sorted[1]: a sort s for the (infinite) terms to be defined (mnemonic: *structure*), and a sort d for the *data*. Thus, every $f \in \mathcal{C}$ is assumed to be of type $d^m \times s^n \to s$ for some $m, n \in \mathbb{N}$, where $\mathsf{ar}_d(f) = m$, $\mathsf{ar}_s(f) = n$, and $\mathsf{ar}(f) = \mathsf{ar}_d(f) + \mathsf{ar}_s(f)$.

In case $\mathsf{ar}_s(f) > 0$ for all $f \in \mathcal{C}$ then no finite terms exist. This holds for example for streams. In case $\mathsf{ar}_d(f) = 0$ for all $f \in \mathcal{C}$ then no position of sort $\mathcal{D}$ exist, and terms do not depend on $\mathcal{D}$.

Example 6.1.2 (Streams)**.** Let $\mathcal{D}$ be an arbitrary given non-empty data set, and let $\mathcal{C} = \{:\}$, with $\mathsf{ar}_d(:) = \mathsf{ar}_s(:) = 1$. Then $\mathcal{T}^\infty(\mathcal{C} \uplus \mathcal{D})$ coincides with the usual notion of *streams* over $\mathcal{D}$, being functions from $\mathbb{N}$ to $\mathcal{D}$. More precisely, a function $f : \mathbb{N} \to \mathcal{D}$ gives rise to an infinite term t defined by $t(2^n) = :$ and $t(2^n.1) = f(n)$ for every $n \in \mathbb{N}$, and $t(w) = \bot$ for all other strings $w \in \mathbb{N}^*$. Conversely, every $t : \mathbb{N}^* \to \mathcal{C} \uplus \mathcal{D}$ satisfying the requirements of the definition of a term is of this shape. Note that if $|\mathcal{D}| = 1$, then there exists only one such term.

In case $\mathcal{D}$ is finite, an alternative approach is not to consider the binary constructor ':', but unary constructors for every element of $\mathcal{D}$. In this approach $\mathcal{D}$ does not play a role and is irrelevant.

Example 6.1.3 (Finite and infinite lists)**.** Let $\mathcal{D}$ be an arbitrary given non-empty data set, and let $\mathcal{C} = \{:, \mathsf{nil}\}$, with $\mathsf{ar}_d(:) = \mathsf{ar}_s(:) = 1$ and $\mathsf{ar}_d(\mathsf{nil}) = \mathsf{ar}_s(\mathsf{nil}) = 0$. Then $\mathcal{T}^\infty(\mathcal{C} \uplus \mathcal{D})$ covers both the *streams* over $\mathcal{D}$ as in Example 6.1.2 and the usual (finite) lists. As in Example 6.1.2, a function $f : \mathbb{N} \to \mathcal{D}$ gives rise to an infinite term t defined by $t(2^n) = :$ and $t(2^n.1) = f(n)$ for every $n \in \mathbb{N}$, and $t(w) = \bot$ for all other strings $w \in \mathbb{N}^*$. The only way nil can occur is where $t(2^n) = \mathsf{nil}$ for some $n \geq 0$, $t(2^i) = :$ and $t(2^i.1) \in \mathcal{D}$ for every $i < n$, and $t(w) = \bot$ for all other strings $w \in \mathbb{N}^*$, in this way representing a finite list of length n. Conversely, every $t : \mathbb{N}^* \to \mathcal{C} \uplus \mathcal{D}$ satisfying the requirements of the definition of a term is of one of these two shapes. In the literature this combination of finite and infinite lists is sometimes called *lazy lists*.

Example 6.1.4 (Binary trees)**.** Several variants of infinite binary trees fit in the format. A few examples are the following:

- Infinite binary trees with nodes labeled by $\mathcal{D}$ are obtained by choosing $\mathcal{C} = \{\mathsf{b}\}$ with $\mathsf{ar}_d(\mathsf{b}) = 1$ and $\mathsf{ar}_s(\mathsf{b}) = 2$. In Example 6.2.7 the nodes are labeled by $\mathcal{D} \times \mathcal{D}$, obtained by choosing $\mathsf{ar}_d(\mathsf{b}) = 2$ instead.
- The combination of finite and infinite binary trees with nodes labeled by $\mathcal{D}$ is obtained by choosing $\mathcal{C} = \{\mathsf{b}, \mathsf{nil}\}$ with $\mathsf{ar}_d(\mathsf{b}) = 1$, $\mathsf{ar}_s(\mathsf{b}) = 2$ and $\mathsf{ar}_d(\mathsf{nil}) = \mathsf{ar}_s(\mathsf{nil}) = 0$. In Example 6.2.2 the nodes are unlabeled, obtained by choosing $\mathsf{ar}_d(\mathsf{b}) = 0$ instead.

[1]In [Isi08, Isi10] an arbitrary many-sorted setting is proposed. The presented approaches easily generalize to a more general many-sorted setting, but for notational convenience only the restriction to the two-sorted setting is considered.

Specifications

A *specification* gives the symbols and rules that shall be used to compute an intended infinite element of $\mathcal{T}^\infty(\mathcal{C} \uplus \mathcal{D})$. As stated above, the two sorts s and d are considered. For the sort d, the data elements $\mathcal{D}$ are assumed to be ground normal forms of a terminating TRS $\mathcal{R}_d$ over a data signature Σ_d. Note that this implies that all symbols in Σ_d have types of the form $d^m \to d$ with $m \geq 0$. The real specification is given by the TRS $\mathcal{R}_s$, containing rewrite rules of a special shape and where both sides have sort s. These rules are over the signature $\Sigma_d \uplus \Sigma_s$, where the signature Σ_s contains all constructors $\mathcal{C}$ and additionally some defined symbols. Although the goal is to define elements of $\mathcal{T}^\infty(\mathcal{C} \uplus \mathcal{D})$, which are usually infinite, all terms in the specification are finite, and rewriting always refers to rewriting of finite terms. It is assumed that all terms are well-sorted, that is, the sort of a term used as argument of a function symbol is the one expected by the type of the function symbol.

The restrictions imposed on specifications are given below in the definition of *proper* specifications.

Definition 6.1.5. A *proper specification* is a tuple $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$, where Σ_d is the signature of data symbols, each of type $d^m \to d$ (then the data arity of such a symbol g is defined to be $\mathsf{ar}_d(g) = m$), Σ_s is the signature of structure symbols f, which have types of the shape $d^m \times s^n \to s$ (and data arity $\mathsf{ar}_d(f) = m$, structure arity $\mathsf{ar}_s(f) = n$), $\mathcal{C} \subseteq \Sigma_s$ is a set of *constructors*, $\mathcal{R}_d$ is a terminating TRS over the signature Σ_d, and $\mathcal{R}_s$ is a TRS containing rules $f(u_1, \ldots, u_m, t_1, \ldots, t_n) \to t$ satisfying the following properties:

- $f \in \Sigma_s \setminus \mathcal{C}$ with $\mathsf{ar}_d(f) = m$, $\mathsf{ar}_s(f) = n$,
- $f(u_1, \ldots, u_m, t_1, \ldots, t_n)$ is a well-sorted linear term,
- t is a well-sorted term of sort s, and
- for all $1 \leq i \leq n$ and for all $p \in \mathrm{Pos}(t_i)$ such that $t_i|_p$ is not a variable and $\mathrm{root}(t_i|_p) \in \Sigma_s$, it holds that $\mathrm{root}(t_i|_{p'}) \notin \mathcal{C}$ for all $p' < p$ (i.e., no structure symbol is below a constructor).

Furthermore, $\mathcal{R}_s$ is required to be *exhaustive*, meaning that for every defined function symbol $f \in \Sigma_s \setminus \mathcal{C}$ with $\mathsf{ar}_d(f) = m$, $\mathsf{ar}_s(f) = n$, ground normal forms $u_1, \ldots, u_m \in \mathcal{T}(\Sigma_d)$, and ground terms $t_1, \ldots, t_n \in \mathcal{T}(\Sigma_d \cup \Sigma_s)$ such that for every $1 \leq i \leq n$, $t_i = c_i(u'_1, \ldots, u'_k, t'_1, \ldots, t'_l)$ with $u'_j \in \mathcal{T}(\Sigma_d)$ being a normal form for $1 \leq j \leq k = \mathsf{ar}_d(c_i)$ and $c_i \in \mathcal{C}$, there exists at least one rule $\ell \to r \in \mathcal{R}_s$ such that ℓ matches the term $f(u_1, \ldots, u_m, t_1, \ldots, t_n)$.

A proper specification $\mathcal{S}$ is called *orthogonal*, if $\mathcal{R}_d \cup \mathcal{R}_s$ is orthogonal, otherwise it is called *non-orthogonal*.

The above definition coincides with the definition of proper specifications given in [ZR10a] for orthogonal proper specifications[2], which in turn are a generalization of proper stream specifications as given in [Zan09, ZR10b]. Fixing $\mathcal{C}, \mathcal{D}$, typically

[2]To see this, one should observe that a defined symbol cannot occur on a non-root position of a left-hand side. This holds since otherwise the innermost such symbol would have variables and constructors as structure arguments and data arguments that do not unify with any of the data rules (due to orthogonality), which therefore are normal forms and can be instantiated to ground normal forms. Thus, exhaustiveness would require a left-hand side to match this term when instantiating all structure variables with some terms having a constructor root, which would give a contradiction to non-overlappingness.

a proper specification will be given by $\mathcal{R}_d, \mathcal{R}_s$ in which Σ_d, Σ_s and the arities are left implicit since they are implied by the terms occurring in $\mathcal{R}_d, \mathcal{R}_s$. If a proper specification is only given by $\mathcal{R}_s$, then $\mathcal{R}_d$ is assumed to be empty.

Productivity

A specification is called *productive* for a given ground term of sort s if every finite part of the intended resulting infinite terms can be computed in finitely many steps. However, it has been left unclear what computations have to be considered. In the following, two different notions of productivity will be presented: *weak* productivity and *strong* productivity. Both of these notions were already defined in [End10] and are akin to weak and strong normalization in the realm of termination.

Weak productivity requires the existence of a reduction to a constructor normal form (which can be an infinite term). Thus, weak productivity is equivalent to the following.

Definition 6.1.6. A proper specification $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ is *weakly productive* for a ground term t of sort s if for every $k \in \mathbb{N}$ there is a reduction $t \rightarrow^*_{\mathcal{R}_d \cup \mathcal{R}_s} t'$ for which every symbol of sort s in t' on depth less than k is in $\mathcal{C}$.

To define strong productivity, the notion of outermost-fair rewrite sequences has to be introduced first.

Definition 6.1.7.

- A redex is called *outermost* iff it is not a strict subterm of another redex.
- A redex $t|_p = \ell\sigma$ is said to *survive* a reduction step $t \rightarrow_{\ell' \rightarrow r', q} t'$ if $p \parallel q$, or if $p < q$ and $t' = t[\ell\sigma']_p$ for some substitution σ' (i.e., the same rule can still be applied at p).
- A rewrite sequence (reduction) is called *outermost-fair*, iff there is no outermost redex that survives as an outermost redex infinitely long.
- A rewrite sequence (reduction) is called *maximal*, iff it is infinite or ends in a *normal form* (a term that cannot be rewritten further).

This allows to define the notion of strong productivity, as in [End10]. Here, it is again noted that a constructor normal form can be characterized by having, for every $d \in \mathbb{N}$, only constructors on depth d or less.

Definition 6.1.8. A proper specification $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ is called *strongly productive* iff for every ground term t of sort s all maximal outermost-fair rewrite sequences starting in t end in (i.e., have as limit for infinite sequences) a constructor normal form.

Corollary 6.1.9. *A proper specification* $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ *is* strongly productive *iff for all* $k \in \mathbb{N}$ *and all maximal outermost-fair rewrite sequences* $t_0 \rightarrow_{\mathcal{R}_d \cup \mathcal{R}_s} t_1 \rightarrow_{\mathcal{R}_d \cup \mathcal{R}_s} \ldots$ *starting in a ground term* t_0 *of sort s, there exists* $j \in \mathbb{N}$ *such that* $\mathrm{root}(t_j|_p) \in \mathcal{C}$ *for all positions* $p \in \mathrm{Pos}(t_j)$ *of depth* k *or less.*

In general, weak and strong productivity differ, since only requiring the existence of a reduction to a normal form (or to a constructor prefix of arbitrary depth) does not guarantee a computation to reach it. This is the case because of the possible non-deterministic choices that exist in non-orthogonal proper specifications. The following example illustrates this fact.

Example 6.1.10. Consider two proper specifications with the TRSs $\mathcal{R}_s$ and $\mathcal{R}'_s$ consisting of the following rules:

$$\begin{array}{rlcl} \mathcal{R}_s: & \mathsf{maybe} & \to & 0 : \mathsf{maybe} \\ & \mathsf{maybe} & \to & \mathsf{maybe} \end{array} \qquad \begin{array}{rlcl} \mathcal{R}'_s: & \mathsf{random} & \to & 0 : \mathsf{random} \\ & \mathsf{random} & \to & 1 : \mathsf{random} \end{array}$$

Both specifications are not orthogonal, since both the rules for maybe and those for random overlap. The specification containing $\mathcal{R}_s$ is not strongly productive, since it admits the infinite outermost-fair reduction $\mathsf{maybe} \to \mathsf{maybe} \to \ldots$ that never produces any constructors. However, there exists an infinite reduction producing infinitely many constructors starting in the term maybe, namely $\mathsf{maybe} \to 0 : \mathsf{maybe} \to 0 : 0 : \mathsf{maybe} \to \ldots$. Therefore, this specification is weakly productive.

For the specification containing the TRS $\mathcal{R}'_s$, i.e., the rules for random, the specification is strongly productive (and therefore also weakly productive), since no matter what rule of random is chosen, an element of the stream is created.

However, for the case of orthogonal proper specifications, weak and strong productivity coincide, as was shown in [End10]. Hence, for orthogonal proper specifications, it is sufficient to only speak about *productivity*.

An important consequence of productivity of an orthogonal specification is *well-definedness*: Every term admits a unique interpretation as an infinite term. Intuitively, existence follows from taking the limit of the process of computing a constructor on every level, and reduce data terms to normal form. Uniqueness follows form orthogonality. An investigation of well-definedness of orthogonal stream specifications has been performed in [Zan09].

As in [ZR10b], productivity is required for all finite ground terms of sort s rather than a single one. This is different from [EGH$^+$07, EGH08] where productivity of an initial start-term is investigated. The following two propositions state that when considering all terms, reaching a constructor on every arbitrary depth is equivalent to reaching a constructor at the root. As the latter characterizations are simpler, they form the basis of all further observations on productivity in this thesis. In [Isi08, Isi10] productivity is also required for infinite terms, being a stronger restriction than considering all finite terms. This will be illustrated in Example 6.2.5.

Proposition 6.1.11. *A proper specification* $(\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ *is* weakly productive *if and only if every ground term t of sort s admits a reduction $t \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} t'$ for which* $\mathrm{root}(t') \in \mathcal{C}$.

Proof. The "only if" direction of the proposition is obvious. For the "if" direction, the following claim is proved by induction on k.

> **Claim:** Let $k \in \mathbb{N}$, and for all ground terms t of sort s, $t \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} t'$ with $\mathrm{root}(t') \in \mathcal{C}$. Then $t \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} t''$ for a term t'' in which every symbol of sort s on depth less than k is in $\mathcal{C}$.

If $k = 1$, then the claim directly holds by choosing $t'' = t'$.

Otherwise, $t \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} t' = c(u_1, \ldots, u_m, t_1, \ldots, t_n)$ with $\mathrm{root}(t') = c \in \mathcal{C}$, with c of type $d^m \times s^n \to s$. Applying the induction hypothesis to $t_1, \ldots, t_n$ yields

$t_i \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} t''_i$, where all symbols of sort s in t''_i on depth $< k-1$ are from $\mathcal{C}$, for $i = 1, \ldots, n$. Now

$$t \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} f(u_1, \ldots, u_m, t_1, \ldots, t_n) \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} c(u_1, \ldots, u_m, t''_1, \ldots, t''_n)$$

proves the claim. □

A similar characterization also exists for strong productivity. Of course, for strong productivity the existence of a reduction to a term having a constructor as root symbol is not sufficient. Instead, all maximal outermost-fair reductions have to be considered.

Proposition 6.1.12. *A proper specification* $(\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ *is* strongly productive*, iff for every maximal outermost-fair reduction* $t_0 \to_{\mathcal{R}_d \cup \mathcal{R}_s} t_1 \to_{\mathcal{R}_d \cup \mathcal{R}_s} \ldots$ *with* t_0 *being a ground term of sort* s *there exists* $k \in \mathbb{N}$ *such that* $\mathrm{root}(t_k) \in \mathcal{C}$*.*

Proof. The "only if"-direction is trivial. For the "if"-direction, it is shown inductively that for every depth $z \in \mathbb{N}$ and every maximal outermost-fair reduction $\rho \equiv t_0 \to_{p_0} t_1 \to_{p_1} \ldots$ there exists an index $j \in \mathbb{N}$ such that for all positions $p \in \mathrm{Pos}(t_j)$ of sort s with $|p| < z$, $\mathrm{root}(t_j|_p) \in \mathcal{C}$.

For $z = 0$, the index j can be set to 0, thus here the claim trivially holds. Otherwise, an index $k \in \mathbb{N}$ exists such that $\mathrm{root}(t_k) \in \mathcal{C}$. Let $t_k = c(u'_1, \ldots, u'_m, t'_1, \ldots, t'_n)$ with $c \in \mathcal{C}$. Because c is a constructor, $p_l > \epsilon$ for all $l \geq k$. Define sets $P_r = \{p_l - r \mid l \geq k, p_l \geq r\}$ for $1 \leq r \leq n$ (i.e., the positions in the remaining maximal outermost-fair reduction that occur in argument r). Then, for $1 \leq r \leq n$ and $P_r = \{p^r_0, p^r_1, \ldots\}$ the reduction $t'_r = t_{r,0} \to_{p^r_0} t_{r,1} \to_{p^r_1} \ldots$ is also a maximal outermost-fair reduction, otherwise an infinitely long surviving outermost redex would also be an infinitely long surviving outermost redex of the reduction ρ. The induction hypothesis for $z-1$ yields indices j_r for every $1 \leq r \leq n$ such that $\mathrm{root}(t_{r,j_r}|_p) \in \mathcal{C}$ for all positions $p \in \mathrm{Pos}(t_{r,j_r})$ with $|p| < z-1$. Since all these reductions were taken from the original reduction, define $j = k + \sum_{i=1}^{n} j_i$, which shows that the initial reduction has the form $t_0 \to^* t_k = c(u'_1, \ldots, u'_m, t'_1, \ldots, t'_n) \to^* c(u''_1, \ldots, u''_m, t''_1, \ldots, t''_n) = t_j$, where $t_{r,j_r} \to^* t''_r$ for every $1 \leq r \leq n$. Since there are only constructors in t_{r,j_r} for depths $< z-1$, these constructors are still present in t''_r. This proves the proposition, since $c \in \mathcal{C}$ and thus for all positions $p \in \mathrm{Pos}(t_j)$ of sort s with $|p| < z$, $\mathrm{root}(t_j|_p) \in \mathcal{C}$. □

6.2 Productivity of Orthogonal Specifications

This section presents techniques that can be used to prove productivity of orthogonal proper specifications, as defined in the previous Section 6.1. Such specifications can be used to describe computations that are deterministic, hence they must be completely specified. As discussed in the previous section, computations are specified by a number of rewrite rules that are interpreted as a lazy functional program. Then productivity can be characterized and investigated as a property of term rewriting, as was investigated before in [EGH$^+$07, Isi08, EGH08, ZR10b, Isi10]. The work presented in this section is based on [ZR10a].

Streams, as was shown in Example 6.1.2, can be seen as infinite terms. Even when restricting to data structures representing the result of a computation, it is natural not to restrict to streams. In case the computation possibly ends, then the result is not a stream but a finite list, and when parallelism is considered, naturally infinite trees come in. Hence, this section develops techniques for automatically proving productivity of specifications in a wide range of infinite data structures, including streams, the combination with finite lists, and several kinds of infinite trees. Earlier techniques specifically for stream specifications were given in [EGH$^+$07, EGH08, ZR10b]. A key idea of the presented approach is to prove productivity by proving termination of *context-sensitive rewriting* [Luc98, Luc02], that is, rewriting in which rewriting is disallowed inside particular arguments of particular symbols. As strong tools like AProVE [GSKT06] and μ-Term [L$^+$] have been developed to prove termination of context-sensitive rewriting automatically, the power of these tools can now be exploited to prove productivity automatically. As the underlying technique is completely different from the technique of [EGH$^+$07, EGH08], it is expected that both approaches have their own merits. Indeed, there are examples where the technique of [EGH$^+$07, EGH08] fails whereas the presented technique based on context-sensitive termination succeeds. The comparison the other way around is hard to make as the technique of [EGH$^+$07, EGH08] only applies for proving productivity for a single ground term, whereas here productivity is proven for all ground terms.

The first technique to prove productivity of an orthogonal proper specification is given below. It is a simple syntactic criterion, which can also be seen as a particular case of the analysis of friendly nesting specifications as given in [EGH08].

Theorem 6.2.1. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be an orthogonal proper specification in which for every $\ell \to r \in \mathcal{R}_s$ the term r is not a variable and* $\mathrm{root}(r) \in \mathcal{C}$*. Then $\mathcal{S}$ is productive.*

Proof. According to Proposition 6.1.11 for every ground term t of sort s it suffices to prove that $t \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} t'$ for a term t' satisfying $\mathrm{root}(t') \in \mathcal{C}$. This is done by induction on t. Let $t = f(u_1, \ldots, u_m, t_1, \ldots, t_n)$ for $\mathsf{ar}_d(f) = m$, $\mathsf{ar}_s(f) = n$. If $f \in \mathcal{C}$, nothing has to be done. So assume that $f \in \Sigma_s \setminus \mathcal{C}$. As they are ground terms of sort d, all u_i rewrite to elements of $\mathcal{D}$. By the induction hypothesis, all t_i rewrite to terms with root in $\mathcal{C}$, and in which the arguments of sort d rewrite to elements of $\mathcal{D}$. Now by the exhaustiveness requirement of properness, the resulting term matches with the left-hand side of a rule from $\mathcal{R}_s$. Due to the assumption, by rewriting according to this rule a term is obtained of which the root is in $\mathcal{C}$. $\square$

This theorem is sufficient to prove productivity of several specifications. As an example, productivity of tree specifications is considered below.

Example 6.2.2. Choose $\mathcal{C} = \{\mathsf{b}, \mathsf{nil}\}$ with $\mathsf{ar}_s(\mathsf{b}) = 2$ and $\mathsf{ar}_d(\mathsf{b}) = \mathsf{ar}_d(\mathsf{nil}) = \mathsf{ar}_s(\mathsf{nil}) = 0$ representing the combination of finite and infinite unlabeled binary trees. Then

$$\mathsf{t} \quad \to \quad \mathsf{b}(\mathsf{b}(\mathsf{nil}, \mathsf{t}), \mathsf{t})$$

is an orthogonal proper specification that is productive due to Theorem 6.2.1. The symbol t represents an infinite unlabeled tree in which the number of nodes on depth n is exactly the n-th Fibonacci number.

However, the syntactic criterion of Theorem 6.2.1 is rather weak, meaning that numerous productive specifications will not be identified as such. Therefore, in the following, a technique based on context-sensitive termination is presented.

Proving Productivity by Context-Sensitive Termination

As intended for generating infinite terms, the orthogonal TRS $\mathcal{R}_d \cup \mathcal{R}_s$ will never be terminating. However, when disallowing rewriting inside arguments of sort s of constructor symbols, it may be terminating. To disallow rewriting, context-sensitive rewriting makes use of a replacement map. The specific replacement map used for proving productivity is defined next.

Definition 6.2.3. Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be an orthogonal proper specification. The replacement map $\mu_\mathcal{S}$ is defined as $\mu_\mathcal{S}(f) = \{1, \ldots, \mathsf{ar}_d(f)\}$ for all $f \in \Sigma_d \cup \mathcal{C}$ and $\mu_\mathcal{S}(f) = \{1, \ldots, \mathsf{ar}_d(f) + \mathsf{ar}_s(f)\}$ for all $f \in \Sigma_s \setminus \mathcal{C}$.

The main theorem shows that if $\mu_\mathcal{S}$-termination holds, then the specification is productive. In the following, it is allowed to leave out the subscript $\mathcal{S}$ if the specification is clear from the context.

Theorem 6.2.4. *Let $(\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be an orthogonal proper specification for which $\mathcal{R}_d \cup \mathcal{R}_s$ is $\mu_\mathcal{S}$-terminating. Then the specification is productive.*

Proof. Define a ground μ-normal form to be a ground term that can not be rewritten by μ-rewriting. The following claim is proven by induction on t:

Claim: If t is a ground μ-normal form of sort s, then $\mathrm{root}(t) \in \mathcal{C}$.

Assume $\mathrm{root}(t) \notin \mathcal{C}$. Then $t = f(u_1, \ldots, u_m, t_1, \ldots, t_n)$ for $f \in \Sigma_s$, $u_1, \ldots, u_m$ are of sort d, and $t_1, \ldots, t_n$ are of sort s. Since $\mu(f) = \{1, \ldots, n+m\}$, they are all ground μ-normal forms. So $u_1, \ldots, u_m \in \mathcal{D}$. By the induction hypothesis all t_i have their roots in $\mathcal{C}$. Since t_i is a μ-normal form and the arguments of sort d are in $\mu(c)$ for every $c \in \mathcal{C}$, the arguments of t_i of sort d are all in $\mathcal{D}$. By the exhaustiveness requirement, a rule is applicable to t on the root level, so satisfies the restriction of μ-rewriting, contradicting the assumption that t is a μ-normal form. This concludes the proof of the claim.

According to Proposition 6.1.11, for productivity it has to be proven that every ground term t of sort s rewrites to a term having its root in $\mathcal{C}$. Apply μ-rewriting on t as long as possible. Due to μ-termination this will end in a term on which μ-rewriting is not possible, so a ground μ-normal form. Due to the claim this ground μ-normal form has its root in $\mathcal{C}$. □

This theorem allows to prove productivity of examples that have more involved definitions. The following examples shall illustrate this.

Example 6.2.5. Consider the stream specification given by the TRS $\mathcal{R}_s$:

$$\begin{array}{lcl} \mathsf{ones} & \to & 1 : \mathsf{ones} \end{array} \qquad \begin{array}{rcl} \mathsf{f}(0 : xs) & \to & \mathsf{f}(xs) \\ \mathsf{f}(1 : xs) & \to & 1 : \mathsf{f}(xs) \end{array}$$

Productivity for all ground terms including $\mathsf{f}(\mathsf{ones})$ follows from Theorem 6.2.4: Entering this rewrite system in the tool AProVE [GSKT06] or μ-Term [L$^+$] together with the context-sensitivity information that rewriting is disallowed in the second argument of ':' fully automatically yields a proof of context-sensitive termination.

In this specification f is the stream function that removes all zeros. So productivity depends on the fact that the stream of all zeros does not occur as the interpretation of a subterm of any ground term in this specification. For instance, by adding the rule zeroes $\to 0$: zeroes the specification is not productive any more as f(zeroes) does not rewrite to a term having a constructor as its root.

This also shows the difference between the requirement of productivity of all finite ground terms as is used in this thesis and the requirement in [Isi08, Isi10] of productivity of all terms, including infinite terms. There this example is not productive on the infinite term representing the stream of all zeros. Finally, it should be mentioned that the technique from [EGH08] fails to prove productivity for f(ones), since the specification is not *data obliviously* productive, i.e., it is not productive when identifying all data elements (in this case 0 and 1) with a single data element $\bullet$.

Example 6.2.6. Below, the sorted stream of Hamming numbers is specified, which are all positive natural numbers that are not divisible by other prime numbers than 2, 3 and 5, in ascending order. Here, the natural numbers are represented in Peano notation, i.e., $\mathcal{D} = \{\mathsf{s}^n(0) \mid n \geq 0\}$. For $+$ and $*$ the standard rules are used. Furthermore, a comparison function cmp is needed, for which $\mathsf{cmp}(n, m)$ yields 0 if $n = m$, $\mathsf{s}(0)$ if $n > m$, and $\mathsf{s}(\mathsf{s}(0))$ if $n < m$. So $\mathcal{R}_d$ consists of the following rules:

$$\begin{array}{rcl@{\qquad}rcl} x + 0 & \to & x & \mathsf{cmp}(0,0) & \to & 0 \\ x + \mathsf{s}(y) & \to & \mathsf{s}(x+y) & \mathsf{cmp}(\mathsf{s}(x),0) & \to & \mathsf{s}(0) \\ x * 0 & \to & 0 & \mathsf{cmp}(0,\mathsf{s}(x)) & \to & \mathsf{s}(\mathsf{s}(0)) \\ x * \mathsf{s}(y) & \to & (x*y)+x & \mathsf{cmp}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{cmp}(x,y) \end{array}$$

In $\mathcal{R}_s$ the function mul multiplies a stream element-wise by a number, whereas the function merge merges two sorted streams, using the auxiliary function f. Finally, the constant h creates the sorted stream of Hamming numbers. The rules of $\mathcal{R}_s$ read:

$$\mathsf{mul}(x, y : ys) \to x * y : \mathsf{mul}(x, ys)$$

$$\begin{array}{rcl} \mathsf{merge}(x : xs, y : ys) & \to & \mathsf{f}(\mathsf{cmp}(x,y), x : xs, y : ys) \\ \mathsf{f}(0, x : xs, y : ys) & \to & x : \mathsf{merge}(xs, ys) \\ \mathsf{f}(\mathsf{s}(0), xs, y : ys) & \to & y : \mathsf{merge}(xs, ys) \\ \mathsf{f}(\mathsf{s}(\mathsf{s}(z)), x : xs, ys) & \to & x : \mathsf{merge}(xs, ys) \end{array}$$

$$\mathsf{h} \to \mathsf{s}(0) : \mathsf{merge}(\mathsf{merge}(\mathsf{mul}(\mathsf{s}^2(0),\mathsf{h}),\mathsf{mul}(\mathsf{s}^3(0),\mathsf{h})),\mathsf{mul}(\mathsf{s}^5(0),\mathsf{h}))$$

The above is an orthogonal proper stream specification, being the folklore functional program for generating Hamming numbers, up to notational details. Productivity can be proved fully automatically: Calling μ-Term [L$^+$] together with the context-sensitivity information that rewriting is disallowed in the second argument of ':' yields a proof of context-sensitive termination. So by Theorem 6.2.4 productivity can be concluded.

For completeness it shall be mentioned that the tool of [EGH$^+$07, EGH08] also finds a proof of productivity of h in this example.

Example 6.2.7. The Calkin-Wilf tree [CW00] is a binary tree in which every node is labeled by a pair of natural numbers. The root is labeled by $(1, 1)$, and every node labeled by (m, n) has children labeled by $(m, m + n)$ and $(m + n, n)$. It can be proved that for all natural numbers $m, n > 0$ that are relatively prime the pair (m, n) occurs exactly once as a label of a node, and no other pairs occur. So the labels of the nodes represent positive rational numbers, and every positive rational number m/n occurs exactly once as a pair (m, n). There is one constructor b with $\mathsf{ar}_d(\mathsf{b}) = \mathsf{ar}_s(\mathsf{b}) = 2$. The data set $\mathcal{D}$ consisting of the natural numbers is taken from

Example 6.2.6, as is the symbol $+$ and its two rules. Now the Calkin-Wilf tree cw is defined by

$$\mathsf{cw} \;\to\; \mathsf{f}(\mathsf{s}(0), \mathsf{s}(0)) \qquad\qquad \mathsf{f}(x, y) \;\to\; \mathsf{b}(x, y, \mathsf{f}(x, x + y), \mathsf{f}(x + y, y)).$$

Productivity of this specification can be proved by μ-Term [L$^+$], which proves context-sensitive termination for the replacement map $\mu(\mathsf{cw}) = \emptyset$ and $\mu(\mathsf{f}) = \mu(\mathsf{b}) = \mu(+) = \{1, 2\}$, hence proving productivity by Theorem 6.2.4.

A related structure is the Stern-Brocot tree [Ste58, Bro61], that was independently discovered by the mathematician Moritz Stern and the watchmaker Achille Brocot in the 19-th century. It also is an infinite tree that contains every rational number exactly once, and it even forms a binary search tree, i.e., all elements in the left subtree of a node a smaller and all elements in the right subtree of a node are larger than the rational number that node is labeled with. It is constructed by repeatedly computing the *mediant* $(m + m', n + n')$ of two pairs (m, n) and (m', n'), which has the property that $\frac{m}{n} < \frac{m+m'}{n+n'} < \frac{m'}{n'}$ for $\frac{m}{n} < \frac{m'}{n'}$. Again, the specification for the tree makes use of the constructor b with $\mathsf{ar}_d(\mathsf{b}) = \mathsf{ar}_s(\mathsf{b}) = 2$, the data set $\mathcal{D}$ representing natural numbers, and the function $+$ with its two rules implementing addition. Then the Stern-Brocot tree sb is defined by the following two rules in $\mathcal{R}_s$:

$$\begin{aligned} \mathsf{sb} \;&\to\; \mathsf{g}(0, \mathsf{s}(0),\; \mathsf{s}(0), 0) \\ \mathsf{g}(m, n,\; m', n') \;&\to \\ &\mathsf{b}(m + m', n + n',\; \mathsf{g}(m, n,\; m + m', n + n'), \mathsf{g}(m + m', n + n',\; m', n') \end{aligned}$$

Context-sensitive termination of this specification can also be proven by μ-Term [L$^+$], where $\mu(\mathsf{sb}) = \emptyset$ and $\mu(\mathsf{g}) = \mu(\mathsf{b}) = \mu(+) = \{1, 2\}$. Thus, according to Theorem 6.2.4, productivity of this example has been proven.

Theorem 6.2.4 can be seen as a strengthening of Theorem 6.2.1: if all roots of right-hand sides of rules from R_s are in $\mathcal{C}$ then $\mathcal{R}_d \cup \mathcal{R}_s$ is μ-terminating, as is shown in the following proposition.

Proposition 6.2.8. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be an orthogonal proper specification in which for every $\ell \to r$ in $\mathcal{R}_s$ the term r is not a variable and* $\mathrm{root}(r) \in \mathcal{C}$. *Then $\mathcal{R}_d \cup \mathcal{R}_s$ is μ-terminating.*

Proof. Assume there exists an infinite μ-reduction. For every term in this reduction count the number of symbols from Σ_s that are on allowed positions. Due to the assumptions by every $\mathcal{R}_d$-step this number remains the same, while by every $\mathcal{R}_s$-step this number decreases by one. So this reduction contains only finitely many $\mathcal{R}_s$-steps. After these finitely many $\mathcal{R}_s$-steps an infinite $\mathcal{R}_d$-reduction remains, contradicting the assumption that $\mathcal{R}_d$ is terminating. □

The reverse direction of Theorem 6.2.4 does not hold, as is illustrated in the next example.

Example 6.2.9. Consider the proper (stream) specification $(\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$, where $\Sigma_d = \{0, 1\}$, $\mathcal{R}_d = \emptyset$, $\mathcal{C} = \{:\}$ with $\mathsf{ar}_d(:) = \mathsf{ar}_s(:) = 1$, and $\mathcal{R}_s$ being the below TRS:

$$\begin{aligned} \mathsf{p} \;&\to\; \mathsf{zip}(\mathsf{alt}, \mathsf{p}) \\ \mathsf{alt} \;&\to\; 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : xs, ys) \;&\to\; x : \mathsf{zip}(ys, xs) \end{aligned}$$

This specification is productive, as will be shown later in Example 6.2.11. However, it admits an infinite context-sensitive reduction $\mathsf{p} \to \mathsf{zip}(\mathsf{alt}, \mathsf{p})$ which is continued by repeatedly reducing the redex p.

The stream p describes the sequence of right and left turns in the well-known *dragon curve*, obtained by repeatedly folding a paper ribbon in the same direction.

Transformations for Proving Productivity

To be able to handle examples like the dragon curve, transformations of such specifications are investigated, such that productivity of the original system can be concluded from productivity of the transformed one. Whenever productivity of a specification cannot be determined directly, then one of these transformations is applied and productivity of the transformed specification is checked instead.

One such transformation is the reduction of right-hand sides, that is, a rule $\ell \to r$ of $\mathcal{R}_s$ is replaced by $\ell \to r'$ for a term r' satisfying $r \to^*_{\mathcal{R}_d \cup \mathcal{R}_s} r'$. Write $\mathcal{R} = \mathcal{R}_d \cup \mathcal{R}_s$, and write $\mathcal{R}'$ for the result of this replacement. Then, by construction, $\to_{\mathcal{R}'} \subseteq \to^+_{\mathcal{R}}$, and $\to_{\mathcal{R}} \subseteq \to_{\mathcal{R}'} \circ \leftarrow^*_{\mathcal{R}}$, that is, every $\to_{\mathcal{R}}$-step can be followed by zero or more $\to_{\mathcal{R}}$-steps to obtain an $\to_{\mathcal{R}'}$-step. The below theorem is formulated in this more general setting, such that it is applicable more generally than only for reduction of right-hand sides.

Theorem 6.2.10. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ and $\mathcal{S}' = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}'_s)$ be two orthogonal proper specifications satisfying $\to_{\mathcal{R}'} \subseteq \to^+_{\mathcal{R}}$ for $\mathcal{R} = \mathcal{R}_d \cup \mathcal{R}_s$ and $\mathcal{R}' = \mathcal{R}_d \cup \mathcal{R}'_s$. If $\mathcal{S}'$ is productive, then $\mathcal{S}$ is productive, too.*

Proof. Let $\mathcal{S}'$ be productive, i.e., every ground term t of sort s admits a reduction $t \to^*_{\mathcal{R}'} t'$ for which $\mathrm{root}(t') \in \mathcal{C}$. Then, $\to_{\mathcal{R}'} \subseteq \to^+_{\mathcal{R}}$ allows to conclude $t \to^*_{\mathcal{R}} t'$, proving productivity of $\mathcal{S}$. □

Example 6.2.11. The above Theorem 6.2.10 is applied to Example 6.2.9. Observe that the right-hand side of the rule $\mathsf{p} \to \mathsf{zip}(\mathsf{alt}, \mathsf{p})$ can be rewritten as follows:

$$\mathsf{zip}(\mathsf{alt}, \mathsf{p}) \to \mathsf{zip}(0 : 1 : \mathsf{alt}, \mathsf{p}) \to 0 : \mathsf{zip}(\mathsf{p}, 1 : \mathsf{alt})$$

Hence, the specification can be transformed by replacing $\mathcal{R}_s$ with the TRS $\mathcal{R}'_s$ consisting of the following rules:

$$\begin{aligned} \mathsf{p} &\to 0 : \mathsf{zip}(\mathsf{p}, 1 : \mathsf{alt}) \\ \mathsf{alt} &\to 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : xs, ys) &\to x : \mathsf{zip}(ys, xs) \end{aligned}$$

Clearly, this is a proper specification that is productive due to Theorem 6.2.1. Now productivity of the original specification follows from Theorem 6.2.10 and $\to_{\mathcal{R}'_s} \subseteq \to^+_{\mathcal{R}_s}$.

Concluding productivity of the original system from productivity of the transformed system is called *soundness*, the converse is called *completeness*. The following example shows the incompleteness of Theorem 6.2.10.

Example 6.2.12. Consider the two orthogonal proper (stream) specifications $\mathcal{S}$ and $\mathcal{S}'$ defined by

$$\mathcal{R}_s: \quad \begin{aligned} \mathsf{a} &\to \mathsf{f}(\mathsf{a}) \\ \mathsf{f}(xs) &\to 0 : xs \end{aligned} \qquad\qquad \mathcal{R}'_s: \quad \begin{aligned} \mathsf{a} &\to \mathsf{f}(\mathsf{a}) \\ \mathsf{f}(x : xs) &\to 0 : x : xs \end{aligned}$$

Here $\mathcal{C} = \{:\}$, $R_d = \emptyset$, $\Sigma_d = \{0\}$. Since $\mathsf{a} \rightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{a}) \rightarrow_{\mathcal{R}} 0 : \mathsf{a}$ and $\mathsf{f}(\cdots) \rightarrow_{\mathcal{R}} 0 : \cdots$, the specification $\mathcal{S}$ is productive, as a and f are the only symbols in Σ_s.

For the TRS $\mathcal{R}'_s$ it holds that $\rightarrow_{\mathcal{R}'_s} \subseteq \rightarrow^{+}_{\mathcal{R}_s}$, since any step with the rule $\mathsf{f}(x : xs) \rightarrow 0 : x : xs$ of $\mathcal{R}'_s$ can also be done with the rule $\mathsf{f}(xs) \rightarrow 0 : xs$ of $\mathcal{R}_s$. However, $\mathcal{S}'$ is not productive, as the only reduction starting in a is $\mathsf{a} \rightarrow_{\mathcal{R}'} \mathsf{f}(\mathsf{a}) \rightarrow_{\mathcal{R}'} \mathsf{f}(\mathsf{f}(\mathsf{a})) \rightarrow_{\mathcal{R}'} \cdots$ in which the root is never in $\mathcal{C}$.

Next, it is shown that with the extra requirement $\rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R}'} \circ \leftarrow^{*}_{\mathcal{R}}$, as holds for reduction of right-hand sides, both soundness and completeness hold.

Theorem 6.2.13. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ and $\mathcal{S}' = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}'_s)$ be orthogonal proper specifications satisfying $\rightarrow_{\mathcal{R}'} \subseteq \rightarrow^{+}_{\mathcal{R}}$ and $\rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R}'} \circ \leftarrow^{*}_{\mathcal{R}}$ for $\mathcal{R} = \mathcal{R}_d \cup \mathcal{R}_s$ and $\mathcal{R}' = \mathcal{R}_d \cup \mathcal{R}'_s$.*

Then $\mathcal{S}$ is productive if and only if $\mathcal{S}'$ is productive.

Proof. The "if" direction follows from Theorem 6.2.10.
For the "only-if" direction, the following claim is proven first:

> **Claim:** If $t \rightarrow_{\mathcal{R}} t'$ and $t \rightarrow^{*}_{\mathcal{R}} t''$, then there exists a term v satisfying $t' \rightarrow^{*}_{\mathcal{R}} v$ and $t'' \rightarrow^{*}_{\mathcal{R}'} v$.

Let $t \rightarrow_{\mathcal{R}} t'$ be an application of the rule $\ell \rightarrow r$ in $\mathcal{R}$, so $t = C[\ell\sigma]$ and $t' = C[r\sigma]$ for some context C and substitution σ. According to the Parallel Moves Lemma ([Ter03, Lemma 4.3.3]), $t'' = C'''[\ell\sigma_1, \ldots, \ell\sigma_n]$, and t', t'' have a common $\mathcal{R}$-reduct $C'''[r\sigma_1, \ldots, r\sigma_n]$. Due to $\ell\sigma_i \rightarrow_{\mathcal{R}} r\sigma_i$ and $\rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R}'} \circ \leftarrow^{*}_{\mathcal{R}}$ there exist t_i satisfying $\ell\sigma_i \rightarrow_{\mathcal{R}'} t_i$ and $r\sigma_i \rightarrow^{*}_{\mathcal{R}} t_i$, for all $i = 1, \ldots, n$. Now choosing $v = C'''[t_1, \ldots, t_n]$ proves the claim.

Using this claim, by induction on the number of $\rightarrow_{\mathcal{R}}$-steps from t to t' one proves the generalized claim: If $t \rightarrow^{*}_{\mathcal{R}} t'$ and $t \rightarrow^{*}_{\mathcal{R}} t''$, then there exists a term v satisfying $t' \rightarrow^{*}_{\mathcal{R}} v$ and $t'' \rightarrow^{*}_{\mathcal{R}'} v$.

Let t be an arbitrary ground term of sort s. Due to productivity of $\mathcal{S}$ there exists t' satisfying $t \rightarrow^{*}_{\mathcal{R}} t'$ and $\mathrm{root}(t') \in \mathcal{C}$. Applying the generalized claim for $t'' = t$ yields a term v satisfying $t' \rightarrow^{*}_{\mathcal{R}} v$ and $t \rightarrow^{*}_{\mathcal{R}'} v$. Since $\mathrm{root}(t') \in \mathcal{C}$ and $t' \rightarrow^{*}_{\mathcal{R}} v$ it must also be the case that $\mathrm{root}(v) \in \mathcal{C}$. Now $t \rightarrow^{*}_{\mathcal{R}'} v$ implies productivity of $\mathcal{S}'$. $\square$

Example 6.2.12 generalizes to a general application of Theorem 6.2.10 other than rewriting right-hand sides as follows. Assume a rule from $\mathcal{R}_s$ in a proper transformation contains an s-variable xs in the left-hand side being an argument of the root. Then for every $c \in \mathcal{C}$ this rule may be replaced by an instance of the same rule, obtained by replacing xs by $c(x_1, \ldots, x_m, xs_1, \ldots, xs_n)$, where $\mathsf{ar}_d(c) = m, \mathsf{ar}_s(c) = n$. If this is done simultaneously for every $c \in \mathcal{C}$, so replacing the original rule by $|\mathcal{C}|$ instances, then the result is again a proper specification. Also the requirements of Theorem 6.2.10 hold, even $\rightarrow_{\mathcal{R}'} \subseteq \rightarrow_{\mathcal{R}}$. This transformation is shown by an example.

Example 6.2.14. Productivity of the following variant of Example 6.2.9 is analyzed, in which p has been replaced by a stream function, and $\mathcal{R}_s$ is the below TRS:

$$\begin{aligned} \mathsf{p}(xs) &\rightarrow \mathsf{zip}(xs, \mathsf{p}(xs)) \\ \mathsf{alt} &\rightarrow 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : xs, ys) &\rightarrow x : \mathsf{zip}(ys, xs) \end{aligned}$$

Proving productivity by Theorem 6.2.1 fails. Also proving productivity with the technique of Theorem 6.2.4 fails, since there exists the infinite context-sensitive reduction

$$\mathsf{p}(\mathsf{alt}) \;\to\; \mathsf{zip}(\mathsf{alt}, \underline{\mathsf{p}(\mathsf{alt})}) \;\to\; \dots.$$

Furthermore, reducing the right-hand side of $\mathsf{p}(xs) \to \mathsf{zip}(\sigma, \mathsf{p}(xs))$ can only be done by applying the first rule, not creating a constructor as the root of the right-hand side. What blocks rewriting using the zip rule is the variable xs in the first argument of zip. Therefore, Theorem 6.2.10 is applied as sketched above. Note that $\mathcal{C} = \{:\}$, so the rule $\mathsf{p}(xs) \to \mathsf{zip}(xs, \mathsf{p}(xs))$ is replaced by the single rule $\mathsf{p}(x : xs) \to \mathsf{zip}(x : xs, \mathsf{p}(x : xs))$ to obtain the TRS $\mathcal{R}_s'$. This now allows to rewrite the new right-hand side by the zip rule, replacing the previous rule by $\mathsf{p}(x : xs) \to x : \mathsf{zip}(\mathsf{p}(x : xs), xs)$, i.e., the TRS $\mathcal{R}_s''$ is obtained which consists of the following rules:

$$\begin{aligned}
\mathsf{p}(x : xs) &\to x : \mathsf{zip}(\mathsf{p}(x : xs), xs) \\
\mathsf{alt} &\to 0 : 1 : \mathsf{alt} \\
\mathsf{zip}(x : xs, ys) &\to x : \mathsf{zip}(ys, xs)
\end{aligned}$$

Productivity of $\mathcal{R}_s''$ follows from Theorem 6.2.1. This implies productivity of $\mathcal{R}_s'$ due to Theorem 6.2.10 which in turn implies productivity of the initial specification $\mathcal{S}$, again due to Theorem 6.2.10.

Example 6.2.15. For stream computations it is often natural to also use finite lists. The data structure combining streams and finite lists is obtained by choosing $\mathcal{C} = \{:, \mathsf{nil}\}$, with $\mathsf{ar}_d(:) = \mathsf{ar}_s(:) = 1$ and $\mathsf{ar}_d(\mathsf{nil}) = \mathsf{ar}_s(\mathsf{nil}) = 0$, as mentioned in Example 6.1.3. An example using this is defining the sorted stream $p = 1 : 2 : 2 : 3 : 3 : 3 : 4 : \cdots$ of natural numbers, in which the representation of the number n occurs exactly n times for every $n \in \mathbb{N}$.[3] Auxiliary functions are conc, implementing concatenation, copy for which $\mathsf{copy}(k, n)$ is the finite list of k copies of n for $k, n \in \mathbb{N}$, and a function f, used for generating $\mathsf{p} = \mathsf{f}(0)$. Taking $\mathcal{D}$ to be the set of ground terms over $\{0, \mathsf{s}\}$ and $\mathcal{R}_d = \emptyset$, $\mathcal{R}_s$ is chosen to consist of the following rules:

$$\begin{aligned}
\mathsf{p} &\to \mathsf{f}(0) & \mathsf{f}(x) &\to \mathsf{conc}(\mathsf{copy}(x, x), \mathsf{f}(\mathsf{s}(x))) \\
\mathsf{copy}(\mathsf{s}(x), y) &\to y : \mathsf{copy}(x, y) & \mathsf{conc}(\mathsf{nil}, xs) &\to xs \\
\mathsf{copy}(0, x) &\to \mathsf{nil} & \mathsf{conc}(x : xs, ys) &\to x : \mathsf{conc}(xs, ys)
\end{aligned}$$

Note that productivity of this system is not trivial: if the rule for f is replaced by $\mathsf{f}(x) \to \mathsf{conc}(\mathsf{copy}(x, x), \mathsf{f}(x))$, then the system is not productive.

Productivity cannot be proved directly by Theorem 6.2.1 or Theorem 6.2.4; context-sensitive termination does not even hold for the single f rule. However by replacing the f rule by the two instances

$$\mathsf{f}(0) \to \mathsf{conc}(\mathsf{copy}(0,0), \mathsf{f}(\mathsf{s}(0))) \text{ and } \mathsf{f}(\mathsf{s}(x)) \to \mathsf{conc}(\mathsf{copy}(\mathsf{s}(x), \mathsf{s}(x)), \mathsf{f}(\mathsf{s}(\mathsf{s}(x)))),$$

and then applying rewriting right-hand sides by which these two rules are replaced by

$$\mathsf{f}(0) \to \mathsf{f}(\mathsf{s}(0)) \quad \text{and} \quad \mathsf{f}(\mathsf{s}(x)) \to \mathsf{s}(x) : \mathsf{conc}(\mathsf{copy}(x, \mathsf{s}(x)), \mathsf{f}(\mathsf{s}(\mathsf{s}(x))))$$

[3]The same stream is easily defined by a specification not involving finite lists, but here this extended data structure and the use of standard operations like conc shall be illustrated.

yields a proper specification for which context-sensitive termination is proved by AProVE [GSKT06] or μ-Term [L$^+$], proving productivity of the original example by Theorem 6.2.10 and Theorem 6.2.4.

Example 6.2.16. Finally, an example in binary trees shall be considered, in which the nodes are labeled by natural numbers, so there is one constructor $\mathsf{b} : d \times s^2 \to s$ and $\mathcal{D}$ consists of ground terms over $\{0, \mathsf{s}\}$. The rules are

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{c} & \to & \mathsf{b}(0, \mathsf{f}(\mathsf{g}(0), \mathsf{left}(\mathsf{c})), \mathsf{g}(0)) & \mathsf{left}(\mathsf{b}(x, xs, ys)) & \to & xs \\ \mathsf{g}(x) & \to & \mathsf{b}(x, \mathsf{g}(\mathsf{s}(x)), \mathsf{g}(\mathsf{s}(x))) & \mathsf{f}(\mathsf{b}(x, xs, ys), zs) & \to & \mathsf{b}(x, ys, \mathsf{f}(zs, xs)) \end{array}$$

To get an impression of the hardness of this example, observe that f and left are similar to zip and tail for streams, respectively, and the recursion in the rule for c has the flavor of $\mathsf{c} \to 0 : \mathsf{zip}(\cdots, \mathsf{tail}(\mathsf{c}))$. The tool described in the following section proves productivity by Theorem 6.2.10 and Theorem 6.2.4, by first rewriting right-hand sides and then proving context-sensitive termination.

Implementation

The presented techniques to prove productivity of orthogonal proper specifications were implemented in a tool to check productivity automatically. It is accessible via the web-interface at the following URL:

```
http://www.win.tue.nl/~mraffels/productivity
```

The input format requires the following ingredients:

- the variables,
- the operation symbols with their types,
- the rewrite rules.

Details of the format can be seen from the examples that are available. All other information, such as the symbols in $\mathcal{C}$, is extracted by the tool from these ingredients.

As a first step, the tool checks that the input is indeed a proper specification. Checking syntactic requirements, such as no function symbol returning sort d has an argument of sort s, the TRS is 2-sorted and orthogonal, and the left-hand sides have the required shape, are all straightforward. However, to verify the last requirement of a proper specification, namely that the TRS is exhaustive, is a hard job if $\mathcal{D}$ is allowed to be the set of ground normal forms of any terminating orthogonal $\mathcal{R}_d$. Only for the class of proper specifications in which $\mathcal{D}$ consists of the constructor ground terms of sort d exhaustiveness can be checked efficiently. In that case, the terms in $\mathcal{D}$ do not contain symbols occurring as root symbol in a left-hand side of a rule in $\mathcal{R}_d$. To check whether this is the case, *anti-matching*, to be described in the following chapter, is used. It provides a set of terms that match exactly those ground terms not matched by any left-hand side. For left-linear systems such as those considered here, an anti-matching set can be efficiently constructed. It can easily be shown that the normal forms of ground terms w.r.t. $\mathcal{R}_d$ are only constructor terms if and only if there is no anti-matching term that has a defined symbol as root and only terms built from constructors and variables as arguments. The idea of the proof is that such a term could be instantiated to a ground term, which is a normal form due to the anti-matching property. Then, checking exhaustiveness of $\mathcal{R}_s$ has to only consider constructor terms for both data and structure arguments.

To analyze productivity of a given proper specification, the tool first investigates whether Theorem 6.2.1 can be applied directly: it checks whether the roots of all right-hand sides are constructors. If this simple criterion does not hold, then it tries to show context-sensitive termination using the existing termination prover μ-Term, by which productivity will follow by Theorem 6.2.4.

If both of these first attempts fail then the tool tries to transform the given specification. Since rewriting of right-hand sides is both sound and complete, as was shown in Theorem 6.2.13, a productive specification can never be transformed into an unproductive one by this technique. Therefore, this is the first transformation to try. However, large right-hand sides often make it harder for termination tools to prove context-sensitive termination. Therefore, the tool tries to only rewrite positions on right-hand sides that appear to be needed to obtain a constructor prefix tree of a certain, adjustable depth. This is done by traversing the term in an outermost fashion and only trying to rewrite arguments if the possibly matching rules require a constructor for that particular argument. If at least one right-hand side could be rewritten, a new specification with the rewritten right-hand sides is created. Since rewriting of right-hand sides is not guaranteed to terminate, a limit is imposed onto the maximal number of rewriting steps. After rewriting the right-hand sides in this way, the tool again tries to prove productivity of the transformed TRS using the basic techniques, i.e., the syntactic criterion and context-sensitive termination.

As shown in Examples 6.2.14 and 6.2.15, it can be helpful to replace a variable by all constructors of its sort applied to variables. Therefore, in case productivity could not be shown so far, it is tried to instantiate a variable on a position of a right-hand side that is required by the rules for the defined symbol directly above it. Then the instantiated right-hand sides are rewritten again to obtain new specifications for which productivity is analyzed further.

The described transformations are applied in the order of their presentation a number of times. If a set limit of applications of transformations is reached, the tool finally tries to rewrite to deeper context-prefixes on right-hand sides and does a final check for productivity, using a larger timeout value.

Using these heuristics the tool is able to automatically prove productivity of all productive examples presented in this section. This especially includes the below example of a stream specification, which previously could not be proved to be productive by any other automated technique.

Example 6.2.17. Consider the following combination of Example 6.2.9 (describing the paper folding stream) together with the rules for function f taken from Example 6.2.5 (which remove all 0 elements from a stream):

$$\begin{aligned} \mathsf{p} &\to \mathsf{zip}(\mathsf{alt}, \mathsf{p}) & \mathsf{f}(0 : xs) &\to \mathsf{f}(xs) \\ \mathsf{alt} &\to 0 : 1 : \mathsf{alt} & \mathsf{f}(1 : xs) &\to 1 : \mathsf{f}(xs) \\ \mathsf{zip}(x : xs, ys) &\to x : \mathsf{zip}(ys, xs) & & \end{aligned}$$

The tool first performs rewriting of the right-hand side of the p-rule and then proves context-sensitive termination using μ-Term [L$^+$]. Note the subtlety in this example: as soon as a ground term t can be composed of which the interpretation as a stream contains only finitely many ones, then the system will not be productive for $\mathsf{f}(t)$. So as a consequence it can be concluded that the paper folding stream p contains infinitely many ones, as the specification is productive for $\mathsf{f}(\mathsf{p})$.

6.3 Productivity of Non-Orthogonal Specifications

In the case of non-orthogonal proper specifications, two computations starting in the same initial term might end in (or, for infinite reductions, have as limit) two different constructor normal forms. This is natural when non-deterministic choices are allowed that guide the computation. In that case, it is therefore desired that all such choices lead to a constructor normal form. When considering this in the target application of hardware cells, for which stabilization shall be proven, then this corresponds to stabilization regardless of the concrete input values that are being supplied.

Hence, strong productivity should be studied for non-orthogonal specifications. It was already shown in Example 6.1.10 that strong and weak productivity differ for non-orthogonal specifications. In this section, the techniques for proving productivity that were presented in the previous Section 6.2 are extended to also be applicable to prove strong productivity of non-orthogonal proper specification. As a characterization of strong productivity, Proposition 6.1.12 will be used, which only requires the existence of a term starting with constructor in every maximal outermost-fair reduction sequence.

A first technique to prove strong productivity of proper specifications is given next. It is a simple syntactic check that determines whether every right-hand side of sort s starts with a constructor. For orthogonal proper specifications, this was already observed in Theorem 6.2.1.

Theorem 6.3.1. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be a proper specification. If for all rules $\ell \to r \in \mathcal{R}_s$, r is not a variable and $\mathrm{root}(r) \in \mathcal{C}$, then $\mathcal{S}$ is strongly productive.*

Proof. Let $\rho \equiv t_0 \to_{p_0} t_1 \to_{p_1} \dots$ be a maximal outermost-fair reduction and let $t_0 = f(u'_1, \dots, u'_m, t'_1, \dots, t'_n)$ with $\mathsf{ar}_d(f) = m$ and $\mathsf{ar}_s(f) = n$. If $f \in \mathcal{C}$ then nothing has to be proven, so assume $f \in \Sigma_s \setminus \mathcal{C}$. Structural induction on t_0 is performed to prove that $\mathrm{root}(t_k) \in \mathcal{C}$ for some $k \in \mathbb{N}$.

The induction hypothesis states that for every $1 \leq i \leq n$ and every maximal outermost-fair reduction $t'_i = t_{i,0} \to t_{i,1} \to \dots$ there exists an index $k_i \in \mathbb{N}$ such that $\mathrm{root}(t_{i,k_i}) \in \mathcal{C}$.

Assume that for all $l \in \mathbb{N}$, $p_l \neq \epsilon$. If t_0 contains a redex at position ϵ, this is an outermost redex that survives infinitely long, contradicting the assumption that ρ is outermost fair. Thus, $t_0 \not\to_\epsilon$. As in the proof of Proposition 6.1.12, define $P_r = \{p_l - r \mid l \in \mathbb{N}, p_l \geq r\}$ for $1 \leq r \leq n$ (i.e., the positions in the maximal outermost-fair reduction ρ that occur in argument r). Then, for $1 \leq r \leq n$ and $P_r = \{p^r_0, p^r_1, \dots\}$ the reduction $t'_r = t_{r,0} \to_{p^r_0} t_{r,1} \to_{p^r_1} \dots$ is also a maximal outermost-fair reduction, otherwise an infinitely long surviving outermost redex would also be an infinitely long surviving outermost redex of the reduction ρ. Therefore, indices $k_i \in \mathbb{N}$ exist such that $\mathrm{root}(t_{i,k_i}) \in \mathcal{C}$, due to the induction hypothesis. This makes reduction ρ have the shape $t_0 = f(u'_1, \dots, u'_m, t'_1, \dots, t'_n) \to^* f(u''_1, \dots, u''_m, t''_1, \dots, t''_n) = t_j$ for some $j \in \mathbb{N}$, where $u''_1, \dots, u''_m$ are normal forms (since the reduction ρ is maximal outermost-fair and $\mathcal{R}_d$ is terminating) and $t_{i,k_i} \to^* t''_i$, thus also $\mathrm{root}(t''_i) \in \mathcal{C}$. Because $\mathcal{R}_s$ is exhaustive, the term t_j must contain a redex at the root position ϵ, which of course is outermost. This gives rise to a contradiction to ρ being outermost fair, as this outermost redex survives infinitely often, because $p_l \neq \epsilon$ for all $l \in \mathbb{N}$.

Therefore, $p_{k-1} = \epsilon$ for some $k - 1 \in \mathbb{N}$ and the reduction ρ has the shape $t_0 \to^* t_{k-1} \to_\epsilon t_k = r\sigma$, where the last step is with respect to some rule $\ell \to r \in \mathcal{R}_s$. By the assumption on the shape of the rules in $\mathcal{R}_s$, $\mathrm{root}(r) \in \mathcal{C}$, hence also $\mathrm{root}(r\sigma) \in \mathcal{C}$, which proves productivity according to Proposition 6.1.12. □

This technique is sufficient to prove strong productivity of the proper specification consisting of the two rules for random in Example 6.1.10, since both have right-hand sides with the constructor : at the root. However, it is easy to create examples which are strongly productive, but do not satisfy the syntactic requirements of Theorem 6.3.1.

Example 6.3.2. Consider the proper specification with the following TRS $\mathcal{R}_s$:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{ones} & \to & 1 : \mathsf{ones} & \mathsf{finZeroes} & \to & 0 : \mathsf{ones} \\
\mathsf{finZeroes} & \to & 0 : 0 : \mathsf{ones} & \mathsf{finZeroes} & \to & 0 : 0 : 0 : \mathsf{ones} \\
\mathsf{f}(0 : xs) & \to & \mathsf{f}(xs) & \mathsf{f}(1 : xs) & \to & 1 : \mathsf{f}(xs)
\end{array}$$

The constant $\mathsf{finZeroes}$ produces non-deterministically a stream that starts with one, two, or three zeroes followed by an infinite stream of ones. Function f takes a binary stream as argument and filters out all occurrences of zeroes. Thus, productivity of this example proves that only a finite number of zeroes can be produced. This however cannot be proven with the technique of Theorem 6.3.1, since the right-hand side of the rule $\mathsf{f}(0 : xs) \to \mathsf{f}(xs)$ does not start with the constructor ':'.

Below, also the technique based on context-sensitive termination, which was presented in Section 6.2, is extended to the non-orthogonal setting. The idea is to disallow rewriting in structure arguments of constructors, thus context-sensitive termination implies that for every ground term of sort s, a term starting with a constructor can be reached (due to the exhaustiveness requirement). As was observed by Endrullis and Hendriks recently in [EH11], this set of blocked positions can be enlarged, making the approach even stronger.

Below, the extended technique for proving productivity by showing termination of a corresponding context-sensitive TRS is given for the more general proper specifications, which also may include non-determinism. This version already includes an adaption of the improvement mentioned above.

Definition 6.3.3. Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be a proper specification. The replacement map $\mu_\mathcal{S} : \Sigma_d \cup \Sigma_s \to 2^\mathbb{N}$ is defined as follows: [4]

- $\mu_\mathcal{S}(f) = \{1, \dots, \mathsf{ar}_d(f)\}$, if $f \in \Sigma_d \cup \mathcal{C}$
- $\mu_\mathcal{S}(f) = \{1, \dots, \mathsf{ar}_d(f) + \mathsf{ar}_s(f)\} \setminus \{1 \leq i \leq \mathsf{ar}_d(f) + \mathsf{ar}_s(f) \mid t|_i$ is a variable for all $\ell \to r \in \mathcal{R}_s$ and all non-variable subterms t of ℓ with $\mathrm{root}(t) = f\}$,[5] otherwise

In the remainder, the subscript $\mathcal{S}$ is left out if the specification is clear from the context. The replacement map $\mu_\mathcal{S}$ is *canonical* [Luc02] for the left-linear TRS $\mathcal{R}_s$, guaranteeing that non-variable positions of left-hand sides are allowed. In the above

[4] Note that in [EH11], Endrullis and Hendriks consider orthogonal TRSs and also block arguments of symbols in Σ_d which only contain variables. This however is problematic when allowing data rules that are not left-linear. Example:

$$\begin{array}{lrcl@{\qquad}rcl@{\qquad}rcl}
\mathcal{R}_s : & \mathsf{f}(1) & \to & \mathsf{f}(\mathsf{d}(0, \mathsf{d}(1, 0))) & \mathsf{f}(0) & \to & \mathsf{c} \in \mathcal{C} & & & \\
\mathcal{R}_d : & \mathsf{d}(x, x) & \to & 1 & \mathsf{d}(0, x) & \to & 0 & \mathsf{d}(1, x) & \to & 0
\end{array}$$

Here, the term $\mathsf{f}(\mathsf{d}(0, \mathsf{d}(1, 0)))$ can only be μ-rewritten to the term $\mathsf{f}(0)$ (which then in turn has to be rewritten to c) if defining $\mu(\mathsf{d}) = \{1\}$, since the subterm $\mathsf{d}(1, 0)$ can never be rewritten to 0. However, the example is not strongly productive, as reducing in this way gives rise to an infinite outermost-fair reduction $\mathsf{f}(\mathsf{d}(0, \mathsf{d}(1, 0))) \to \mathsf{f}(\mathsf{d}(0, 0)) \to \mathsf{f}(1) \to \dots$. Blocking arguments of data symbols can only be done when $\mathcal{R}_d$ is left-linear, too.

[5] The requirement of t not being a variable ensures that $\mathrm{root}(t)$ is defined.

definition, it is extended to the possibly non-left-linear TRS $\mathcal{R}_d \cup \mathcal{R}_s$ by allowing all arguments of symbols from Σ_d.

The main result of this section is that also for possibly non-orthogonal proper specifications, μ-termination implies productivity.

Theorem 6.3.4. *A proper specification $\mathcal{S} = (\Sigma_d,\ \Sigma_s,\ \mathcal{C},\ \mathcal{R}_d,\ \mathcal{R}_s)$ is strongly productive, if $\mathcal{R}_d \cup \mathcal{R}_s$ is $\mu_{\mathcal{S}}$-terminating.*

Before proving the above theorem, it will first be shown that it subsumes Theorem 6.3.1. Intuitively, this holds because structure arguments of constructors are blocked, and if every right-hand side of $\mathcal{R}_s$ starts with a constructor then the number of allowed redexes of sort s in a term steadily decreases. This is akin to the argument in Proposition 6.2.8, which showed the corresponding result for orthogonal proper specifications.

Proposition 6.3.5. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be a proper specification. If for all rules $\ell \to r \in \mathcal{R}_s$ it holds that* $\operatorname{root}(r) \in \mathcal{C}$*, then $\mathcal{R}_d \cup \mathcal{R}_s$ is $\mu_{\mathcal{S}}$-terminating.*

Proof. Let $t \in \mathcal{T}(\Sigma_d \cup \Sigma_s, \mathcal{V})$ be well-typed. If t has sort d, then all subterms must also be of sort d, as symbols from Σ_d only have arguments of that sort. Hence, rewriting can only be done with rules from $\mathcal{R}_d$, which is assumed to be terminating.

Otherwise, let t be of sort s and assume that t starts an infinite μ-reduction $t = t_0 \overset{\mu}{\to}_{\ell_0 \to r_0, p_0} t_1 \overset{\mu}{\to}_{\ell_1 \to r_1, p_1} t_2 \overset{\mu}{\to}_{\ell_2 \to r_2, p_2} \ldots$. Define $\operatorname{Pos}_\mu^{\mathrm{red}_s}(t') = \{p \in \operatorname{Pos}_\mu(t') \mid t'|_p \text{ is a redex of sort } s\}$ for any term $t' \in \mathcal{T}(\Sigma_d \cup \Sigma_s, \mathcal{V})$. It will be proven that in every step $t_i \overset{\mu}{\to}_{\ell_i \to r_i, p_i} t_{i+1}$ of the infinite reduction, $|\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_{i+1})| \leq |\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i)|$ and that for steps with $\ell_i \to r_i \in \mathcal{R}_s$, it even holds that $|\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_{i+1})| < |\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i)|$. To this end, case analysis of the rule $\ell_i \to r_i$ is performed. If $\ell_i \to r_i \in \mathcal{R}_d$, then $t_i = t_i[\ell_i\sigma_i]_{p_i}$ and $t_{i+1} = t_i[r_i\sigma_i]_{p_i}$ for some substitution σ_i. Because $\ell_i, r_i \in \mathcal{T}(\Sigma_d, \mathcal{V})$, $|\operatorname{Pos}_\mu^{\mathrm{red}_s}(\ell_i)| = |\operatorname{Pos}_\mu^{\mathrm{red}_s}(r_i)| = 0$. Also, for all $x \in \mathcal{V}$, $\sigma_i(x) \in \mathcal{T}(\Sigma_d, \mathcal{V})$ since all symbols in Σ_d have arguments of sort d, and $\operatorname{root}(\ell_i) \in \Sigma_d$. Thus, $\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_{i+1}) = \operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i)$. In the second case, $\ell_i \to r_i \in \mathcal{R}_s$. Let $t_i = t_i[\ell_i\sigma_i]_{p_i}$ and $t_{i+1} = t_i[r_i\sigma_i]_{p_i}$ for some substitution σ_i. Then, $\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i) = \operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i[z]_{p_i}) \uplus \{p_i.p \mid p \in \operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i|_{p_i})\}$ for any variable $z \in \mathcal{V}$ of sort s. For t_{i+1}, it is the case that $\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_{i+1}) = \operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i[r_i\sigma_i]_{p_i}) = \operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i[z]_{p_i}) \uplus \{p_i.p \mid p \in \operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i[r_i\sigma_i]_{p_i}|_{p_i})\}$ for any variable $z \in \mathcal{V}$ of sort s. Here, it holds that $\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i|_{p_i}) = \operatorname{Pos}_\mu^{\mathrm{red}_s}(\ell_i\sigma_i) \ni \epsilon$, therefore $p_i \in \operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i)$. Furthermore, $\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i[r_i\sigma_i]_{p_i}|_{p_i}) = \operatorname{Pos}_\mu^{\mathrm{red}_s}(r_i\sigma_i) = \emptyset$, since $\operatorname{root}(r_i) \in \mathcal{C}$ by assumption, hence $\mu(\operatorname{root}(r_i)) = \{1, \ldots, \mathsf{ar}_d(\operatorname{root}(r_i))\}$ and because symbols from Σ_d only have arguments of sort d. Thus, $\operatorname{Pos}_\mu^{\mathrm{red}_s}(t_{i+1}) \subsetneq \operatorname{Pos}_\mu^{\mathrm{red}_s}(t_i)$.

Combining these observations, only finitely many reductions with rules from $\mathcal{R}_s$ exist in the infinite reduction. Thus, an infinite tail of steps with rules from $\mathcal{R}_d$ exists. This however contradicts the assumption that $\mathcal{R}_d$ is terminating, hence no infinite μ-reduction can exist which proves μ-termination of $\mathcal{R}_d \cup \mathcal{R}_s$. ☐

Hence, analyzing context-sensitive termination only would be sufficient. However, the syntactic check of Theorem 6.3.1 can be done very fast and should therefore be the first method to try.

In order to prove Theorem 6.3.4 the following lemma is needed, which shows that in every ground term not starting with a constructor there exists a redex that is not blocked by the replacement map μ.

Lemma 6.3.6. *Let $\mathcal{S}$ be a proper specification. For all ground terms t of sort s with $\mathrm{root}(t) \notin \mathcal{C}$ there exists a position $p \in \mathrm{Pos}_\mu(t)$ such that $t \to_p$.*

Proof. Let $t = f(u_1, \dots, u_m, t_1, \dots, t_n)$. Structural induction on t is performed. If $u_i \to_{p'}$ for some $1 \le i \le m$ with $i \in \mu(f)$, then $t \to_{i.p'}$ and $i.p' \in \mathrm{Pos}_\mu(t)$ since arguments of data symbols are never blocked. Thus, assume in the remainder that u_i is a ground normal form w.r.t. $\mathcal{R}_d$ for all $1 \le i \le m$ with $i \in \mu(f)$. If $\mathrm{root}(t_i) \in \mathcal{C}$ for all $1 \le i \le n$ such that $m + i \in \mu(f)$, then either there exists a $1 \le j \le \mathsf{ar}_d(\mathrm{root}(t_i))$ and position $p \in \mathrm{Pos}(t_i|_j)$ such that $t_i|_j \to_{\mathcal{R}_d}$, or all data arguments $t_i|_j$ are ground normal forms. In the first case, the position $i.j.p$ is an allowed position, which proves the lemma. Otherwise, in case all data arguments are ground normal forms, $t \to_\epsilon$ must hold by the exhaustiveness requirement (and because all arguments u_j, t_j with $j \notin \mu(f)$ are being matched by pairwise different variables, due to left-linearity). Finally, the case has to be considered where a $1 \le i \le n$ exists with $m + i \in \mu(f)$ such that $\mathrm{root}(t_i) \notin \mathcal{C}$. By the induction hypothesis, $t_i \to_{p'}$ for some $p' \in \mathrm{Pos}_\mu(t_i)$. Therefore, $(m+i).p' \in \mathrm{Pos}_\mu(t)$ and $t \to_{m+i.p'}$ hold. □

A second lemma that is required for the proof of Theorem 6.3.4 states that a specialized version of the Parallel Moves Lemma [BN98] holds for proper specifications. This lemma allows to swap the order of reductions blocked by μ with reductions not blocked by μ. To formulate the lemma, the notion of a parallel reduction step $t \overset{\shortparallel}{\longrightarrow}_P t'$ is needed, which is defined for a set $P = \{p_1, \dots, p_n\} \subseteq \mathrm{Pos}(t)$ where for every pair $1 \le i < j \le n$ it holds that $p_i \parallel p_j$ and a term $t = t[\ell_1\sigma_1]_{p_1} \dots [\ell_n\sigma_n]_{p_n}$ as $t' = t[r_1\sigma_1]_{p_1} \dots [r_n\sigma_n]_{p_n}$ for rules $\ell_i \to r_i \in \mathcal{R}_d \cup \mathcal{R}_s$ and substitutions σ_i, $1 \le i \le n$.

Lemma 6.3.7. *Let $\mathcal{S}$ be a proper specification. For all ground terms t, t', t'' and positions $P \subseteq \mathrm{blocked}_\mu(t)$, $p \in \mathrm{Pos}_\mu(t')$ with $t \overset{\shortparallel}{\longrightarrow}_P t' \to_{\ell \to r, p} t''$, a term $\hat{t}$ and a set $P' \subseteq \mathrm{Pos}(\hat{t})$ exist such that $t \to_{\ell \to r, p} \hat{t} \overset{\shortparallel}{\longrightarrow}_{P'} t''$.*

Proof. Let $P = \{p_1, \dots, p_k\} \subseteq \mathrm{blocked}_\mu(t)$. Then $t = t[\ell_1\sigma_1]_{p_1} \dots [\ell_k\sigma_k]_{p_k} \overset{\shortparallel}{\longrightarrow}_P t[r_1\sigma_1]_{p_1} \dots [r_k\sigma_k]_{p_k} = t' = t'[\ell\sigma]_p$ for some rules $\ell_1 \to r_1, \dots, \ell_k \to r_k, \ell \to r \in \mathcal{R}_d \cup \mathcal{R}_s$ and substitutions $\sigma_1, \dots, \sigma_k, \sigma$. W.l.o.g., let $0 \le j \le k$ be such that $p_i \not\parallel p$ for all $1 \le i \le j$ and $p_i \parallel p$ for all $j < i \le k$. Since $p \in \mathrm{Pos}_\mu(t')$ and $p_i \in \mathrm{blocked}_\mu(t')$, it must hold that $p < p_i$ for all $1 \le i \le j$. Therefore, the term t' must have the following shape:

$$t' = t\left[\ell\sigma[r_1\sigma_1]_{p_1-p} \dots [r_j\sigma_j]_{p_j-p}\right]_p [r_{j+1}\sigma_{j+1}]_{p_{j+1}} \dots [r_k\sigma_k]_{p_k}$$

If $\ell \to r \in \mathcal{R}_d$, then it must hold that $j = 0$, since arguments of data symbols are never blocked. Hence, the lemma trivially holds in this case, as all reductions are on independent positions.

Otherwise, $\ell \to r \in \mathcal{R}_s$. Because the positions p_i for $1 \le i \le j$ are blocked, it must be the case that they are either below a variable in all rules containing a certain symbol f (hence, they are also below a variable in ℓ), or they are below a structure argument of a constructor $c \in \mathcal{C}$. By requirement of specifications, if a constructor is present on a left-hand side of a rule, all its structure arguments must be variables. Thus, it can be concluded that all positions p_i, and thereby all terms $r_i\sigma_i$, are below some variable of ℓ in t'. Additionally, the left-hand side ℓ is required to be linear, therefore there exist pairwise different variables $x_1, \dots, x_j$, contexts $C_1, \dots, C_j$, and a substitution σ' being like σ except that $\sigma'(x_i) = x_i$ for $1 \leq i \leq j$ such that:

$$\begin{aligned} t' = {} & t\left[\ell\sigma'\{x_1{:=}C_1[r_1\sigma_1],\ldots,x_j{:=}C_j[r_j\sigma_j]\}\right]_p\ [r_{j+1}\sigma_{j+1}]_{p_{j+1}}\cdots[r_k\sigma_k]_{p_k} \\ \rightarrow_p {} & t\left[r\sigma'\{x_1{:=}C_1[r_1\sigma_1],\ldots,x_j{:=}C_j[r_j\sigma_j]\}\right]_p\ [r_{j+1}\sigma_{j+1}]_{p_{j+1}}\cdots[r_k\sigma_k]_{p_k} = t'' \end{aligned}$$

It can be concluded that $p \in \mathrm{Pos}_\mu(t)$, as all reduction steps in $t \overset{\shortparallel}{\rightarrow}_P t'$ are either below or independent of p. Thus:

$$\begin{aligned} t = {} & t\left[\ell\sigma'\{x_1{:=}C_1[\ell_1\sigma_1],\ldots,x_j{:=}C_j[\ell_j\sigma_j]\}\right]_p\ [\ell_{j+1}\sigma_{j+1}]_{p_{j+1}}\cdots[\ell_k\sigma_k]_{p_k} \\ \rightarrow_p {} & t\left[r\sigma'\{x_1{:=}C_1[\ell_1\sigma_1],\ldots,x_j{:=}C_j[\ell_j\sigma_j]\}\right]_p\ [\ell_{j+1}\sigma_{j+1}]_{p_{j+1}}\cdots[\ell_k\sigma_k]_{p_k} = \hat{t} \\ \overset{\shortparallel}{\rightarrow}_{P'} {} & t\left[r\sigma'\{x_1{:=}C_1[r_1\sigma_1],\ldots,x_j{:=}C_j[r_j\sigma_j]\}\right]_p\ [r_{j+1}\sigma_{j+1}]_{p_{j+1}}\cdots[r_k\sigma_k]_{p_k} = t'' \end{aligned}$$

In the second reduction step, the positions of the terms $\ell_i\sigma_i$ in $\hat{t}$ constitute the set $P' \subseteq \mathrm{Pos}(\hat{t})$. □

The two above lemmas allow to prove the main Theorem 6.3.4, showing that context-sensitive termination implies productivity of the considered proper specification.

Proof of Theorem 6.3.4. Assume $\mathcal{S}$ is not strongly productive. Then, a maximal outermost-fair reduction sequence $\rho \equiv t_0 \rightarrow t_1 \rightarrow \ldots$ exists where for all $k \in \mathbb{N}$, $\mathrm{root}(t_k) \notin \mathcal{C}$.

This reduction sequence is infinite, since otherwise it would end in a term t_m for some $m \in \mathbb{N}$ with $\mathrm{root}(t_m) \notin \mathcal{C}$. Then however, according to Lemma 6.3.6, the term t_m would contain a redex, giving a contradiction to the sequence being maximal.

The sequence might however perform reductions that are below a variable argument of a constructor or below a variable in all left-hand sides of a defined symbol. These reduction steps are not allowed when considering context-sensitive rewriting with respect to μ. Such reductions however can be reordered. First, due to Lemma 6.3.6, there is always a redex which is not blocked, thus there is also an outermost such one. Because the reduction is outermost-fair, and because reductions below a variable cannot change the matching of a rule, as shown in Lemma 6.3.7, such redexes must be contracted an infinite number of times in the infinite reduction sequence ρ. Thus, the reduction steps in ρ can be reordered: If there is a (parallel) reduction below a variable before performing a step that is allowed by μ, then these two steps are swapped using Lemma 6.3.7. Repeating this construction yields an infinite reduction sequence ρ' consisting of steps which are not blocked by μ. Thus, this is an infinite μ-reduction sequence, showing that $\mathcal{R}_d \cup \mathcal{R}_s$ is not μ-terminating, which therefore proves the theorem. □

The technique of Theorem 6.3.4, i.e., proving μ-termination of the corresponding context-sensitive TRS, is able to prove strong productivity of Example 6.3.2. By Definition 6.3.3, the corresponding replacement map μ is defined as $\mu(\mathsf{0}) = \mu(\mathsf{1}) = \mu(\mathsf{ones}) = \mu(\mathsf{finZeroes}) = \emptyset$ and $\mu(\mathsf{f}) = \mu(:) = \{1\}$, i.e., rewriting is allowed on all positions except those that are inside a second argument of the constructor :. Context-sensitive termination of the TRS together with the above replacement map μ can for example be shown by the tool AProVE [GSKT06]. Thus, productivity of that example has been shown according to Theorem 6.3.4.

Another example, which illustrates the improvement of [EH11] incorporated in Definition 6.3.3 that blocks more argument positions, is given below.

Example 6.3.8. Consider the following proper specification, given by the TRS $\mathcal{R}_s$:

$$\mathsf{a} \to \mathsf{f}(1 : \mathsf{a}, \mathsf{a}) \qquad \begin{aligned} \mathsf{f}(x : xs, ys) &\to x : ys \\ \mathsf{f}(\mathsf{f}(xs, ys), zs) &\to \mathsf{f}(xs, \mathsf{f}(ys, zs)) \end{aligned}$$

When defining $\mu(1) = \mu(\mathsf{a}) = \emptyset$ and $\mu(:) = \{1\}$ by the first case of Definition 6.3.3, and defining $\mu(\mathsf{f}) = \{1, 2\}$ (i.e., not removing any argument positions, as was done in the orthogonal case in Definition 6.2.3), then an infinite μ-reduction exists:

$$\mathsf{a} \xrightarrow{\mu} \mathsf{f}(1 : \mathsf{a}, \mathsf{a}) \xrightarrow{\mu} \mathsf{f}(1 : \mathsf{a}, \mathsf{f}(1 : \mathsf{a}, \underline{\mathsf{a}})) \xrightarrow{\mu} \ldots$$

This reduction can be continued in the above style by reducing the underlined redex further, which will always create the term a on an allowed position of the form 2^n. However, such positions are not required for any of the f-rules to be applicable; for both rules it holds that all subterms of left-hand sides that start with the symbol f, which are the terms $\mathsf{f}(x : xs, ys)$, $\mathsf{f}(\mathsf{f}(xs, ys), zs)$, and $\mathsf{f}(xs, ys)$, have a variable as second argument. Thus, according to Definition 6.3.3, the replacement map μ' can be defined to be like μ, except that $\mu'(\mathsf{f}) = \{1\}$. With this improved replacement map, μ'-termination of the above TRS can for example be proven by the tool AProVE [GSKT06], which implies productivity by Theorem 6.3.4.

Checking productivity in this way, i.e., by checking context-sensitive termination, can only prove productivity but not disprove it. This is illustrated in the next example.

Example 6.3.9. Consider the proper specification with the following rules in $\mathcal{R}_s$:

$$\mathsf{a} \to \mathsf{f}(\mathsf{a}) \qquad \mathsf{f}(x : xs) \to x : \mathsf{f}(xs) \qquad \mathsf{f}(xs) \to 1 : xs$$

Starting in the term a, an infinite μ-reduction starting with $\mathsf{a} \to \mathsf{f}(\underline{\mathsf{a}})$ exists, which can be continued by reducing the underlined redex repeatedly, since $\mu(\mathsf{f}) = \{1\}$. Thus, the example is not μ-terminating. However, the specification is strongly productive, as can be shown by a case distinction based on the root symbol of some arbitrary ground term t. In case $\mathrm{root}(t) = :$, then nothing has to be done, according to Proposition 6.1.12. Otherwise, if $\mathrm{root}(t) = \mathsf{a}$, then any maximal outermost-fair reduction must start with $t = \mathsf{a} \to \mathsf{f}(\mathsf{a})$, thus the analysis can be reduced to the final case, where $\mathrm{root}(t) = \mathsf{f}$. In this last case, $t = \mathsf{f}(t')$. When rewriting the term t', i.e., performing reductions below the root, then the term still has the shape $\mathsf{f}(\tilde{t})$ for some term $\tilde{t}$ such that $t' \to^* \tilde{t}$. Hence, any such term is a redex with respect to the third rule and therefore must be eventually reduced in an outermost-fair reduction. Either the second or the third rule must be applied to such a term $\mathsf{f}(\tilde{t})$, both of which create a constructor : at the root. Therefore, also in this case any outermost-fair reduction reaches a term having a constructor as root symbol, which proves strong productivity due to Proposition 6.1.12.

In the remainder of this section it shall be illustrated that the requirements of proper specifications in Definition 6.1.5, namely that the TRS $\mathcal{R}_s$ should be left-linear and that structure arguments of constructors in left-hand sides must not be structure symbols, i.e., that they must be variables, are required for soundness of Theorem 6.3.4. The first example specification is not left-linear and not strongly productive, but μ-terminating.

Example 6.3.10. Consider the non-proper specification containing the following rules in $\mathcal{R}_s$, where $\mathcal{C} = \{\mathsf{a}, \mathsf{c}\}$:

$$\begin{array}{rclcrcl}
\mathsf{b} & \to & \mathsf{a} & \qquad & \mathsf{f}(\mathsf{c}(x,x)) & \to & \mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{b})) \\
\mathsf{f}(\mathsf{a}) & \to & \mathsf{a} & \qquad & \mathsf{f}(\mathsf{c}(x,y)) & \to & \mathsf{c}(x,y)
\end{array}$$

The example specification is not strongly productive, as it admits the infinite outermost-fair reduction sequence $\mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{a})) \to \mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{b})) \to \mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{a})) \to \dots$. However, the TRS can be proven μ-terminating by the tool AProVE [GSKT06], where $\mu(\mathsf{f}) = \{1\}$ and $\mu(\mathsf{a}) = \mu(\mathsf{b}) = \mu(\mathsf{c}) = \emptyset$. This is the case because rewriting below the constructor c is not allowed, thus the second step of the above reduction sequence is blocked. The reason why Theorem 6.3.4 fails is the reordering of reductions, since in this example a reduction of the form $t \overset{\shortparallel}{\rightarrow}_P t' \rightarrow_{\ell\to r,p} t''$ (here: $\mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{b})) \overset{\shortparallel}{\rightarrow}_{\{1.1\}} \mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{a})) \rightarrow_{\mathsf{f}(\mathsf{c}(x,x))\to\mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{b})),\epsilon} \mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{b})))$ does not imply that $t \rightarrow_{\ell\to r,p}$ (in the example, $\mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{b})) \not\rightarrow_{\mathsf{f}(\mathsf{c}(x,x))\to\mathsf{f}(\mathsf{c}(\mathsf{a},\mathsf{b})),\epsilon}$), i.e., Lemma 6.3.7 does not hold.

The next example illustrates why non-variable structure arguments of constructors are not allowed in left-hand sides.

Example 6.3.11. Let $\mathcal{R}_s$ contain the following rules, and let $\mathcal{C} = \{\mathsf{a}, \mathsf{c}\}$.

$$\begin{array}{rclcrcl}
\mathsf{b} & \to & \mathsf{a} & \qquad & \mathsf{f}(\mathsf{c}(\mathsf{a})) & \to & \mathsf{f}(\mathsf{c}(\mathsf{b})) \\
\mathsf{f}(\mathsf{a}) & \to & \mathsf{a} & \qquad & \mathsf{f}(\mathsf{c}(x)) & \to & x
\end{array}$$

The TRS $\mathcal{R}_s$ is context-sensitive terminating, but the corresponding specification is not strongly productive due to the infinite outermost-fair reduction sequence $\mathsf{f}(\mathsf{c}(\mathsf{a})) \to \mathsf{f}(\mathsf{c}(\mathsf{b})) \to \mathsf{f}(\mathsf{c}(\mathsf{a})) \to \dots$. Here, the second step is not allowed when performing context-sensitive rewriting, since $\mu(\mathsf{c}) = \emptyset$. Using the tool AProVE [GSKT06], context-sensitive termination of the above TRS together with the replacement map μ can be shown.

However, this example can be unrolled, which makes the resulting specification proper, by introducing a fresh symbol g and replacing the two rules for f in the right column above by the following three rules:

$$\mathsf{f}(\mathsf{c}(x)) \to \mathsf{g}(x) \qquad\qquad \mathsf{g}(\mathsf{a}) \to \mathsf{f}(\mathsf{c}(\mathsf{b})) \qquad\qquad \mathsf{g}(x) \to x$$

Then in the corresponding context-sensitive TRS, $\mu(\mathsf{f}) = \mu(\mathsf{g}) = \{1\}$ and $\mu(\mathsf{a}) = \mu(\mathsf{b}) = \mu(\mathsf{c}) = \emptyset$. This context-sensitive TRS is not μ-terminating, since it admits the infinite reduction $\mathsf{f}(\mathsf{c}(\mathsf{a})) \overset{\mu}{\to} \mathsf{g}(\mathsf{a}) \overset{\mu}{\to} \mathsf{f}(\mathsf{c}(\mathsf{b})) \overset{\mu}{\to} \mathsf{g}(\mathsf{b}) \overset{\mu}{\to} \mathsf{g}(\mathsf{a}) \overset{\mu}{\to} \dots$.

It should be noted that the restriction for left-hand sides to only contain variables in constructor arguments was already made in [ZR10a]. This is the case because matching constructors nested within constructors would otherwise invalidate the approach of disallowing rewriting inside structure arguments of constructors.

6.4 Proving Productivity of Hardware Cells

Proving productivity can be used to verify stabilization of Hardware circuits. In such a circuit, the inputs can be seen as an infinite stream of zeroes and ones, which in general can occur in any arbitrary sequence. Furthermore, a circuit contains a number of internal signals, which also carry different Boolean values over time. To

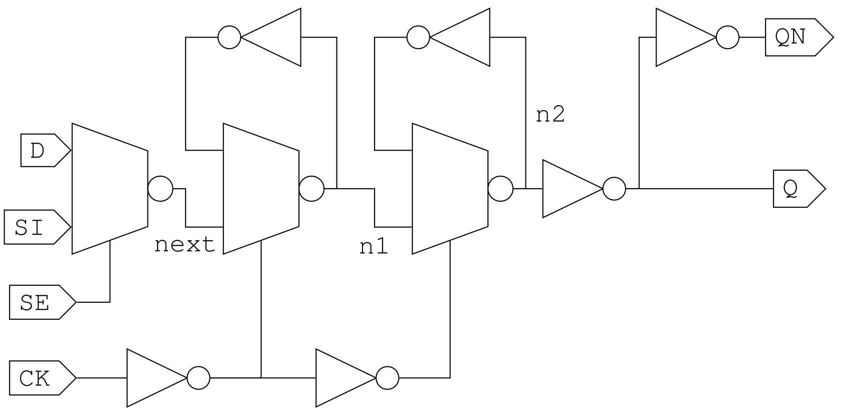


Figure 6.1: Example circuit of scanable D flip-flop

store a value over time, feedback loops are used. In such a loop, a value that is computed from some logic function is also used as an input to that function. Thus, it is desired that such values stabilize, instead of oscillating indefinitely.

To check this, productivity analysis can be used. This will be illustrated in this section by means of an example that will be considered throughout the rest of this section.

Consider the circuit shown in Figure 6.1, implementing a scanable D flip-flop. This circuit first selects, based on the value of the input `SE` (scan enable), either the negation of the data input `D` (in case `SE`=0) or the negation of the scan data input `SI` (in case `SE`=1). This value, called `next` in Figure 6.1, is then fed into another multiplexer (mux), for which a feedback loop exists. This mux is controlled by the negation of the clock input `CK`. If the clock is 0 then the negated value of `next` is forwarded to the output `n1`, otherwise the stored value of `n1` is kept. Similarly, `n2` implements such a latch structure, however this time the latch forwards the negation of the `n1` input in case `CK` is 1, and it keeps its value when `CK` is 0. The outputs `Q` and `QN` are computed from this stored value `n2`.

Note that a lot of the negations are only contained to refresh the signals, otherwise a high voltage value might decay and not be detected properly anymore.

From the example circuit, a proper specification can be created, where the data symbols consist of the two Boolean values 0 and 1 and the symbol not used for negating by means of the following two rules:

$$\mathsf{not}(0) \to 1 \qquad\qquad \mathsf{not}(1) \to 0$$

The structures of interest are the infinite streams containing Boolean values, thus the set of constructors is $\mathcal{C} = \{:\}$. The structure TRS $\mathcal{R}_s$ contains the rules shown in Figure 6.2. It should be remarked that in the these rules, the negations of the clock have been removed, to ease readability. The defined function symbols next, n1, n2, q, and qn reflect the wires and output signals with the corresponding name in Figure 6.1. The constant rand is added to abstract the values of the inputs. It provides a random stream of Boolean values, thus it is able to represent any sequence of input values provided to the circuit. The rules of the symbol next implement the

$$
\begin{array}{rcl}
\mathsf{rand} & \to & 0 : \mathsf{rand} \\
\mathsf{rand} & \to & 1 : \mathsf{rand} \\
\\
\mathsf{next}(0 : ses, d : ds, si : sis) & \to & \mathsf{not}(d) : \mathsf{next}(ses, ds, sis) \\
\mathsf{next}(1 : ses, d : ds, si : sis) & \to & \mathsf{not}(si) : \mathsf{next}(ses, ds, sis) \\
\\
\mathsf{n1}(0 : cks, nextv : nexts, n1l) & \to & \mathsf{not}(nextv) : \mathsf{n1}(cks, nexts, \mathsf{not}(nextv)) \\
\mathsf{n1}(1 : cks, nextv : nexts, n1l) & \to & \mathsf{n1}'(cks, nexts, n1l, \mathsf{not}(\mathsf{not}(n1l))) \\
\mathsf{n1}'(cks, nexts, 0, 0) & \to & 0 : \mathsf{n1}(cks, nexts, 0) \\
\mathsf{n1}'(cks, nexts, 1, 1) & \to & 1 : \mathsf{n1}(cks, nexts, 1) \\
\mathsf{n1}'(cks, nexts, 0, 1) & \to & \mathsf{n1}'(cks, nexts, 1, \mathsf{not}(\mathsf{not}(1))) \\
\mathsf{n1}'(cks, nexts, 1, 0) & \to & \mathsf{n1}'(cks, nexts, 0, \mathsf{not}(\mathsf{not}(0))) \\
\\
\mathsf{n2}(0 : cks, n1v : n1s, n2l) & \to & \mathsf{n2}'(cks, n1s, n2l, \mathsf{not}(\mathsf{not}(n2l))) \\
\mathsf{n2}(1 : cks, n1v : n1s, n2l) & \to & \mathsf{not}(n1v) : \mathsf{n2}(cks, n1s, \mathsf{not}(n1v))) \\
\mathsf{n2}'(cks, n1s, 0, 0) & \to & 0 : \mathsf{n2}(cks, n1s, 0) \\
\mathsf{n2}'(cks, n1s, 1, 1) & \to & 1 : \mathsf{n2}(cks, n1s, 1) \\
\mathsf{n2}'(cks, n1s, 0, 1) & \to & \mathsf{n2}'(cks, n1s, 1, \mathsf{not}(\mathsf{not}(1))) \\
\mathsf{n2}'(cks, n1s, 1, 0) & \to & \mathsf{n2}'(cks, n1s, 0, \mathsf{not}(\mathsf{not}(0))) \\
\\
\mathsf{q}(n2v : n2s) & \to & \mathsf{not}(n2v) : \mathsf{q}(n2s) \\
\\
\mathsf{qn}(qv : qs) & \to & \mathsf{not}(qv) : \mathsf{qn}(qs)
\end{array}
$$

Figure 6.2: Structure TRS $\mathcal{R}_s$ for the circuit in Figure 6.1

mux selecting either the next data input value d in case the next scan enable input value si is 0, or the next scan input value si in case si is 1.

The output of n1 is also computed by a mux, however, here the previous output value has to be considered due to the feedback loop. This cycle is broken by introducing a new parameter *n1l* that stores the previously output value. Then, based on the next value of the clock ck, the input stream *nextv* : *nexts* coming from the previously described multiplexer, and from the previous output value the next value of the stream at n1 is computed. If the clock ck is 0, then the latch simply outputs the value *nextv* and continues on the remaining streams, setting the parameter *n1l* to this value to remember the computed value. Otherwise, if ck is 1, then the feedback loop is active and has to be evaluated until it stabilizes. This is done by the function n1′. It has as arguments the remaining input stream of the clock, the remaining input stream of the scan multiplexer, and the previous output value and the newly computed output value. If both of these values are the same, then the value of the wire n1 has stabilized and hence can be output. The tail of the output stream is computed by again calling the function n1 with the remaining streams for the clock and the scan multiplexer. Otherwise, the new output value (the last argument of n1′) differs from the old output value (the penultimate argument of n1′). In that case, the new output value becomes the old output value and the new output is recomputed.

This is repeated until eventually the output value stabilizes, or it will oscillate and never produce a stable output.

Similar to the function n1, the function n2 computes stable values for the corresponding wire in Figure 6.1. Again, the parameter *n2l* is added to store a previously output value, and the auxiliary function n2′ is used to compute a stable value for the feedback loop. The only difference to the function n1 is that the cases of the clock are inverted, due to the additional inverter in Figure 6.1 that feeds the select input of the multiplexer that computes n2.

Finally, the functions q and qn implement the two inverters that feed the corresponding output signals in Figure 6.1.

The above specification is productive, since the TRS $\mathcal{R}_d \cup \mathcal{R}_s$ can be proven context-sensitive terminating, for example by the tool AProVE [GSKT06]. Hence, according to Theorem 6.3.4, the specification is productive, meaning that every ground term of sort s rewrites to a constructor term. This especially holds for the ground terms $t_{\mathsf{q}} = \mathsf{q}(t_{\mathsf{n2}})$ and $t_{\mathsf{qn}} = \mathsf{qn}(t_{\mathsf{q}})$, where the ground term t_{n2} is an arbitrary instantiation of the term $\hat{t}_{\mathsf{n2}} = \mathsf{n2}(\mathsf{rand}, \mathsf{n1}(\mathsf{rand}, \mathsf{nexts}(\mathsf{rand}, \mathsf{rand}, \mathsf{rand}), n1l), n2l)$, i.e., the variables *n1l* and *n2l* are instantiated arbitrarily with either 0 or 1. Thus, the circuit produces an infinite stream of stable output values, regardless of its initial state and input streams, and does not oscillate infinitely long.

A similar question exists in the context of the synchronous language Esterel [Ber99, PBEB07], an imperative language that is often used to describe embedded systems. In this language, so-called *reactions* are described, which are responses to certain input signals that take zero time. In such a response, a signal can either be present or absent, but not both. An Esterel program is then said to be *constructive*, if from the current state, the set of signals that is present or absent can be uniquely computed for any valuation of the input signals. This is similar to the productivity question investigated here. When viewing an Esterel program as producing a stream of sets of signals that have been emitted in the current reaction, then productivity of the program proves that in any state and for any input valuation the program does compute a well-defined set of emitted signals. However, in contrast to the approach presented in this section, formal semantics of Esterel such as [Ber99, PBEB07, TdS05] treat a situation in which a signal cannot be computed to be either emitted or non-emitted as deadlock, whereas the productivity analysis requires such situations to behave as a livelock, i.e., a situation where further steps are possible but do not produce any output (next stream element). Another semantics of Esterel that is specifically geared towards checking constructiveness is presented in [Mou09]. There, constructiveness of a subset of the Esterel language is defined as the property that a proof tree is well-founded, which in turn means that a non-constructive program gives rise to an infinite proof tree. It would be interesting to investigate whether this semantics can be used to formulate the question of constructiveness as a productivity problem. If this is the case, then one should also look into extending the semantics to the full Esterel language.

Productivity is also related to the Kahn principle of Kahn Process Networks (KPNs) [Kah74, GB10], which are commonly used in the streaming media domain to model computations at an abstract level. A KPN describes a number of parallel processes that communicate via unbounded fifo buffers. For KPNs, the Kahn principle states that the operational semantics and the denotational semantics coincide, i.e., the intended behavior can indeed be achieved by executing the operational rules. This is also the case for productivity, where executing the rules creates the desired object, e.g., a stream. A crucial property of KPNs is that they are independent of the relative delays of the buffers, i.e., a process always waits until all inputs are available to compute its result. Thus, the computation is independent of the order of input arrivals. This is also the case for stream specifications, where the result of a function can only be computed when all required input streams have provided their first element; the order of computing these is irrelevant. A difference is however that the fifo buffers in a KPN always start in an empty state. This requires the functions that take such buffers as input to be also defined for finite input sequences, which is not required for productivity where one can also reason solely about infinite objects.

6.5 Summary

This chapter presented techniques to automatically prove productivity of specifications of infinite objects such as streams. Previously, several techniques were developed for proving productivity of stream specifications, but not for other infinite data structures like infinite trees and combination of streams and finite lists. First, in Section 6.2, orthogonal specifications were considered. It was shown that by proving termination of a corresponding context-sensitive rewrite system [Luc02, GM04], productivity of a specification can be concluded. However, the other direction does not hold, i.e., from context-sensitive non-termination one must not conclude non-productivity. Hence, productivity-preserving transformations, such as rewriting of right-hand sides and case analysis, were presented to increase the strength of the productivity analysis.

There are examples where this approach outperforms all earlier techniques. For instance, the techniques from [EGH$^+$07, EGH08] fail to prove productivity of Example 6.2.5, as these techniques are data-oblivious, i.e., they do not take the value of data elements into account. For this specific example the technique from [ZR10b] succeeds, but it fails as soon binary stream operations such as zip come in.

Productivity has been characterized and investigated as a property of term rewriting, as was done before in the literature, see for example [EGH$^+$07, Isi08, EGH08, ZR10b, Isi10, ZR10a]. However, all these approaches were limited to orthogonal specifications, which is not suitable for analysis of hardware cells. As was already stated at the beginning of this chapter, the sequences of input values provided to a hardware cell are unknown, hence all possible sequences have to be considered. This is not possible with the restriction to orthogonal specifications, since there for example the constant rand, which produces all possible streams consisting of the values 0 and 1 by the two simple rules $\mathsf{rand} \to 0 : \mathsf{rand}$ and $\mathsf{rand} \to 1 : \mathsf{rand}$, are not allowed. Hence, this restriction was lifted in Section 6.3, to be able to abstract from the concrete input sequences provided to a hardware cell using the constant rand. For such non-orthogonal specifications the desired productivity should hold in all possible execution traces, not just some. Therefore, the notion of *strong productivity* [End10] was studied. Techniques were presented to also prove this form of productivity, which were similar to those presented for the orthogonal case. Also, a recent improvement presented in [EH11], that allows to block even more argument positions, could be extended to the non-orthogonal setting. However, the result of [EH11], showing that this improvement makes context-sensitive termination equivalent to productivity of strongly sequential orthogonal specifications, does not carry over to non-orthogonal specifications. It was shown in Section 6.4 that the extension to non-orthogonal specifications is able to prove productivity of stream specifications created from hardware circuits, which implies stabilization of the circuit regardless of the external input sequences that are provided.

However, the transformations that could be used in the orthogonal setting are not correct for non-orthogonal specifications; rewriting of right-hand sides for example is not correct since it considers only a single possible reduction sequence, but, in the non-orthogonal setting, all reduction sequences must be considered. This can be seen for example for the rules of the constant maybe, given as $\mathsf{maybe} \to 0 : \mathsf{maybe}$ and $\mathsf{maybe} \to \mathsf{maybe}$ in Example 6.1.10. The right-hand side of the second rule can be rewritten to $0 : \mathsf{maybe}$ by the first rule, which would be a productive specification according to Theorem 6.3.1. However, the original specification is of course not strongly productive. It would be interesting to investigate whether for example narrowing could be used instead.

Still some restrictions had to be imposed onto possibly non-orthogonal proper specifications. The most severe restriction is the requirement of left-linear rules. Dropping this requirement however would make Theorem 6.3.4, which showed that productivity can be concluded from context-sensitive termination, unsound. This was demonstrated in Example 6.3.10. Similarly, Example 6.3.11 showed that also the requirement that structure arguments of constructors must be variables cannot be dropped without losing soundness of Theorem 6.3.4. This requirement however is not that severe in practice, since many specifications can be unfolded by introducing fresh symbols, as was presented in [EH11, Zan09].

Some ideas in this chapter are related to earlier observations for orthogonal specifications. In [Hin08] the observation was made that if right-hand sides of stream definitions have the stream constructor ':' as its root, then well-definedness can be concluded, comparable to Theorem 6.2.1. A similar observation can be made about process algebra, where a recursive specification is called *guarded* if right-hand sides can be rewritten to a choice among terms all having a constructor on top, see for example [BBR10, Section 5.5]. In that setting every specification has at least one solution, while guardedness also implies there is at most one solution ([BBR10, Theorem 5.5.11]). So guardedness implies well-definedness, being of the flavor of combining Theorem 6.2.1 with rewriting right-hand sides. From both of these observations well-definedness is obtained, which is a slightly weaker notion than productivity. An investigation of well-definedness for stream specifications based on termination was made in [Zan09]. It should be stressed that productivity is strictly stronger than well-definedness, which is shown by the orthogonal proper (stream) specification $\mathsf{a} \to \mathsf{f}(\mathsf{a}),\ \ \mathsf{f}(x : xs) \to 0 : \mathsf{a}$. This specification is well-defined, since any term has to be interpreted as the infinite stream of zeroes. However, starting in the constant a, no stream constructor is ever produced. Well-definedness only requires that a unique solution exists, whereas productivity also requires the given rules to be able to compute such a unique solution. In the synchronous language Esterel [Ber99, PBEB07], a notion related to well-definedness is *logical correctness*, which requires that a unique model exists for a program. Similar to the above, the *constructiveness* of Esterel programs, which resembles productivity, is strictly stronger than logical correctness. It is shown for example in [Ber99] that there are Esterel programs which are logically correct but not constructive.

Another related notion is the Kahn principle of Kahn process networks [Kah74, GB10], which states that the operational rules can be applied to achieve the denotational behavior. This is similar to productivity of a stream specification, which expresses that by applying the rewriting rules the intended streams can be computed.

Chapter **7**

Productivity Analysis by Outermost Termination

The previous chapter presented techniques to prove productivity by means of context-sensitive termination. There, outermost-fair reduction sequences were considered to prove productivity, which was already observed in [End10]. In an outermost-fair reduction sequence not every reduction step is outermost. It is only required that every outermost redex is either eventually reduced, or it eventually is not an outermost redex anymore.

In this chapter, productivity of orthogonal specifications is investigated again, where the link to outermost rewriting is made explicit. In Section 7.1, which is based on [ZR10b], a special form of outermost rewriting called *balanced outermost rewriting* is considered. These rewriting sequences are similar to outermost-fair reductions, however it is required that all redexes are outermost, i.e., non-outermost redexes are never allowed to be contracted. The main result of this section is that if balanced outermost termination of a specification and the overflow rules $\{c(x_1, \dots, x_{\mathsf{ar}(c)}) \to \mathsf{overflow} \mid c \in \mathcal{C}\}$ can be proven, then the specification is productive.

In special cases, where no data rules exist and at most one structure argument is used, every outermost reduction sequence is balanced outermost. Furthermore, outermost termination always implies balanced outermost termination. Hence, outermost termination can be used to prove productivity. Section 7.2 presents one technique to prove outermost termination, by transforming such a problem into a standard termination problem. In this way, powerful existing termination provers such as AProVE [GSKT06], Jambox [End], μ-Term [L$^+$], or TTT2 [KSZM] can be used. This section is based on the work previously described in [RZ09].

7.1 Proving Productivity by Balanced Outermost Termination

The specifications considered in this section differ slightly from those that were defined in Definition 6.1.5. Here, *strictly proper* specifications are considered, for which also data arguments of constructors are required to be variables on left-hand sides of rules. This however can again be achieved by unfolding in many cases [Zan09, EH11]. This section is an extension of the work presented in [ZR10b], where only (strictly proper) stream specifications were treated.

Definition 7.1.1. Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be an orthogonal proper specification. Then $\mathcal{S}$ is called *strictly proper*, if for all $f(u_1, \ldots, u_m, t_1, \ldots, t_n) \to r \in \mathcal{R}_s$ with $\mathsf{ar}_d(f) = m$ and $\mathsf{ar}_s(f) = n$ it holds that if t_i is not a variable and $\mathrm{root}(t_i) \in \mathcal{C}$, then $t_i|_j \in \mathcal{V}$ for all $1 \leq j \leq \mathsf{ar}(\mathrm{root}(t_i))$.

The following example demonstrates the difference between orthogonal proper specifications and orthogonal strictly proper specifications.

Example 7.1.2. Consider the following orthogonal proper stream specification, given by the TRS $\mathcal{R}_s$. This specification was already considered in Example 6.2.5 of the previous chapter.

$$\begin{array}{rclcrcl} \mathsf{ones} & \to & 1 : \mathsf{ones} & \quad & \mathsf{f}(0 : xs) & \to & \mathsf{f}(xs) \\ & & & & \mathsf{f}(1 : xs) & \to & 1 : \mathsf{f}(xs) \end{array}$$

This specification is not strictly proper, since the arguments 0 and 1 of the constructor ':' on the left-hand sides for the function f are not variables. This however can be solved by unfolding, which introduces a fresh function symbol g with $\mathsf{ar}_d(g) = \mathsf{ar}_s(g) = 1$ and replaces the two rules for f. The unfolded specification then is the following:

$$\begin{array}{rclcrcl} \mathsf{ones} & \to & 1 : \mathsf{ones} & \quad & \mathsf{g}(0, xs) & \to & \mathsf{f}(xs) \\ \mathsf{f}(x : xs) & \to & \mathsf{g}(x, xs) & & \mathsf{g}(1, xs) & \to & 1 : \mathsf{f}(xs) \end{array}$$

This specification is strictly proper, since now the matching of the first element in the argument stream of f is deferred into the rules for g.

Next, the notion of balanced outermost rewriting is defined, which is a property of infinite reductions.

Definition 7.1.3. Let $\mathcal{R}$ be an arbitrary TRS. An infinite outermost reduction

$$t_1 \to_{p_1} t_2 \to_{p_2} t_3 \to_{p_3} t_4 \cdots$$

with respect to $\mathcal{R}$ is called *balanced outermost* if for every i and every redex of t_i on position q there exists $j \geq i$ such that $p_j \leq q$.

The TRS $\mathcal{R}$ is called *balanced outermost terminating* if it does not admit an infinite balanced outermost reduction.

A direct consequence is that for any infinite outermost reduction that is not balanced and contains a redex on position p in some term, every term later in the reduction has a redex on position p, too.

As an example the stream specification for the Thue Morse sequence is considered.

Example 7.1.4. The Thue Morse sequence morse is given by the following orthogonal strictly proper (stream) specification, where $\mathcal{R}_d$ contains the two rules $\mathsf{not}(0) \to 1$ and $\mathsf{not}(1) \to 0$ and the structure TRS $\mathcal{R}_s$ consists of the rules given below.

$$\begin{array}{rclcrcl} \mathsf{morse} & \to & 0 : \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})) & \quad & \mathsf{tail}(x : xs) & \to & xs \\ \mathsf{inv}(x : xs) & \to & \mathsf{not}(x) : \mathsf{inv}(xs) & & \mathsf{zip}(x : xs, ys) & \to & x : \mathsf{zip}(ys, xs) \end{array}$$

The infinite reduction

$$\begin{array}{rcl} \mathsf{tail}(\mathsf{morse}) & \to & \mathsf{tail}(0 : \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))) \\ & \to & \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})) \end{array}$$

continued by repeating this reduction forever on the created subterm $\mathsf{tail}(\mathsf{morse})$, is outermost, but not balanced, since the redex morse on position 1.1 in the term $\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))$ is never rewritten, and neither a higher redex. By forcing the infinite outermost reduction to be balanced, this redex should be rewritten, after which the rule for inv can be applied, and has to be applied due to balancedness, after which the first argument of zip will have ':' as its root, after which outermost reduction will choose the zip rule and create the constructor ':' as the root.

This leads to the main theorem regarding productivity of this chapter, showing that balanced outermost termination allows to conclude productivity for orthogonal strictly proper specifications.

Theorem 7.1.5. *An orthogonal strictly proper specification $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ is (weakly and stronly) productive, if $\mathcal{R}_d \cup \mathcal{R}_s \cup \{c(x_1, \ldots, x_{\mathsf{ar}(c)}) \to \mathsf{overflow} \mid c \in \mathcal{C}\}$ is balanced outermost terminating, where $x_i \neq x_j$ for all $1 \leq i < j \leq \mathsf{ar}(c)$ and $\mathsf{overflow} \notin \Sigma_d \cup \Sigma_s$ is a fresh symbol.*

To prove the above theorem, using the special shape of orthogonal strictly proper specifications, first a lemma is proven that states that any ground term not having a constructor as root symbol contains a redex that is not below any constructor.

Lemma 7.1.6. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be an orthogonal strictly proper specification, and let t be a ground term of sort s with $\mathrm{root}(t) \notin \mathcal{C}$. Then there exists a position $p \in \mathrm{Pos}(t)$ such that $t \to_p$ and for all $p' < p$, $\mathrm{root}(t|_{p'}) \notin \mathcal{C}$.*

Proof. This lemma follows from Lemma 6.3.6 in Section 6.3, together with the observation that arguments of constructors never need to be rewritten, due to the requirement that all arguments are variables in left-hand sides of strictly proper specifications. □

This lemma allows to prove this section's main theorem. Note that only orthogonal specifications are considered, hence weak and strong productivity coincide and it is sufficient to prove weak productivity, i.e., the existence of a reduction to a term starting with a constructor, cf. Proposition 6.1.11 in Section 6.1.

Proof of Theorem 7.1.5. Assume t is a ground term of sort s that is not productive, i.e., it does not rewrite to a term with a constructor as its root symbol. This allows to construct an infinite balanced outermost reduction w.r.t. $\mathcal{R}_d \cup \mathcal{R}_s \cup \{c(x_1, \ldots, x_{\mathsf{ar}(c)}) \to \mathsf{overflow} \mid c \in \mathcal{C}\}$: According to Lemma 7.1.6, there exists a position $p \in \mathrm{Pos}(t)$ such that $t \to_p$ and for all $p' < p$, $\mathrm{root}(t|_{p'}) \notin \mathcal{C}$. Hence, there exists a position $q_1 \leq p$ such that for some term t_1, $t \to_{q_1} t_1$ is an outermost step w.r.t. $\mathcal{R}_d \cup \mathcal{R}_s$. Since also for all $q' < q_1$, $\mathrm{root}(t|_{q'}) \notin \mathcal{C}$, this is also an outermost step w.r.t. $\mathcal{R}_d \cup \mathcal{R}_s \cup \{c(x_1, \ldots, x_{\mathsf{ar}(c)}) \to \mathsf{overflow} \mid c \in \mathcal{C}\}$. Also t_1 is not productive, otherwise, if t_1 would rewrite to a term with a constructor as its root symbol, then so would t. Hence, this argument can be repeated to obtain an infinite outermost reduction $t = t_0 \to_{q_1} t_1 \to_{q_2} t_2 \to_{q_3} \ldots$.

There might however be a term t_i and a redex on a position $p \in \mathrm{Pos}(t_i)$ that is never reduced or consumed in the constructed infinite outermost reduction. However, then there is never a reduction step above p in the remaining reduction, i.e., for all $j > i$, $q_j \not\leq p$. Since the reduction consists of outermost steps, it can be concluded that $q_j \not> p$, otherwise $t_{j-1} \to_{q_j} t_j$ would not be outermost. Hence, $q_j \parallel p$ for all $j > i$. Let $p' \leq p$ such that $t_i \to_{p'}$ is an outermost step. Then also $p' \parallel q_j$ for all

$j > i$, since $q_j \leq p' \leq p$ would contradict the assumption that $q_j \not\leq p$ and $q_j > p'$ would contradict the assumption that $t_{j-1} \to_{q_j} t_j$ is an outermost step. Therefore, the redex at position p' can be reduced at any time, without affecting reducibility of the redexes at positions q_j. These however might now become non-outermost steps. So let $t_0 \to^* t_i \to_{q_{i+1}} \cdots \to_{q_k} t_k \to_{p'} t'_{k+1}$ for some $k > i$ such that $t'_{k+1} \to_{q_{k+1}}$ is not an outermost step. But then the above reasoning that there is a redex on a position not below a constructor symbol in t'_{k+1} and following terms can be applied again, yielding another infinite outermost reduction for which the redex of t_i at position p is reduced or consumed. Repeating this construction gives an infinite balanced outermost reduction, which therefore proves the theorem. □

It should be remarked that the reverse direction of Theorem 7.1.5, other than it was stated in [ZR10b], does not hold in general. This was first discovered in [EH11], where a counterexample was given. This counterexample used structure functions with up to three arguments. Below, another counterexample is given that only uses function symbols with at most two arguments.

Example 7.1.7. Let $\mathcal{R}_s$ consist of the following rules:

$$\begin{aligned}
\mathsf{b} &\to \mathsf{f}(\mathsf{g}(\mathsf{zeroes}, \mathsf{zeroes}), \mathsf{b}) \\
\mathsf{zeroes} &\to 0 : \mathsf{zeroes} \\
\mathsf{f}(x : xs, ys) &\to 0 : \mathsf{zeroes} \\
\mathsf{g}(x : xs, y : ys) &\to 0 : \mathsf{zeroes}
\end{aligned}$$

The corresponding specification is orthogonal and strictly proper. Furthermore, it is productive, as can be seen by rewriting the right-hand side of the first rule as follows (where the reduced redexes are underlined):

$$\begin{aligned}
\mathsf{f}(\mathsf{g}(\underline{\mathsf{zeroes}}, \mathsf{zeroes}), \mathsf{b}) &\to \mathsf{f}(\mathsf{g}(0 : \mathsf{zeroes}, \underline{\mathsf{zeroes}}), \mathsf{b}) \\
&\to \mathsf{f}(\underline{\mathsf{g}(0 : \mathsf{zeroes}, 0 : \mathsf{zeroes})}, \mathsf{b}) \\
&\to \underline{\mathsf{f}(0 : \mathsf{zeroes}, \mathsf{b})} \to 0 : \mathsf{zeroes}
\end{aligned}$$

By Theorems 6.2.1 and 6.2.10 of the previous chapter productivity of the initial specification has therefore been proven.

However, when adding the rule $x : xs \to \mathsf{overflow}$ it admits an infinite balanced outermost reduction, where again the reduced redexes are underlined:

$$\begin{aligned}
\mathsf{b} &\to \mathsf{f}(\mathsf{g}(\underline{\mathsf{zeroes}}, \mathsf{zeroes}), \mathsf{b}) \\
&\to \mathsf{f}(\mathsf{g}(\underline{0 : \mathsf{zeroes}}, \mathsf{zeroes}), \mathsf{b}) \\
&\to \mathsf{f}(\mathsf{g}(\mathsf{overflow}, \underline{\mathsf{zeroes}}), \mathsf{b}) \\
&\to \mathsf{f}(\mathsf{g}(\mathsf{overflow}, \underline{0 : \mathsf{zeroes}}), \mathsf{b}) \\
&\to \mathsf{f}(\mathsf{g}(\mathsf{overflow}, \mathsf{overflow}), \underline{\mathsf{b}}) \\
&\to \ldots
\end{aligned}$$

This reduction can be continued by repeating the above reduction of the symbol b that is underlined in the last term. Note that $\mathsf{g}(\mathsf{overflow}, \mathsf{overflow})$ is a normal form, hence the reduction is balanced.

Using Outermost Termination Tools

Balancedness is obtained for free in case there are no rewrite rules for the data, i.e., $\mathcal{R}_d = \emptyset$, and there are no rules in $\mathcal{R}_s$ that have more than one argument of structure type s. This claim will be proven below.

Proposition 7.1.8. *Let $(\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ be an orthogonal strictly proper specification with $\mathcal{R}_d = \emptyset$ and the type of all $f \in \Sigma_s \setminus \mathcal{C}$ is of the form $d^m \times s^n \to s$ for some $m \in \mathbb{N}$, $n \in \{0,1\}$.*

Then every infinite outermost reduction $t_0 \to t_1 \to t_2 \to \dots$ is balanced.

Proof. Structural induction is performed to show that for any outermost reduction step $t \to_p t'$, $p \le p'$ holds for all positions $p' \in \mathrm{Pos}(t)$ for which $t|_{p'}$ is a redex.

If t is a term of sort d, then t is a normal form, due to $\mathcal{R}_d = \emptyset$. Hence, it does not contain any redex.

If $t = c \in \Sigma_s$ is a structure constant, then $\mathrm{Pos}(t) = \{\epsilon\}$. Thus, there is at most one redex which proves this case.

Otherwise, if $t = f(u_1, \dots, u_n)$ with $\mathsf{ar}_s(f) = 0$ (i.e., there is no argument of structure sort), then again $t \to_\epsilon$ holds. This is due to $\mathcal{R}_d = \emptyset$ and the exhaustiveness requirement of proper specifications, note that no data operations are allowed with arguments of structure sort. So this case has also been proven.

In the final case to consider, $t = f(u_1, \dots, u_n, \tilde{t})$ for some $f \in \Sigma_s$ with $\mathsf{ar}_s(f) = 1$. If $t \to_\epsilon$, then $p = \epsilon$ must hold, as $t \to_p t'$ was assumed to be an outermost step, which proves this case. Therefore, assume that $t \not\to_\epsilon$, i.e., $p > \epsilon$. Since $u_1, \dots, u_n$ are normal forms, because of $\mathcal{R}_d = \emptyset$, it must be the case that for all $p' \in \mathrm{Pos}(t)$ with $t \to_{p'}$, $m+1 \le p'$, hence this especially holds for p as well. Therefore, the induction hypothesis proves for the outermost reduction step $\tilde{t} \to_{p-(m+1)} \tilde{t}'$ that $p - (m{+}1) \le p''$ for all positions $p'' \in \mathrm{Pos}(\tilde{t})$ with $\tilde{t}|_{p''}$ being a redex. So, $p = (m{+}1).(p - (m{+}1))$ and for all $p' \in \mathrm{Pos}(t)$ with $t|_{p'}$ being a redex it holds that $p' = (m{+}1).p''$. Hence, it also holds that $p \le p'$, proving this final case and therefore the proposition. □

The specification of the Thue Morse sequence given in Example 7.1.4 shows the necessity of requiring at most one argument to be of structure sort. It was already observed that the infinite reduction

$$\begin{aligned}\mathsf{tail}(\mathsf{morse}) &\to \mathsf{tail}(0 : \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))\\ &\to \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))\\ &\to \dots,\end{aligned}$$

continued by repeatedly reducing the redex $\mathsf{tail}(\mathsf{morse})$, is outermost but not balanced. To show that also the requirement $\mathcal{R}_d = \emptyset$ is needed, an example is presented next that allows to construct an infinite outermost reduction that is not balanced. Consider the stream specification

$$\begin{aligned}\mathsf{tail}(x : xs) &\to xs\\ \mathsf{a} &\to 0 : \mathsf{f}(\mathsf{not}(1), \mathsf{tail}(\mathsf{a}))\\ \mathsf{f}(0, xs) &\to 1 : \mathsf{f}(0, xs)\\ \mathsf{f}(1, xs) &\to 0 : \mathsf{f}(1, xs)\end{aligned}$$

together with the rules $\mathcal{R}_d = \{\mathsf{not}(0) \to 1, \mathsf{not}(1) \to 0\}$. This orthogonal strictly proper specification is productive, as can for example be checked with the tool described in Section 6.2 or the productivity tool of [EGH08]. However, there also exists an infinite outermost reduction, namely

$$\mathsf{tail}(\mathsf{a}) \to \mathsf{tail}(0 : \mathsf{f}(\mathsf{not}(1), \mathsf{tail}(\mathsf{a}))) \to \mathsf{f}(\mathsf{not}(1), \mathsf{tail}(\mathsf{a})) \to \dots,$$

which is continued by repeatedly reducing the redex $\mathsf{tail}(\mathsf{a})$. This redex is outermost, since both rules having the symbol f as root require either 0 or 1 as first argument.

To apply one of these rules, the outermost redex $\mathsf{not}(1)$ would have to be reduced first, which shows that the above infinite outermost reduction is not balanced.

To also present an example that does satisfy the requirements of Property 7.1.8, an alternative definition of the Thue Morse stream is proven productive below.

Example 7.1.9. Consider the following definition of the Thue Morse stream:

$$\begin{array}{rcl} \mathsf{morse} & \to & 0 : \mathsf{a} \\ \mathsf{a} & \to & 1 : \mathsf{f}(\mathsf{a}) \\ \mathsf{f}(0 : xs) & \to & 0 : 1 : \mathsf{f}(xs) \\ \mathsf{f}(1 : xs) & \to & 1 : 0 : \mathsf{f}(xs) \end{array}$$

This TRS $\mathcal{R}_s$ is an orthogonal proper specification, but it is not strictly proper. After unfolding, an orthogonal strictly proper specification is obtained that still satisfies the requirements of Property 7.1.8. Thus, when adding the rule $x : xs \to \mathsf{overflow}$, outermost termination of the following TRS has to be shown:

$$\begin{array}{rcl} \mathsf{morse} & \to & 0 : \mathsf{a} \\ \mathsf{a} & \to & 1 : \mathsf{f}(\mathsf{a}) \\ \mathsf{f}(x : xs) & \to & \mathsf{g}(x, xs) \\ \mathsf{g}(0, xs) & \to & 0 : 1 : \mathsf{f}(xs) \\ \mathsf{g}(1, xs) & \to & 1 : 0 : \mathsf{f}(xs) \\ x : xs & \to & \mathsf{overflow} \end{array}$$

Outermost termination of the above TRS can for instance be proven using the transformation presented in the next Section 7.2 and AProVE [GSKT06] as a termination prover, or using the approach presented in [EH09]. This allows to conclude that the above stream specification is productive.

The next example is interesting, since it is not *friendly nesting*, a condition required by [EGH08] to be applicable. Essentially, a stream specification is friendly nesting if the right-hand sides of every nested symbol start with ':', which is clearly not the case for the second rule below.

Example 7.1.10. Consider the following TRS $\mathcal{R}_s$:

$$\begin{array}{rcl} \mathsf{a} & \to & 1 : \mathsf{a} \\ \mathsf{f}(x : xs) & \to & \mathsf{g}(x, xs) \\ \mathsf{g}(0, xs) & \to & 1 : \mathsf{f}(xs) \\ \mathsf{g}(1, xs) & \to & 0 : \mathsf{f}(\mathsf{f}(xs)) \end{array}$$

As can easily be seen, the above example fits into the orthogonal strictly proper specification format considered in this section and it satisfies the requirements of Property 7.1.8. After adding the rule $x : xs \to \mathsf{overflow}$, outermost termination can be proved automatically using the above techniques, which allows to conclude productivity of the example.

Finally, it should be remarked that outermost termination always implies balanced outermost termination, since an infinite balanced outermost reduction consists only of outermost steps. Thus, also for orthogonal strictly proper specifications that have either data rules or symbols with more than one structure argument productivity can be checked in the way presented here. However, in practice it is seldomly the case that for this type of specifications outermost termination can be proven.

7.2 Transformational Outermost Termination Analysis

A lot of work has been done on automatically proving termination and innermost termination. However, also termination with respect to the outermost strategy makes sense. For instance, this is the standard strategy in the functional programming language Haskell [Pey03], and it can be specified in CafeOBJ [FD98] and Maude [CDE$^+$03]. This section will focus on the most general variant of the outermost strategy: reducing a redex is always allowed if it is not a proper subterm of another redex. Termination with respect to this strategy will shortly be called *outermost termination*. This is different from the approaches for proving termination of Haskell presented in [PSS97, GRSK$^+$11], where termination is only proven for a specific set of terms (ground instantiations of a so-called *start term*) and not for every possible term. Furthermore, the language Haskell does not allow overlapping rules, i.e., there is at most one rule applicable for every term, and all arguments on left-hand sides are constructor terms.

A direct application of outermost termination analysis was presented in the previous Section 7.1. It showed that balanced outermost termination can be used to analyze productivity. In case general outermost termination can be proven, then this entails that there are no infinite balanced outermost reductions. Furthermore, the previous section identified cases where infinite outermost sequences and infinite balanced outermost sequences coincide.

The approach presented in this section works by transforming a given TRS into another TRS such that ground outermost termination of the original TRS is equivalent to termination of the transformed TRS. It will be shown that when fixing the signature there may be a difference between outermost termination and ground outermost termination, but by adding fresh constants there is no difference any more. Therefore it is not a restriction to focus on ground outermost termination.

The crucial ingredient of the presented transformation is *anti-matching*: for L being a set of terms such that it matches all terms that can be rewritten with the given TRS, a set S_L is needed such that any term matches with a term in S_L if and only if it does not match with a term in L. It turns out that if all terms in L are linear, then a finite set S_L satisfying this requirement can easily be constructed, while there are sets L containing non-linear terms such that every set S_L satisfying this property is infinite. For that reason only the class of *quasi left-linear* TRSs is treated, which are all TRSs where a left-hand side is an instance of a linear left-hand side. Clearly, this class also includes all left-linear TRSs.

Based on anti-matching a straightforward transformation T is given such that every infinite outermost reduction with respect to any quasi left-linear TRS $\mathcal{R}$ gives rise to an infinite $T(\mathcal{R})$-reduction. Several variants of the basic transformation were evaluated and the most powerful one is given in the final definition of the transformation. Additionally, two other transformations are given that can only prove but not disprove ground outermost termination. These transformations are simpler, hence proving termination for them is often easier. Also these transformations make use of an anti-matching set of terms by extracting contexts of rule instances that can never be redexes themselves.

Preliminaries

Below, some notation shall be fixed that is used in the rest of this section. Recall that a substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ is written as $\sigma = \{x_1 := t_1, \ldots, x_m := t_m\}$, which denotes the mapping $\sigma(x) = x$ and $\sigma(x_i) = t_i$ for all $x \neq x_i$ and $1 \leq i \leq m$. The set of all substitutions over Σ and $\mathcal{V}$ will be denoted as $\mathrm{SUB}(\Sigma, \mathcal{V})$. The application of a substitution $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ to a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is denoted $t\sigma$ and replaces all variables by their corresponding terms. Such a term $t\sigma$ is called an *instance* of t. Two terms $t, t' \in \mathcal{T}(\Sigma, \mathcal{V})$ are said to *unify*, if a *unifier* $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ exists such that $t\sigma = t'\sigma$. A term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is said to *match* a term $t' \in \mathcal{T}(\Sigma, \mathcal{V})$, if a substitution $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ exists such that $t\sigma = t'$.

A standard property relating linearity and unification is the following.

Lemma 7.2.1. *Let $t, t' \in \mathcal{T}(\Sigma, \mathcal{V})$ be two linear terms with $\mathcal{V}(t) \cap \mathcal{V}(t') = \emptyset$ that do not unify. Then there is a position $p \in \mathrm{Pos}(t) \cap \mathrm{Pos}(t')$ such that $t|_p$ and $t'|_p$ are no variables and $\mathrm{root}(t|_p) \neq \mathrm{root}(t'|_p)$.*

Proof (Sketch). For linear terms $t, t' \in \mathcal{T}(\Sigma, \mathcal{V})$ with $\mathcal{V}(t) \cap \mathcal{V}(u) = \emptyset$, it holds as an invariant of the standard Martelli-Montanari unification algorithm as given in [Ter03, Section 7.7] that every variable occurs at most once. Thus, the only way to get the result **fail** is by having two terms that have different root symbols. $\square$

A term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *outermost rewrites* to a term $t' \in \mathcal{T}(\Sigma, \mathcal{V})$ with a rule $\ell \to r \in \mathcal{R}$ at a position $p \in \mathrm{Pos}(t)$, denoted $t \overset{o}{\to}_{\ell \to r, p} t'$, if $t \to_{\ell \to r, p} t'$ and $t|_p$ is an outermost redex, that is for all positions $p' \in \mathrm{Pos}(t)$ with $p' < p$ it holds that $t \not\to_{\mathcal{R}, p'}$.

Given a TRS $\mathcal{R}$, the set of left-hand sides of that TRS is defined to be $\mathrm{lhs}(\mathcal{R}) = \{\ell \mid \ell \to r \in \mathcal{R}\}$. A TRS $\mathcal{R}$ is called *outermost terminating* if there is no infinite $\overset{o}{\to}$-chain. Given a TRS $\mathcal{R}$, $\mathcal{R}$ is called *ground terminating* (*outermost ground terminating*), if there is no infinite sequence $t_1, t_2, t_3 \ldots \in \mathcal{T}(\Sigma)$ of ground terms such that $t_i \to_{\mathcal{R}} t_{i+1}$ ($t_i \overset{o}{\to}_{\mathcal{R}} t_{i+1}$) for all $i \in \mathbb{N}$. The following example shows that outermost termination for arbitrary terms may differ from outermost ground termination.

Example 7.2.2. Consider the following left-linear term rewrite system $\mathcal{R}$ over the signature $\Sigma = \{\mathsf{f}, \mathsf{a}, \mathsf{b}\}$:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{f}(\mathsf{a}, x) & \to & \mathsf{a} & \mathsf{f}(x, \mathsf{a}) & \to & \mathsf{f}(x, \mathsf{b}) \\ \mathsf{f}(\mathsf{b}, x) & \to & \mathsf{a} & \mathsf{b} & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{f}(x, y), z) & \to & \mathsf{a} & & & \end{array}$$

For arbitrary terms, the following infinite outermost reduction exists:

$$\mathsf{f}(x, \mathsf{a}) \overset{o}{\to} \mathsf{f}(x, \mathsf{b}) \overset{o}{\to} \mathsf{f}(x, \mathsf{a}) \overset{o}{\to} \cdots$$

But, when instantiating the variable x by any arbitrary ground term $t \in \mathcal{T}(\Sigma)$ in the above reduction, then one of the three rules on the left is applicable at the root position. Hence, the reduction $\mathsf{f}(t, \mathsf{b}) \to \mathsf{f}(t, \mathsf{a})$ is no longer an outermost reduction. In fact, the above term rewrite system is outermost ground terminating, as will be shown later.

This difference between outermost termination and ground outermost termination only occurs when fixing the signature. It is easy to see that by replacing variables in any infinite outermost reduction by fresh constants, the result is an infinite outermost ground reduction. For quasi left-linear TRSs adding one fresh constant suffices.

Transformation of Outermost Termination to Standard Termination

The idea of the transformation is to only allow a reduction when a certain control symbol down marks the current redex. After having reduced a term, the control symbol is replaced by another control symbol up that is moved outwards. Only when the root of the term is encountered, then the control symbol is replaced by the down symbol again. In order to find the next outermost redex, the symbol down may only descend into subterms when no left-hand side is applicable to the term. For this purpose, a set S_L is needed such that its elements match exactly those terms that are not matched by a left-hand side. Such a set S_L is called *anti-matching*, as defined below.

Definition 7.2.3. A set $S_L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ is called *anti-matching* w.r.t. a set $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$, if the following holds for all ground terms $t \in \mathcal{T}(\Sigma)$:

$$\nexists \ell \in L, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V}) : t = \ell\tau \iff \exists s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V}) : t = s\sigma$$

Using such an anti-matching set, the transformation can be defined formally.

Definition 7.2.4. Let $\mathcal{R}$ be a TRS over a signature Σ and let $S_L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be an anti-matching set w.r.t. $L = \mathrm{lhs}(\mathcal{R})$.

Choose four fresh unary symbols $\mathsf{top}, \mathsf{up}, \mathsf{down}, \mathsf{block} \notin \Sigma$, and let $\Sigma^\natural = \{f^\natural \mid f \in \Sigma, \mathsf{ar}(f) > 0\}$ be such that $\Sigma \cap \Sigma^\natural = \emptyset$. The TRS $T(\mathcal{R})$ over the signature $\Sigma \cup \Sigma^\natural \cup \{\mathsf{top}, \mathsf{up}, \mathsf{down}, \mathsf{block}\}$ is defined to consist of the following rules:

- $\mathsf{down}(\ell) \to \mathsf{up}(r)$, for all rules $\ell \to r$ in $\mathcal{R}$;
- $\mathsf{top}(\mathsf{up}(x)) \to \mathsf{top}(\mathsf{down}(x))$;
- $\mathsf{down}(f(t_1, \ldots, t_n)) \to f^\natural(\mathsf{block}(t_1), \ldots, \mathsf{down}(t_i), \ldots, \mathsf{block}(t_n))$, for all $f(t_1, \ldots, t_n) \in S_L$ and all $i \in \{1, \ldots, n\}$;
- $f^\natural(\mathsf{block}(x_1), \ldots, \mathsf{up}(x_i), \ldots, \mathsf{block}(x_n)) \to \mathsf{up}(f(x_1, \ldots, x_n))$, for all $f \in \Sigma$ and all $i \in \{1, \ldots, n\}$, where $\mathsf{ar}(f) = n$ and $x_1, \ldots, x_n$ are distinct variables.

For an infinite TRS $\mathcal{R}$, clearly $T(\mathcal{R})$ is infinite, too. The TRS $T(\mathcal{R})$ can also become infinite for a finite TRS $\mathcal{R}$ that is not quasi left-linear, since then any anti-matching set S_L might be infinite. This is demonstrated later, when anti-matching is discussed in detail. For quasi left-linear TRSs however, a finite anti-matching set always exists and can be constructed automatically.

Theorem 7.2.5. *For a quasi left-linear TRS $\mathcal{R}$, there exists a finite, computable, and (up to variable renaming) unique set $S_L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ of linear terms that is anti-matching w.r.t. $L = \mathrm{lhs}(\mathcal{R})$.*

The proof of this theorem will be given later. However, the correctness of the transformation does not depend on finiteness of S_L.

In the transformation, the already mentioned symbols down and up are introduced to control the position of the next redex. The symbol top is used to denote the root position of the term, where the search of the next redex has to turn downwards again. The last of these fresh control symbols is the symbol block. Its purpose is to disallow evaluations where an up symbol appears at the root position of a term without having applied a rule of the form $\mathsf{down}(\ell) \to \mathsf{up}(r)$. Furthermore, for every function symbol f of the original rewrite system, a new marked symbol $f^\natural$ is created which has the same arity as f. This allows to use different interpretations for these symbols in the often used reduction pairs. Thus, it can be distinguished whether a symbol is above or below one of the control symbols, which often makes termination proofs easier.

An outermost rewrite step can be modelled by a sequence of steps in the transformed system. This is shown in the following lemma.

Lemma 7.2.6. *Let $\mathcal{R}$ be a TRS over a signature Σ, let $t, t' \in \mathcal{T}(\Sigma)$.*
If $t \overset{o}{\rightarrow}_{\mathcal{R}} t'$, then $\mathsf{down}(t) \rightarrow^+_{T(\mathcal{R})} \mathsf{up}(t')$.

Proof. Let $t, t' \in \mathcal{T}(\Sigma)$ be two ground terms and let $t \overset{o}{\rightarrow}_{\ell \rightarrow r, p} t'$ for some rule $\ell \rightarrow r \in \mathcal{R}$ and some position $p \in \mathrm{Pos}(t)$. Induction is done over the length of p.

In case $p = \epsilon$, then directly the rule $\mathsf{down}(\ell) \rightarrow \mathsf{up}(r)$ must be in $T(\mathcal{R})$, which shows the desired property.

Otherwise, let $p = i.p'$ for some $p' \in \mathrm{Pos}(t|_i)$ and let $t = f(t_1, \ldots, t_n)$. Hence, $t \not\rightarrow_\epsilon$ and $t' = f(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_n)$ for some $t'_i \in \mathcal{T}(\Sigma)$. Since t is a ground term, $t\tau = t$ and $t\tau = t \neq \ell'\sigma$ holds for all $\tau, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ and all $\ell' \in \mathrm{lhs}(\mathcal{R})$. Hence, by Definition 7.2.3 a term $s \in S_L$ and a substitution $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ exist such that $s\sigma = t$. Let $s = f(s_1, \ldots, s_n)$. Then a rule $\mathsf{down}(f(s_1, \ldots, s_n)) \rightarrow f^\natural(\mathsf{block}(s_1), \ldots, \mathsf{down}(s_i), \ldots, \mathsf{block}(s_n)) \in T(\mathcal{R})$ exists that is applicable to the term $\mathsf{down}(t)$ and therefore gives the reduction $\mathsf{down}(t) = \mathsf{down}(f(t_1, \ldots, t_n)) \rightarrow f^\natural(\mathsf{block}(t_1), \ldots, \mathsf{down}(t_i), \ldots, \mathsf{block}(t_n))$. For the subterm t_i it holds that $t_i \overset{o}{\rightarrow}_{\ell \rightarrow r, p'} t'_i$. Here, the induction hypothesis is applicable and yields a reduction $\mathsf{down}(t_i) \rightarrow^+ \mathsf{up}(t'_i)$. When applying this together with the rule $f^\natural(\mathsf{block}(x_1), \ldots, \mathsf{up}(x_i), \ldots, \mathsf{block}(x_n)) \rightarrow \mathsf{up}(f(x_1, \ldots, x_n)) \in T(\mathcal{R})$ the following reduction is obtained:

$$\begin{aligned}
\mathsf{down}(t) = \mathsf{down}(f(t_1, \ldots, t_n)) &\rightarrow \ f^\natural(\mathsf{block}(t_1), \ldots, \mathsf{down}(t_i), \ldots, \mathsf{block}(t_n)) \\
&\rightarrow^+ f^\natural(\mathsf{block}(t_1), \ldots, \mathsf{up}(t'_i), \ldots, \mathsf{block}(t_n)) \\
&\rightarrow \ \mathsf{up}(f(t_1, \ldots, t'_i, \ldots, t_n)) = \mathsf{up}(t')
\end{aligned}$$

This completes the proof of the lemma. □

Using the above lemma, the main theorem can be proven which shows that the presented transformation is sound, i.e., from termination of the transformed TRS outermost ground termination of the original TRS can be concluded.

Theorem 7.2.7. *Let $\mathcal{R}$ be a TRS over a signature Σ for which $T(\mathcal{R})$ is terminating. Then $\mathcal{R}$ is outermost ground terminating.*

Proof. Assume $\mathcal{R}$ is not outermost ground terminating. Then there is an infinite outermost reduction $t_1 \overset{o}{\rightarrow}_{\mathcal{R}} t_2 \overset{o}{\rightarrow}_{\mathcal{R}} \ldots$ for some ground terms $t_1, t_2, \ldots \in \mathcal{T}(\Sigma)$. For each $t_i \overset{o}{\rightarrow}_{\mathcal{R}} t_{i+1}$, $\mathsf{down}(t_i) \rightarrow^+_{T(\mathcal{R})} \mathsf{up}(t_{i+1})$ by Lemma 7.2.6. Due to the rule $\mathsf{top}(\mathsf{up}(x)) \rightarrow \mathsf{top}(\mathsf{down}(x))$ an infinite reduction in the transformed system $T(\mathcal{R})$ is obtained, i.e.,

$$\mathsf{top}(\mathsf{down}(t_1)) \rightarrow^+_{T(\mathcal{R})} \mathsf{top}(\mathsf{up}(t_2)) \rightarrow_{T(\mathcal{R})} \mathsf{top}(\mathsf{down}(t_2)) \rightarrow^+_{T(\mathcal{R})} \cdots,$$

contradicting termination of $T(\mathcal{R})$. □

Below the case of (arbitrary) outermost termination shall be considered, not just outermost ground termination. For a quasi left-linear TRS $\mathcal{R}$ over the signature Σ a new TRS $\mathcal{R}'$ is created which has the same rules as $\mathcal{R}$, but now is defined to be over the signature $\Sigma' = \Sigma \cup \{\mathsf{c}\}$, where $\mathsf{c} \notin \Sigma$ is a fresh constant (i.e., it has arity 0). Then an infinite reduction $t_1 \overset{o}{\to}_{\mathcal{R}} t_2 \overset{o}{\to}_{\mathcal{R}} t_3 \overset{o}{\to}_{\mathcal{R}} \ldots$ for some terms $t_1, t_2, t_3, \ldots \in \mathcal{T}(\Sigma, \mathcal{V})$ implies that $t_1\sigma_1 \overset{o}{\to}_{\mathcal{R}'} t_2\sigma_2 \overset{o}{\to}_{\mathcal{R}'} t_3\sigma_3 \overset{o}{\to}_{\mathcal{R}'} \ldots$ is an infinite reduction of ground terms $t_i\sigma_i \in \mathcal{T}(\Sigma)$ for substitutions $\sigma_i = \{x := \mathsf{c} \mid x \in \mathcal{V}(t_i)\}$, where $i \geq 1$. This holds, since no left-hand side of the rewrite system $\mathcal{R}'$ matches a subterm of $t_i\sigma_i$ which $\mathcal{R}$ does not match, because no left-hand side of $\mathcal{R}'$ contains the constant c. Furthermore, no redex that previously was an outermost redex can suddenly become a non-outermost redex; for this to be the case a non-linear left-hand side would have to match some subterm of $t_i\sigma_i$, whereas it did not match the corresponding subterm of t_i. However, due to $\mathcal{R}$ being quasi left-linear, this non-linear left-hand side would have to be an instance of a linear one, which would already have matched the subterm in t_i.

In the other direction, one can replace the symbol c in an infinite reduction w.r.t. $\mathcal{R}'$ by a fresh variable, giving an infinite reduction w.r.t. $\mathcal{R}$. Therefore, the term rewrite system $\mathcal{R}$ is outermost terminating, iff the term rewrite system $\mathcal{R}'$ is outermost ground terminating. Such a TRS $\mathcal{R}'$ can then be handled by the transformation presented above to show outermost termination of $\mathcal{R}$.

Other Transformations based on Contexts

There are examples for which the transformed TRS $T(\mathcal{R})$ cannot be proven terminating automatically. For some of these examples the transformations presented in the following are successful. These transformations do not use symbols down and up to find the next redex, but rewrite a redex when an *anti-matching context* is found.

Definition 7.2.8. For a set $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ of terms, a set $C_L \subseteq \mathcal{T}(\Sigma, \mathcal{V} \uplus \{\Box\})$ is called a set of *anti-matching contexts* w.r.t. L, iff

- $\Box \notin C_L$,
- $\Box$ occurs exactly once in every term $C \in C_L$, and
- if for some ground term $t \in \mathcal{T}(\Sigma)$ and position $p \in \mathrm{Pos}(t)$, $t|_p = \ell\sigma$ for some $\ell \in L$ and substitution $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ and $t \neq \ell'\sigma'$ for all $\ell' \in L$ and all $\sigma' \in \mathrm{SUB}(\Sigma, \mathcal{V})$, then there is a $C \in C_L$ and a $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V} \uplus \{\Box\})$ such that $t = C\tau$ and for $p' \in \mathrm{Pos}(C)$ with $C|_{p'} = \Box$ it holds that $p' \leq p$.

The third requirement above is similar to the requirements for an anti-matching set S_L. Thus, the set $\hat{C}_L = \{s[\Box]_i \mid s \in S_L, 1 \leq i \leq \mathsf{ar}(\mathrm{root}(s))\}$ is a set of anti-matching contexts, as can easily be checked. Using a set of anti-matching contexts, another transformation T_2 is defined. Here and in the following, $C[t]$ denotes the term $C[t]_p$, where $p \in \mathrm{Pos}(C)$ is the position of the context C such that $C|_p = \Box$.

Definition 7.2.9. For a TRS $\mathcal{R}$ over a signature Σ, let C_L be a set of anti-matching contexts w.r.t. $L = \mathrm{lhs}(\mathcal{R})$. The TRS $T_2(\mathcal{R})$ over the signature $\Sigma \uplus \{\mathsf{top}\}$ is defined to contain the following rules:

- $\mathsf{top}(\ell) \to \mathsf{top}(r)$ for all $\ell \to r \in \mathcal{R}$, and
- $C[\ell] \to C[r]$ for all $\ell \to r \in \mathcal{R}$ and all $C \in C_L$ such that $\mathcal{V}(\ell) \cap \mathcal{V}(C) = \emptyset$.

It will be shown below that this is a sound transformation, i.e., outermost termination of some TRS $\mathcal{R}$ can be concluded from termination of $T_2(\mathcal{R})$. First, the cases shall be identified in which $T_2(\mathcal{R})$ can be guaranteed to be finite. Clearly, if $\mathcal{R}$ is infinite then $T_2(\mathcal{R})$ is infinite, too. Since the construction of the set $\hat{C}_L$ relies on the anti-matching set S_L, finiteness of $T_2(\mathcal{R})$ can be guaranteed if $\mathcal{R}$ is finite and quasi left-linear. This can be observed from Theorem 7.2.5.

To prove soundness of the transformation T_2, it is first shown that any reduction can be put into the context $\mathsf{top}(\Box)$.

Lemma 7.2.10. *Let $t, t' \in \mathcal{T}(\Sigma)$ be two ground terms with $t \overset{o}{\to}_{\mathcal{R}} t'$. Then* $\mathsf{top}(t) \to_{T_2(\mathcal{R})} \mathsf{top}(t')$.

Proof. Let $t, t' \in \mathcal{T}(\Sigma)$ with $t = t[\ell\sigma]_p \overset{o}{\to}_{\ell \to r, p} t[r\sigma]_p = t'$ for some $\ell \to r \in \mathcal{R}$, some substitution σ, and position $p \in \mathrm{Pos}(t)$. If $p = \epsilon$, then $t = \ell\sigma$ and $t' = r\sigma$. By construction, $\mathsf{top}(\ell) \to \mathsf{top}(r) \in T_2(\mathcal{R})$, hence $\mathsf{top}(t) = \mathsf{top}(\ell\sigma) = \mathsf{top}(\ell)\sigma \to_{T_2(\mathcal{R})} \mathsf{top}(r)\sigma = \mathsf{top}(r\sigma) = \mathsf{top}(t')$.

Otherwise, $p = p'.i$ for some $p' \in \mathrm{Pos}(t)$ and $1 \leq i \leq \mathsf{ar}(\mathrm{root}(t|_{p'})$. Since $t \overset{o}{\to}_p t'$, it must be the case that $t \not\to_{p'}$ (or equivalently, $t|_{p'} \not\to_{\mathcal{R},\epsilon}$), otherwise the reduction would not be outermost. Thus, $t|_{p'} \neq \ell'\sigma'$ for all $\ell' \to r' \in \mathcal{R}$ and substitutions σ', and furthermore $t|_{p'} \to_{\ell \to r, i} t'|_{p'}$. Due to Definition 7.2.9 a context $C \in C_L$ and a substitution $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V} \uplus \{\Box\})$ exist such that $t|_{p'} = C\tau$ and for $p_\Box \in \mathrm{Pos}(C)$ with $C|_{p_\Box} = \Box$ it holds that $p_\Box \leq i$, i.e., either $p_\Box = \epsilon$ or $p_\Box = i$. If $p_\Box = \epsilon$ then $C = \Box$, which is not allowed by Definition 7.2.9. Hence, $p_\Box = i$. For the rule $C[\ell] \to C[r] \in T_2(\mathcal{R})$ it holds that $\mathcal{V}(\ell) \cap \mathcal{V}(C) = \emptyset$. Therefore, $t|_{p'} = C[\ell\sigma]\tau = C[\ell]\sigma\tau \to_{T_2(\mathcal{R}),\epsilon} C[r]\sigma\tau = C[r\sigma]\tau = t|_{p'}$ which implies that $\mathsf{top}(t) \to_{T_2(\mathcal{R}),1.p'} \mathsf{top}(t')$. ☐

This allows to prove soundness of the transformation T_2.

Theorem 7.2.11. *If for a TRS $\mathcal{R}$ and a set of anti-matching contexts $C_{\mathrm{lhs}(\mathcal{R})}$ the TRS $T_2(\mathcal{R})$ is terminating, then $\mathcal{R}$ is outermost ground terminating.*

Proof. Let $t_1 \overset{o}{\to}_{\mathcal{R}} t_2 \overset{o}{\to}_{\mathcal{R}} \ldots$ be an infinite outermost reduction for ground terms $t_1, t_2, \ldots \in \mathcal{T}(\Sigma)$. Then, due to Lemma 7.2.10, $\mathsf{top}(t_1) \to_{T_2(\mathcal{R})} \mathsf{top}(t_2) \to_{T_2(\mathcal{R})} \ldots$ is an infinite reduction w.r.t. the TRS $T_2(\mathcal{R})$, which proves the theorem. ☐

One might wonder why in the construction of $\hat{C}_L$ the hole $\Box$ is always introduced at depth 1, instead of replacing a variable of the term. Why this latter idea is not sound is illustrated in the below example.

Example 7.2.12. Consider the below TRS $\mathcal{R}$ over signature $\Sigma = \{\mathsf{f}, \mathsf{b}\}$:

$$\mathsf{b} \to \mathsf{f}(\mathsf{f}(\mathsf{b})) \qquad \mathsf{f}(\mathsf{b}) \to \mathsf{b} \qquad \mathsf{f}(\mathsf{f}(\mathsf{f}(x))) \to \mathsf{b}$$

For $L = \mathrm{lhs}(\mathcal{R})$, $S_L = \{\mathsf{f}(\mathsf{f}(\mathsf{b}))\}$, i.e., there are no variables in any term contained in S_L. Thus, for $\hat{C}'_L = \emptyset$, the only rules in $T_2(\mathcal{R})$ would be of the shape $\mathsf{top}(\ell) \to \mathsf{top}(r)$. The TRS consisting only of these rules can be shown terminating. However, the above TRS contains the following infinite outermost ground reduction, showing this approach to be unsound:

$$\mathsf{f}(\mathsf{f}(\mathsf{b})) \overset{o}{\to} \mathsf{f}(\mathsf{b}) \overset{o}{\to} \mathsf{b} \overset{o}{\to} \mathsf{f}(\mathsf{f}(\mathsf{b})) \overset{o}{\to} \ldots$$

It should be remarked that $\hat{C}'_L = \emptyset$ is not a set of anti-matching contexts w.r.t. L. This holds because the term $\mathsf{f}(\mathsf{f}(\mathsf{b}))$ is irreducible at the root position but contains the redex $\mathsf{f}(\mathsf{b})$ on position $p = 1$. Hence, the third requirement of Definition 7.2.9 is violated.

The transformation T_2 is always incomplete, i.e., non-termination of $T_2(\mathcal{R})$ does not imply outermost non-termination of $\mathcal{R}$, regardless of the concrete set C_L of anti-matching contexts.

Example 7.2.13. Consider the following TRS $\mathcal{R}$:

$$\mathsf{f}(\mathsf{h}(x)) \to \mathsf{f}(\mathsf{i}(x)) \qquad \mathsf{f}(\mathsf{i}(x)) \to \mathsf{a} \qquad \mathsf{i}(x) \to \mathsf{h}(x)$$

When this TRS is transformed using the transformation T_2, then the context $\mathsf{f}(\Box)$ must be considered, since $\mathsf{f}(\mathsf{f}(\mathsf{h}(x)))$ is not matched by any left-hand side of $\mathcal{R}$, but is reducible at position $p = 1$. Thus, the two rules $\mathsf{f}(\mathsf{f}(\mathsf{h}(x))) \to \mathsf{f}(\mathsf{f}(\mathsf{i}(x)))$ and $\mathsf{f}(\mathsf{i}(x)) \to \mathsf{f}(\mathsf{h}(x))$ must be contained in $T_2(\mathcal{R})$. These two rules form an infinite reduction, but the original TRS $\mathcal{R}$ is outermost ground terminating, as can be shown using the transformation T for example.

The above example failed since the rule $\mathsf{i}(x) \to \mathsf{h}(x)$ was inserted into the context $\mathsf{f}(\Box)$, but such a reduction will never take place due to the left-hand side $\mathsf{f}(\mathsf{i}(x))$. As an improvement, one can therefore make a more fine-grained analysis of possible reductions in arguments of contexts. These contexts are again derived from an anti-matching set S_L.

Definition 7.2.14. Let $\mathcal{R}$ be a TRS over signature Σ and let $S_L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be an anti-matching set w.r.t. $L = \mathrm{lhs}(\mathcal{R})$. The TRS $T_3(\mathcal{R})$ is defined over the signature $\Sigma \uplus \{\mathsf{top}\}$, containing the following rules:

- $\mathsf{top}(\ell) \to \mathsf{top}(r)$ for all $\ell \to r \in \mathcal{R}$, and
- $s[\ell]_i\mu \to s[r]_i\mu$ for all $s \in S_L$, all $1 \leq i \leq \mathsf{ar}(\mathrm{root}(s))$, and all $\ell \to r \in \mathcal{R}$ such that $s|_i$ and ℓ unify with the most general unifier μ.

The above transformation T_3 is also sound.

Theorem 7.2.15. *If the TRS $T_3(\mathcal{R})$ is terminating for a TRS $\mathcal{R}$, then $\mathcal{R}$ is outermost ground terminating.*

Proof Sketch. Observe that Lemma 7.2.10 also holds when replacing transformation T_2 by T_3, since the reduction on a position i below an anti-matching context must be matched by some left-hand side of $\mathcal{R}$, hence it must unify with the argument of an anti-matching term. Thus, a similar proof to that of Theorem 7.2.11 shows transformation T_3 to be sound. $\Box$

The transformation T_3 shall now be compared with T_2 on Example 7.2.13.

Example 7.2.16. For the TRS $\mathcal{R}$ shown in Example 7.2.13, the TRS $T_3(\mathcal{R})$ consists of the following rules, where $S_L = \{\mathsf{h}(x), \mathsf{a}, \mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{f}(x))\}$:

$$\begin{array}{rclrcl}
\mathsf{top}(\mathsf{f}(\mathsf{h}(x))) & \to & \mathsf{top}(\mathsf{f}(\mathsf{i}(x))) & \mathsf{h}(\mathsf{f}(\mathsf{h}(x))) & \to & \mathsf{h}(\mathsf{f}(\mathsf{i}(x))) \\
\mathsf{top}(\mathsf{f}(\mathsf{i}(x))) & \to & \mathsf{top}(\mathsf{a}) & \mathsf{h}(\mathsf{f}(\mathsf{i}(x))) & \to & \mathsf{h}(\mathsf{a}) \\
\mathsf{top}(\mathsf{i}(x)) & \to & \mathsf{top}(\mathsf{h}(x)) & \mathsf{h}(\mathsf{i}(x)) & \to & \mathsf{h}(\mathsf{h}(x)) \\
\mathsf{f}(\mathsf{f}(\mathsf{h}(x))) & \to & \mathsf{f}(\mathsf{f}(\mathsf{i}(x))) & \mathsf{f}(\mathsf{f}(\mathsf{i}(x))) & \to & \mathsf{f}(\mathsf{a})
\end{array}$$

As can be seen, the TRS $T_3(\mathcal{R})$ does not contain the rule $\mathsf{f}(\mathsf{i}(x)) \to \mathsf{f}(\mathsf{h}(x))$, which caused $T_2(\mathcal{R})$ to be non-terminating. And indeed, the TRS $T_3(\mathcal{R})$ can be shown terminating automatically.

However, the improved transformation is still incomplete, as witnessed by the following counter-example.

Example 7.2.17. Consider the following TRS:

$$\mathsf{h}(x) \;\to\; x : 0 : \mathsf{h}(\mathsf{s}(x)) \qquad\qquad \mathsf{s}(x) : xs \;\to\; \mathsf{overflow}$$

Outermost ground termination of this example can be shown using the transformation T. However, $0 : xs \in S_L$ must hold for any anti-matching set S_L. Thus, the rule $0 : \mathsf{h}(x) \to 0 : x : \underline{0 : \mathsf{h}(\mathsf{s}(x))}$ must be contained in $T_3(\mathcal{R})$. For this rule the underlined part shows that the same rule is applicable again, leading to an infinite reduction.

Anti-Matching

To be able to construct the transformed TRS $T(\mathcal{R})$ from a term rewrite system $\mathcal{R}$ that shall be checked for outermost termination, a set L is considered that matches all terms which can be rewritten by $\mathcal{R}$, for example $L = \mathrm{lhs}(\mathcal{R})$. Then, a set S_L of terms has to be found that describe the terms which cannot be rewritten by $\mathcal{R}$. Clearly, this only depends on the left-hand sides. Rewriting w.r.t. a TRS is done by matching the left-hand sides to some other terms. Thus, the set S_L should contain terms that match those ground terms not matched by the terms contained in L. One can imagine that there are several possible sets that satisfy this condition. The goal is to select the smallest such set and to be able to construct it finitely when this is possible.

This problem is treated in its general form: finding a set of terms that match the terms not matched by some terms contained in another set. Only later the case of linear terms is considered, where it will be shown that for this restriction the set S_L is finite and can be computed.

The problem of finding a set of terms that describe the complement of a set of terms is similar to the problem considered by Lassez and Marriot [LM87], where an explicit representation of a set is being searched that is described using counter examples. But their focus is on machine learning, therefore it is hard to directly apply their results. Also the concept of *anti-patterns* as introduced in [KKM07] is related to the problem studied here. Anti-patterns are more general since they allow to introduce negation of patterns at any position in a term, while here only negation of a complete pattern is required. However, this work is not applicable here, since a representation of a set that does not match a given term is required, while an anti-pattern matching problem is to decide whether an anti-pattern matches a single ground term.

Below, first a set S'_L of terms that satisfy the desired property is defined. This set is usually infinite and contains quite a number of redundant terms, i.e., terms that are already matched by other terms contained in S'_L. Thus, another set S_L is defined that consists only of the minimal elements of S'_L w.r.t. an order that expresses whether one term matches the other.

Definition 7.2.18. Let $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be a set of terms. On terms the preorder $\leq$ is defined by

$$t \leq u \iff \exists\sigma : t\sigma = u$$

which induces the definition of its strict part to be

$$t < u \iff t \leq u \land \neg(u \leq t).$$

Now S_L is defined to be the set of minimal elements of the set of terms that do not unify with elements of L, i.e.,

$$\begin{aligned} S'_L &= \{t \in \mathcal{T}(\Sigma, \mathcal{V}) \mid \nexists \ell \in L, \sigma, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V}) : \ell\sigma = t\tau\} \\ S_L &= \{t \in S'_L \mid \nexists u \in S'_L : u < t\} \end{aligned}$$

One might wonder why unification is considered, while term rewriting is concerned with matching. This becomes clear when formulating what kind of terms are sought: the set of terms that *match* those terms which are not *matched* by left-hand sides. This means that two matchings have to be considered at the same time. When assuming the set of variables to be disjoint then this gives rise to a unification problem.

As a next step it is shown that the set S'_L is closed under substitution. This is of interest, since the ground terms that are matched by a term contained in S'_L are considered. Thus, it should be the case that every instantiation of a term from S'_L is also contained in S'_L, such that this especially holds for ground instances.

Lemma 7.2.19. $\{s\sigma \in \mathcal{T}(\Sigma, \mathcal{V}) \mid s \in S'_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = S'_L$.

Proof. "$\supseteq$": Holds trivially for $\sigma = \mathrm{id}$.

"$\subseteq$": Let $s \in S'_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$. Then $s\sigma \in \{s\sigma \in \mathcal{T}(\Sigma, \mathcal{V}) \mid s \in S'_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$. It holds that $\ell\sigma' \neq s\tau'$ for all $\ell \in L$ and all substitutions $\sigma', \tau' \in \mathrm{SUB}(\Sigma, \mathcal{V})$. Therefore, especially $\ell\sigma' \neq s\sigma\tau'$ holds for all substitutions $\sigma', \tau' \in \mathrm{SUB}(\Sigma, \mathcal{V})$. Hence, $s\sigma \in S'_L$. ☐

The set S_L is derived from the set S'_L by taking only the minimal elements of S'_L w.r.t. the order $>$. In order to be able to show that these minimal elements exist, it is shown that this order is well-founded, i.e., there are no infinite descending chains.

Lemma 7.2.20. *The relation $>$ from Definition 7.2.18 is well-founded.*

Proof. Assume, $>$ were not well-founded, i.e., there exists an infinite sequence $t_1 > t_2 > \ldots$ for some $t_1, t_2, \ldots \in \mathcal{T}(\Sigma, \mathcal{V})$. Thus, for every $i > 1$, $t_{i-1}\tau \neq t_i$ for all $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ and substitutions $\sigma_i \in \mathrm{SUB}(\Sigma, \mathcal{V})$ exist such that $t_{i-1} = t_i\sigma_i$. For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, let $\#_\Sigma(t) \in \mathbb{N}$ denote the number of symbols from Σ contained in t and let $\#_{2\times\mathcal{V}}(t) \in \mathbb{N}$ denote the number of variables that occur more than once in t. If for a variable $x_i \in \mathcal{V}(t_i)$ it holds that $\sigma_i(x_i) \notin \mathcal{V}$, then $\#_\Sigma(t_{i-1}) = \#_\Sigma(t_i\sigma_i) >_\mathbb{N} \#_\Sigma(t_i)$. Otherwise, $\sigma_i(x) \in \mathcal{V}$ for all $x \in \mathcal{V}$. Then there must be two variables $x_i, y_i \in \mathcal{V}(t_i)$ with $x_i \neq y_i$ and $\sigma_i(x_i) = \sigma_i(y_i) \in \mathcal{V}$, since otherwise one could define $\tau = \{y := x \mid \sigma_i(x) = y\}$ and would get $t_{i-1}\tau = t_i$. Hence, in this case $\#_\Sigma(t_{i-1}) = \#_\Sigma(t_i)$ and $\#_{2\times\mathcal{V}}(t_{i-1}) >_\mathbb{N} \#_{2\times\mathcal{V}}(t_i)$. Because the lexicographic combination of two well-founded orders is also well-founded, this gives a contradiction since an infinite descending chain for the lexicographic combination of $>_\mathbb{N}$ on $\#_\Sigma$ and $>_\mathbb{N}$ on $\#_{2\times\mathcal{V}}$ was constructed. This proves the lemma. ☐

When removing larger elements from a set w.r.t. $\leq$, then all terms that are matched by removed terms are still being matched by some term in the set. This is proven in the next lemma and will be used to show that S_L still matches the same terms as S'_L.

Lemma 7.2.21. $\{u\sigma \mid u \in U, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = \{u\sigma \mid u \in U \cup U', \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$ *for every* $U, U' \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ *with* $U' = \{u' \mid \exists u \in U : u \leq u'\}$.

Proof. "$\subseteq$": trivial, since $U \subseteq U \cup U'$.

"$\supseteq$": Let $u' \in U \cup U'$, let $\sigma' \in \mathrm{SUB}(\Sigma, \mathcal{V})$. If $u' \in U$, then the property trivially holds. So let $u' \in U' \setminus U$. Then a $u \in U$ exists such that $u \leq u'$, i.e., there is a substitution $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ such that $u' = u\tau$. Hence, $u'\sigma' = u\tau\sigma' \in \{u\sigma \mid u \in U, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$. ☐

For the set S'_L it should be intuitively clear that all terms that are not matched by a term contained in L are matched by a term in that set. Using the above lemma, it can now be shown that this already holds for the set S_L.

Lemma 7.2.22. $\{s\sigma \in \mathcal{T}(\Sigma, \mathcal{V}) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = S'_L$.

Proof. "$\subseteq$": Since $S_L \subseteq S'_L$, this holds due to Lemma 7.2.19.

"$\supseteq$": Since $>$ is well-founded as shown in Lemma 7.2.20, the existence of the minimal elements in S_L is guaranteed. Thus, Lemma 7.2.21 shows the desired property. ☐

This allows to prove that the ground terms matched by S_L are indeed those terms that are not matched by the set L.

Lemma 7.2.23. $\mathcal{T}(\Sigma) \setminus \{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$.

Proof. This lemma is shown in two steps: First it is proven that $\{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} \cap \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = \emptyset$ (showing "$\supseteq$"), and in the second step it is shown that $\mathcal{T}(\Sigma) = \{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} \cup \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$ (showing "$\subseteq$").

For the first step, let $t \in \{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} \cap \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$. Thus, there exist $\ell \in L$ and $\sigma_\ell \in \mathrm{SUB}(\Sigma, \mathcal{V})$ such that $t = \ell\sigma_\ell$ and there exist $s \in S_L \subseteq S'_L$ and $\sigma_s \in \mathrm{SUB}(\Sigma, \mathcal{V})$ such that $t = s\sigma_s$. Putting this together gives $\ell\sigma_\ell = t = s\sigma_s$, which is a contradiction to the definition of S'_L.

To show the second step, observe that clearly $\{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} \cup \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} \subseteq \mathcal{T}(\Sigma)$. So it remains to be shown that $\{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} \cup \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} \supseteq \mathcal{T}(\Sigma)$. For that purpose, let $t \in \mathcal{T}(\Sigma)$ be an arbitrary ground term. In case there exist $\ell \in L$ and $\sigma_\ell \in \mathrm{SUB}(\Sigma, \mathcal{V})$ such that $\ell\sigma_\ell = t$ the property has been shown. Otherwise, for all $\ell \in L$ and all substitutions $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ that satisfy $\ell\sigma \in \mathcal{T}(\Sigma)$ it holds that $\ell\sigma \neq t$. Since t is a ground term, $\mathcal{V}(t) = \emptyset$. This means that $t\tau = t$ for any substitution $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$. Furthermore, for every term $t' \in \mathcal{T}(\Sigma, \mathcal{V})$ with $\mathcal{V}(t') \neq \emptyset$ it holds that $t \neq t'$ which allows to conclude that $t \neq \ell\sigma'$ for all substitutions $\sigma' \in \mathrm{SUB}(\Sigma, \mathcal{V})$ where $\mathcal{V}(\ell\sigma') \neq \emptyset$. Combining these observations, one sees that for all substitutions $\sigma, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ it holds that $\ell\sigma \neq t\tau = t$. Due to the definition of S'_L, $t \in S'_L$ and hence $t \in \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$ by Lemma 7.2.22, which completes the proof. ☐

In the following, only sets L are considered that contain linear terms. It should be observed that this also covers the case of a quasi left-linear TRS $\mathcal{R}$: for such a TRS the set L can be defined to contain all linear left-hand sides of $\mathcal{R}$. Then L still matches the same terms as $\mathrm{lhs}(\mathcal{R})$, due to Lemma 7.2.21. The goal is to show that for a linear set L the set S_L is finite. For that purpose the depth of a term is needed, which is defined as follows.

Definition 7.2.24. The *depth* of a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is defined as $\mathrm{depth}(t) = 0$ if $t \in \mathcal{V}$ and $\mathrm{depth}(f(t_1, \dots, t_n)) = 1 + \max\{\mathrm{depth}(t_1), \dots, \mathrm{depth}(t_n)\}$ for $t = f(t_1, \dots, t_n)$.

The *depth* of a finite set $T \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ is defined as the maximum over the depths of the terms it contains, i.e., $\mathrm{depth}(T) = \max\{\mathrm{depth}(t) \mid t \in T\}$.

Then for example $\mathrm{depth}(\mathsf{f}(x, y)) = 1$, while $\mathrm{depth}(\mathsf{f}(\mathsf{a}, y)) = 2$ for the signature $\Sigma = \{\mathsf{f}, \mathsf{a}\}$. Using this notion of depth, the following lemma can be proven. It provides an upper bound on the depth of the terms contained in S_L for sets L containing only linear terms.

Lemma 7.2.25. *For a set $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ containing only linear terms,* $\mathrm{depth}(S_L) \leq \mathrm{depth}(L)$.

Proof. Assume, there exists a $s \in S_L \subseteq S'_L$ with $\mathrm{depth}(s) > \mathrm{depth}(L)$. Then, $s\sigma \neq \ell\tau$ for all $\ell \in L$ and all substitutions $\sigma, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$. W.l.o.g. it may be assumed that $\mathcal{V}(s)$ and $\mathcal{V}(\ell)$ are disjoint for all $\ell \in L$. Lemma 7.2.1 shows that for every $\ell \in L$ a position $p_\ell \in \mathrm{Pos}(\ell) \cap \mathrm{Pos}(s)$ exists such that $\mathrm{root}(\ell|_{p_\ell}) \neq \mathrm{root}(s|_{p_\ell})$. By definition of $\mathrm{depth}(L)$, it holds that $|p_\ell| < \mathrm{depth}(L)$. Let $\mathrm{trunc}_L(s) \in \mathcal{T}(\Sigma, \mathcal{V})$ denote the term that is derived from s by replacing all subterms at positions of length $\mathrm{depth}(L)$ by fresh variables. By construction, $\mathrm{depth}(\mathrm{trunc}_L(s)) = \mathrm{depth}(L)$, $\mathrm{trunc}_L(s) < s$, and $\mathrm{root}(s|_p) = \mathrm{root}(\mathrm{trunc}_L(s)|_p)$ for all $p \in \mathrm{Pos}(s)$ with $|p| < \mathrm{depth}(L)$. Hence, $\mathrm{root}(\mathrm{trunc}_L(s)|_p) = \mathrm{root}(s|_p) \neq \mathrm{root}(\ell|_{p_\ell})$, i.e., for all substitutions $\sigma, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$, $\mathrm{trunc}_L(s)\sigma \neq \ell\tau$. Thus $\mathrm{trunc}_L(s) \in S'_L$, which contradicts the minimality of s. □

Furthermore, only linear terms have to be considered for the set S_L, if S_L is to be used for anti-matching of ground terms.

Lemma 7.2.26. *Let $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be a set of linear terms. For every $t \in \mathcal{T}(\Sigma) \cap S'_L$ it holds that a linear term $s \in S_L$ and a substitution $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ exist with $s\sigma = t$.*

Proof. Let $t \in \mathcal{T}(\Sigma) \cap S'_L$. Then for all $\ell \in L$ and all $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ the definition of anti-matching gives $\ell\tau \neq t$. Since $\mathcal{T}(\Sigma) \subseteq \mathcal{T}(\Sigma, \mathcal{V})$, Lemma 7.2.1 shows that a position $p_\ell \in \mathrm{Pos}(\ell) \cap \mathrm{Pos}(t)$ exists with $\mathrm{root}(t|_{p_\ell}) \neq \mathrm{root}(\ell|_{p_\ell})$.

There exist $s \in S_L$ and $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ with $s\sigma = t$, due to Lemma 7.2.22. In case s is a linear term, then nothing has to be proven.

Otherwise, start with the term $\mathrm{lin}(s)$ that is created from s by replacing every occurrence of a variable by a fresh variable, thereby generating a linear term. Then clearly, there is a substitution σ' such that $\mathrm{lin}(s)\sigma' = t$. If there is an $\ell' \in L$ and a substitution $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ such that $\mathrm{lin}(s)\tau = \ell'\tau$ (where it is assumed that $\mathcal{V}(\mathrm{lin}(s)) \cap \mathcal{V}(\ell') = \emptyset$), then the variable at a position p_s that is a prefix of $p_{\ell'}$ is replaced by $f(x_1, \dots, x_n)$, where $f = \mathrm{root}(t|_{p_s})$, $\mathsf{ar}(f) = n$, and the x_i are pairwise-disjoint fresh variables. This variable must exist, otherwise ℓ' would match t. This process is repeated until there are no more ℓ' that unify with the thereby constructed term s'. By construction s' is linear and does not unify with any term from L. Furthermore, this term is minimal in S'_L w.r.t. $>$, therefore $s' \in S_L$, which shows the claim. □

From the above lemmas, the following construction yields a set S for a set L of linear terms that satisfies the requirement of Definition 7.2.4. Let $d = \mathrm{depth}(L)$ be the maximal depth of terms occurring in L. Start by S' being the finite set of all linear terms up to renaming of variables of depth $\leq d$. Next remove all terms from S' that unify with L. Finally initialize S to S' and remove all non-minimal elements t from S, i.e., every term t for which a $u \in S$ exists with $u < t$ is removed from S. Lemmas 7.2.25 and 7.2.26 then show that all ground terms that are not matched by L are matched by S.

Using this construction and the above lemmas, Theorem 7.2.5 can now be proven. It states that for a quasi left-linear TRS $\mathcal{R}$ a finite, computable, and unique set S exists that matches exactly those terms that $\mathrm{lhs}(\mathcal{R})$ does not match. Note that only a linear set L that matches all ground terms matched by $\mathrm{lhs}(\mathcal{R})$ has to be consider for a quasi left-linear TRS, as was already observed above.

Proof of Theorem 7.2.5. Let $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be the finite set of linear left-hand sides of $\mathcal{R}$. Then L matches all terms that can be rewritten by $\mathcal{R}$ due to Lemma 7.2.21. Let $S_L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be defined as given in Definition 7.2.18. As can be seen from Lemma 7.2.23, for all ground terms $t \in \mathcal{T}(\Sigma)$ it holds that $t \in \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$ iff $t \notin \{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$. Due to Lemma 7.2.25 S_L is finite, since, up to variable renaming, only finitely many terms whose depth is less than or equal to $\mathrm{depth}(L)$ exist for a finite signature Σ. Lemma 7.2.26 shows that $S = S_L \cap \{t \in \mathcal{T}(\Sigma, \mathcal{V}) \mid t \text{ is linear}\}$, and finally the sketched construction shows that the set S is computable and unique since the minimal elements w.r.t. $>$ are unique. $\square$

Finally, the case of TRSs that are not quasi left-linear shall be studied. For this purpose, let $L = \{\mathsf{f}(x, x)\}$ be the left-hand sides of a TRS over the signature $\Sigma = \{\mathsf{f}, \mathsf{g}\}$. Then for every $n \in \mathbb{N}$ the term $\mathsf{f}(x, \mathsf{g}^n(x))$ is contained in S'_L. Furthermore, there is no term $s \neq \mathsf{f}(x, \mathsf{g}^n(x)) \in S'_L$ such that $s\sigma = \mathsf{f}(x, \mathsf{g}^n(x))$, which shows that S_L is infinite. To show that this is not due to choosing the set S_L, the proposition below states that S_L is the smallest set that has the desired property.

Proposition 7.2.27. *Let $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$. For every $S \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ that satisfies $\forall t \in \mathcal{T}(\Sigma) : (\exists s \in S, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V}) : t = s\sigma) \iff \neg(\exists \ell \in L, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V}) : t = \ell\tau)$ it holds that $S_L \subseteq S \subseteq S'_L$, where variable renamings are disregarded.*

Proof. The inclusion $S \subseteq S'_L$ can be seen directly from the definition of S'_L.

Assume, there is such a set $S \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ with $S_L \nsubseteq S$. Then, there is a term $s' \in S_L$ such that $s' \notin S$. Furthermore, it must be the case that $\{s\sigma \mid s \in S, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = \{s\sigma \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = S'_L$, i.e., there must be an $s \in S$ and a $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ such that $s\sigma = s'$. This implies that $s \leq s'$. In case it also holds that $s' \leq s$, then $s' \in S$, contradicting the assumption. But otherwise $s < s'$ holds, which contradicts the minimality of s'. $\square$

As a consequence of Proposition 7.2.27 and the previously observed fact that for $L = \{\mathsf{f}(x, x)\}$ it holds that $S_L \supseteq \{\mathsf{f}(x, \mathsf{g}^n(x)) \mid n \in \mathbb{N}\}$, it can be concluded that any set S that matches those terms which are not matched by a term in L must be infinite, since already $S_L \subseteq S$ is infinite.

Completeness of the Basic Transformation

The transformations T_2 and T_3 were already shown to be incomplete. In the following, it will be shown that the basic transformation T, which uses the symbols down and up to control the position of the next redex, is complete, i.e., from non-termination of $T(\mathcal{R})$ the outermost non-termination of $\mathcal{R}$ may be concluded.

Theorem 7.2.28. *Let $\mathcal{R}$ be a quasi left-linear TRS over signature Σ containing a constant $c \in \Sigma$ with $\mathsf{ar}(c) = 0$. If $T(\mathcal{R})$ is not terminating, then $\mathcal{R}$ is outermost ground non-terminating.*

To prove the above theorem, the signatures $\Sigma_{\text{ctrl}} = \{\mathsf{top}, \mathsf{down}, \mathsf{up}, \mathsf{block}\} \cup \{f^\natural \mid f \in \Sigma, \mathsf{ar}(f) > 0\}$ and $\Sigma_{\cup\text{ctrl}} = \Sigma \cup \Sigma_{\text{ctrl}}$ are defined.

In order to get rid of terms that occur in an infinite evaluation and contain one of the control symbols from Σ_{ctrl}, the function drop is defined. It replaces all terms having a control symbol as root with a constant. Therefore, it is assumed that the signature Σ contains at least one constant c, i.e., a symbol with $\mathsf{ar}(c) = 0$, as required in Theorem 7.2.28. Note that if no constant existed, then no ground terms could exist either, so a TRS over such a signature would trivially be (outermost) ground terminating.

Definition 7.2.29. Let $c \in \Sigma$ be a constant. For a term $t \in \mathcal{T}(\Sigma_{\cup\text{ctrl}}, \mathcal{V})$ the function $\text{drop} : \mathcal{T}(\Sigma_{\cup\text{ctrl}}, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V})$ is defined as follows:

- $\text{drop}(x) = x$ for all $x \in \mathcal{V}$,
- $\text{drop}(f(t_1, \dots, t_n)) = f(\text{drop}(t_1), \dots, \text{drop}(t_n))$ for all $f \in \Sigma$ of arity n, and
- $\text{drop}(f_{\text{ctrl}}(t_1, \dots, t_n)) = c$ for all $f \in \Sigma_{\text{ctrl}}$ of arity n.

The definition of drop is extended to substitutions $\sigma \in \text{SUB}(\Sigma_{\cup\text{ctrl}}, \mathcal{V})$ by defining $\text{drop}(\sigma)\,(x) = \text{drop}(\sigma(x))$ for all $x \in \mathcal{V}$.

Corollary 7.2.30. *For all terms $t \in \mathcal{T}(\Sigma, \mathcal{V})$ and substitutions $\sigma \in \text{SUB}(\Sigma_{\cup\text{ctrl}}, \mathcal{V})$, $\text{drop}(t\sigma) = t\ \text{drop}(\sigma)$.*

Corollary 7.2.31. *For all terms $t \in \mathcal{T}(\Sigma_{\cup\text{ctrl}}, \mathcal{V})$ and positions $p \in \text{Pos}(t)$ it holds that $\text{drop}(t) = \text{drop}(t[\text{drop}(t|_p)]_p)$.*

In the following lemma it is shown that reductions w.r.t. the TRS $T(\mathcal{R})$ are removed when applying the function drop. Intuitively, this holds because all defined symbols in $T(\mathcal{R})$ are from Σ_{ctrl}.

Lemma 7.2.32. *If $t \to^*_{T(\mathcal{R})} t'$ for some terms $t, t' \in \mathcal{T}(\Sigma_{\cup\text{ctrl}}, \mathcal{V})$, then $\text{drop}(t) = \text{drop}(t')$.*

Proof. Induction on the length k of the reduction $t \to^k_{T(\mathcal{R})} t'$ is performed. If the length k is 0, then $t = t'$ and hence $\text{drop}(t) = \text{drop}(t')$.

Otherwise, the reduction has the form $t \to_{\ell \to r, p} \hat{t} \to^{k-1}_{T(\mathcal{R})} t'$ for some rule $\ell \to r \in T(\mathcal{R})$, some term $\hat{t} \in \mathcal{T}(\Sigma_{\cup\text{ctrl}}, \mathcal{V})$, and some position $p \in \text{Pos}(t)$, where $t|_p = \ell\sigma$ and $\hat{t} = t[r\sigma]_p$ for some $\sigma \in \text{SUB}(\Sigma_{\cup\text{ctrl}}, \mathcal{V})$. Note that for all rules $\ell' \to r' \in T(\mathcal{R})$, both $\text{root}(\ell) \in \Sigma_{\text{ctrl}}$ and $\text{root}(r) \in \Sigma_{\text{ctrl}}$ hold. Therefore, by Corollary 7.2.31, $\text{drop}(t) = \text{drop}(t[\text{drop}(t|_p)]_p) = \text{drop}(t[\text{drop}(\ell\sigma)]_p) = \text{drop}(t[c]_p)$ and

$\mathrm{drop}(\hat{t}) = \mathrm{drop}(\hat{t}[\mathrm{drop}(\hat{t}|_p)]_p) = \mathrm{drop}(t[\mathrm{drop}(r\sigma)]_p) = \mathrm{drop}(t[c]_p)$, which proves that $\mathrm{drop}(t) = \mathrm{drop}(\hat{t})$. Together with the induction hypothesis, which shows that $\mathrm{drop}(\hat{t}) = \mathrm{drop}(t')$, this completes the proof. □

Using the above, it can now be shown that a reduction in the transformed TRS which transforms a down symbol into an up symbol corresponds to an outermost step in the original TRS.

Lemma 7.2.33. *If* $\mathsf{down}(t) \to^*_{T(\mathcal{R})} \mathsf{up}(t')$ *for two terms* $t, t' \in \mathcal{T}(\Sigma_{\cup\mathrm{ctrl}}, \mathcal{V})$*, then* $\mathrm{drop}(t) \overset{o}{\to}_{\mathcal{R}} \mathrm{drop}(t')$.

Proof. Let $\mathsf{down}(t) \to^*_{T(\mathcal{R})} \mathsf{up}(t')$ with $t, t' \in \mathcal{T}(\Sigma_{\cup\mathrm{ctrl}}, \mathcal{V})$. The length of this reduction cannot be 0, since $\mathsf{down} \neq \mathsf{up}$. Thus, the length of the reduction must at least be 1. Induction on this length is performed.

In case the reduction is of the form $\mathsf{down}(t) \overset{>\epsilon}{\to}{}^*_{T(\mathcal{R})} \mathsf{down}(\hat{t}) \to_{\mathsf{down}(\ell)\to\mathsf{up}(r),\epsilon} \mathsf{up}(\hat{t}') \to^*_{T_{\mathcal{R}}} \mathsf{up}(t')$ (where $\overset{>\epsilon}{\to}$ denotes steps below the root), then $\hat{t} = \ell\sigma$ and $\hat{t}' = r\sigma$ for some substitution $\sigma \in \mathrm{SUB}(\Sigma_{\cup\mathrm{ctrl}}, \mathcal{V})$. Since $\ell, r \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathrm{drop}(\ell) = \ell$ and $\mathrm{drop}(r) = r$ hold and furthermore, due to Corollary 7.2.30, $\mathrm{drop}(\ell\sigma) = \ell\,\mathrm{drop}(\sigma)$ and $\mathrm{drop}(r\sigma) = r\,\mathrm{drop}(\sigma)$. Thus, by Lemma 7.2.32, $\mathrm{drop}(t) = \mathrm{drop}(\hat{t}) = \mathrm{drop}(\ell\sigma) = \ell\,\mathrm{drop}(\sigma) \overset{o}{\to}_{\ell\to r,\epsilon} r\,\mathrm{drop}(\sigma) = \mathrm{drop}(r\sigma) = \mathrm{drop}(t')$ is an outermost step w.r.t. $\mathcal{R}$. This step is at the root position, hence trivially outermost. Note that this case always applies if the length of the reduction is 1, hence this also proves the base case of the induction.

Otherwise, the reduction must have the following shape for some terms $u = f(u_1, \dots, u_n) \in S_L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$, $v = f^\natural(\mathsf{block}(u_1), \dots, \mathsf{down}(u_i), \dots, \mathsf{block}(u_n))$, and $t_j, t'_j, t''_j \in \mathcal{T}(\Sigma_{\cup\mathrm{ctrl}}, \mathcal{V})$ for $1 \leq j \leq n$:

$$\begin{array}{lll}
\mathsf{down}(t) & \overset{>\epsilon}{\to}{}^*_{T(\mathcal{R})} & \mathsf{down}(\hat{t}) \\
 & = & \mathsf{down}(f(t_1, \dots, t_n)) \\
 & \to_{\mathsf{down}(u)\to v,\epsilon} & f^\natural(\mathsf{block}(t_1), \dots, \mathsf{down}(t_i), \dots, \mathsf{block}(t_n)) \\
 & \to^*_{T(\mathcal{R})} & f^\natural(\mathsf{block}(t''_1), \dots, \mathsf{up}(t''_i), \dots, \mathsf{block}(t''_n)) \\
 & \to^*_{T(\mathcal{R})} & f^\natural(\mathsf{block}(t'_1), \dots, \mathsf{up}(t'_i), \dots, \mathsf{block}(t'_n)) \\
 & \to_{T(\mathcal{R}),\epsilon} & \mathsf{up}(f(t'_1, \dots, t'_n) \\
 & \to^*_{T(\mathcal{R})} & \mathsf{up}(t')
\end{array}$$

The reduction must contain the term $f^\natural(\mathsf{block}(t''_1), \dots, \mathsf{up}(t''_i), \dots, \mathsf{block}(t''_n))$, as otherwise the symbol up could never appear at the root position. Hence, the reduction $\mathsf{down}(t_i) \to \mathsf{up}(t''_i)$ occurs and must be shorter than the original reduction. This allows to apply the induction hypothesis, which shows that $\mathrm{drop}(t_i) \overset{o}{\to} \mathrm{drop}(t''_i)$. Furthermore, the symbols block and up are constructors of the TRS $T(\mathcal{R})$, thus $t''_j \to^*_{T(\mathcal{R})} t'_j$ for all $1 \leq j \leq n$ and $t_j \to^*_{T(\mathcal{R})} t''_j$ for all $1 \leq j \leq n$ with $j \neq i$. This allows to reorder the reductions, since they are all on independent positions and below variables of the rule $f^\natural(\mathsf{block}(x_1), \dots, \mathsf{up}(x_i), \dots, \mathsf{block}(x_n)) \to f(x_1, \dots, x_n) \in T(\mathcal{R})$:

$$\begin{array}{lll}
\mathsf{down}(t) & \overset{>\epsilon}{\to}{}^*_{T(\mathcal{R})} & \mathsf{down}(f(t_1, \dots, t_n)) \\
 & \to_{\mathsf{down}(u)\to v,\epsilon} & f^\natural(\mathsf{block}(t_1), \dots, \mathsf{down}(t_i), \dots, \mathsf{block}(t_n)) \\
 & \to^*_{T(\mathcal{R})} & f^\natural(\mathsf{block}(t_1), \dots, \mathsf{up}(t''_i), \dots, \mathsf{block}(t_n)) \\
 & \to_{T(\mathcal{R})} & \mathsf{up}(f(t_1, \dots, t''_i, \dots, t_n)) \\
 & \to^*_{T(\mathcal{R})} & \mathsf{up}(f(t'_1, \dots, t'_n) \\
 & \to^*_{T(\mathcal{R})} & \mathsf{up}(t')
\end{array}$$

Here, Lemma 7.2.32 shows that $\mathrm{drop}(t) = \mathrm{drop}(\hat{t}) = \mathrm{drop}(f(t_1, \dots, t_n)) = f(\mathrm{drop}(t_1), \dots, \mathrm{drop}(t_n))$. Furthermore, since $u \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathrm{drop}(\hat{t}) = \mathrm{drop}(u\tau) = u\ \mathrm{drop}(\tau)$ according to Corollary 7.2.30 for some substitution $\tau \in \mathrm{SUB}(\Sigma_{\cup\mathrm{ctrl}}, \mathcal{V})$. Since $u \in S_L$, the definition of anti-matching implies that $\mathrm{drop}(\hat{t}) = u\ \mathrm{drop}(\tau) \neq \ell'\tau'$ for all rules $\ell' \to r' \in \mathcal{R}$ and all substitutions $\tau' \in \mathrm{SUB}(\Sigma, \mathcal{V})$. This shows that the step $\mathrm{drop}(t_i) \overset{o}{\to} \mathrm{drop}(t_i'')$ is also outermost when applying Lemma 7.2.32 to the above reordered reduction, which gives the following outermost reduction:

$$\begin{array}{rl}
\mathrm{drop}(t) = & f(\mathrm{drop}(t_1), \dots, \mathrm{drop}(t_n)) \\
\overset{o}{\to}_{\mathcal{R}} & f(\mathrm{drop}(t_1), \dots, \mathrm{drop}(t_i''), \dots, \mathrm{drop}(t_n)) \\
= & f(\mathrm{drop}(t_1'), \dots, \mathrm{drop}(t_i'), \dots, \mathrm{drop}(t_n')) \\
= & \mathrm{drop}(f(t_1', \dots, t_n')) \\
= & \mathrm{drop}(t')
\end{array}$$

Thereby, the lemma has been proven. □

Since outermost ground termination is considered, a way to obtain ground terms from arbitrary terms is required. For this purpose the function gnd is introduced.

Definition 7.2.34. Let $c \in \Sigma$ be a constant. For a term $t \in \mathcal{T}(\Sigma_{\cup\mathrm{ctrl}}, \mathcal{V})$ the term $\mathrm{gnd}(t) \in \mathcal{T}(\Sigma_{\cup\mathrm{ctrl}})$ is defined as $\mathrm{gnd}(x) = c$ for all $x \in \mathcal{V}$, and $\mathrm{gnd}(f(t_1, \dots, t_n)) = f(\mathrm{gnd}(t_1), \dots, \mathrm{gnd}(t_n))$ for all $f \in \Sigma_{\cup\mathrm{ctrl}}$ with $\mathsf{ar}(f) = n$ and terms $t_1, \dots, t_n \in \mathcal{T}(\Sigma_{\cup\mathrm{ctrl}}, \mathcal{V})$.

The function gnd is extended to substitutions $\sigma \in \mathrm{SUB}(\Sigma_{\cup\mathrm{ctrl}}, \mathcal{V})$ by defining $\mathrm{gnd}(\sigma)\,(x) = \mathrm{gnd}(\sigma(x))$ for all $x \in \mathcal{V}$.

Finally, completeness of the transformation T, as stated in Theorem 7.2.28, can be proven.

Proof of Theorem 7.2.28. Assume that $T(\mathcal{R})$ is not terminating. Then an infinite reduction $t_1 \to_{T(\mathcal{R})} t_2 \to_{T(\mathcal{R})} \dots$ exists. Aoto's property [Aot01] will be applied, which states that termination of a sorted and an unsorted TRS coincide if all variables are of the same sort. For this purpose, three sorts are considered, called 0, 1, and 2. These are assigned to the function symbols as follows: All variables have sort 0, the symbol top has sort mapping $1 \to 2$, the symbols down, up, and block have sort mapping $0 \to 1$, and for every $f \in \Sigma$ with $\mathsf{ar}(f) = n$ the sort mapping of f is $0^n \to 0$ and that of $f^\natural$ is $1^n \to 1$. All rules of the TRS $T(\mathcal{R})$ are compatible with these sorts, cf. Definition 7.2.4.

Thus, there is also an infinite reduction $t_1' \to_{T(\mathcal{R})} t_2' \to_{T(\mathcal{R})} \dots$ of well-sorted terms t_i'. Especially, it can be observed that $\mathrm{root}(t_i'|_p) \neq \mathsf{top}$ for all $1 \leq i \leq n$ and $p \in \mathrm{Pos}(t_i')$ such that $p \neq \epsilon$ and $t_i'|_p \notin \mathcal{V}$, otherwise the term would not be well-sorted.

Assume that in the infinite reduction, $\mathrm{root}(t_1') \neq \mathsf{top}$. This implies that $\mathrm{root}(t_i') \neq \mathsf{top}$ for all $i \geq 1$, due to the sort assignment. Hence, also for $\overline{T(\mathcal{R})} = T(\mathcal{R}) \setminus \{\mathsf{top}(\mathsf{up}(x)) \to \mathsf{top}(\mathsf{down}(x))\}$ it holds that $t_1' \to_{\overline{T(\mathcal{R})}} t_2' \to_{\overline{T(\mathcal{R})}} \dots$ is an infinite reduction. Consider the lexicographic path order $\succ_{\mathrm{lpo}}$ with the precedence $\mathsf{down} \sqsupset f^\natural \sqsupset \mathsf{up} \sqsupset f \sqsupset \mathsf{block}$ for all $f \in \Sigma$. Then, $\mathsf{down}(\ell) \succ_{\mathrm{lpo}} \mathsf{up}(r)$, since $r \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathsf{down}(f(t_1, \dots, t_n)) \succ_{\mathrm{lpo}} f^\natural(\mathsf{block}(t_1), \dots, \mathsf{down}(t_i), \dots, \mathsf{block}(t_n))$ for all $f(t_1, \dots, t_n) \in S_L$ and all argument positions $1 \leq i \leq n = \mathsf{ar}(f)$, and $f^\natural(\mathsf{block}(x_1), \dots, \mathsf{up}(x_i), \dots, \mathsf{block}(x_n)) \succ_{\mathrm{lpo}} \mathsf{up}(f(x_1, \dots, x_n))$ for all $f \in \Sigma$ and $1 \leq i \leq \mathsf{ar}(f) = n$. This shows that $\to_{\overline{T(\mathcal{R})}} \subseteq \succ_{\mathrm{lpo}}$, which contradicts non-termination of $\overline{T(\mathcal{R})}$.

Thus, it can be concluded that $\mathrm{root}(t'_i) = \mathsf{top}$ for all $i \geq 1$. Also, there must be an infinite number of reduction steps at the root position, otherwise a term $\mathsf{top}(\hat{t})$ would exist for which $\hat{t}$ would start an infinite reduction. This cannot occur, as was just shown. The only rule that can be applied at the root is $\mathsf{top}(\mathsf{up}(x)) \to \mathsf{top}(\mathsf{down}(x))$. Thus, the infinite reduction must contain a reduction of the shape $\mathsf{top}(\mathsf{up}(\tilde{t}_1)) \to_{T(\mathcal{R}),\epsilon} \mathsf{top}(\mathsf{down}(\tilde{t}_1)) \to^*_{T(\mathcal{R})} \mathsf{top}(\mathsf{up}(\tilde{t}_2)) \to_{T(\mathcal{R}),\epsilon} \mathsf{top}(\mathsf{down}(\tilde{t}_2)) \to^*_{T(\mathcal{R})} \ldots$. When replacing the terms $\tilde{t}_j$ with $\mathrm{gnd}(\tilde{t}_j)$ for all $j \geq 1$ then the infinite reduction is still possible. Hence, repeated application of Lemma 7.2.33 gives rise to the infinite outermost ground reduction

$$\mathrm{drop}(\mathrm{gnd}(\tilde{t}_1)) \xrightarrow{o}_{\mathcal{R}} \mathrm{drop}(\mathrm{gnd}(\tilde{t}_2)) \xrightarrow{o}_{\mathcal{R}} \ldots,$$

which proves the theorem. □

Implementation and Experiments

The transformations described above have been implemented in a tool called TrafO.[1] Even though the construction of the anti-matching set S_L can certainly be improved, the complete transformation only takes a neglegible amount of time for all of the following examples.

The implementation allows for a number of different variants of the transformation to be used. The above only presented one of these, which proved to be the most effective. In detail, one can choose whether or not to add the blocking symbol block when the symbol down descends into a term that is not matched by a left-hand side of the original term rewrite system. Also, it can be chosen whether a symbol $f \in \Sigma$ should be rewritten to a marked version $f^\natural$ of that symbol or not when descending into terms from S_L. As a last option, one can also use a modified version of the rules for the up symbol that explicitly match terms from S_L, however this modification proved itself not to be effective.

The transformed system is then used as input for the termination provers Jambox [End], TTT2 [KSZM], and AProVE [GSKT06], which were the strongest tools of the 2007 termination competition in the TRS category [MZ07]. The reason why multiple tools were used was that the transformation turned out to produce rewrite systems for which sometimes one tool succeeded in proving termination of the transformed TRS, while at least one of the other tools was unable to do so.

Below some examples are presented. First, it shall be shown that Example 7.2.2 really is outermost ground terminating, as was claimed above. When this example is transformed, the following TRS is created:

Example 7.2.35 (Transformation of Example 7.2.2)**.**

$$\begin{array}{rclcrcl}
\mathsf{top}(\mathsf{up}(x)) & \to & \mathsf{top}(\mathsf{down}(x)) & & \mathsf{down}(\mathsf{b}) & \to & \mathsf{up}(\mathsf{a}) \\
\mathsf{down}(\mathsf{f}(x,\mathsf{a})) & \to & \mathsf{up}(\mathsf{f}(x,\mathsf{b})) & & \mathsf{down}(\mathsf{f}(\mathsf{a},x)) & \to & \mathsf{up}(\mathsf{a}) \\
\mathsf{f}^\natural(\mathsf{block}(x),\mathsf{up}(y)) & \to & \mathsf{up}(\mathsf{f}(x,y)) & & \mathsf{down}(\mathsf{f}(\mathsf{b},x)) & \to & \mathsf{up}(\mathsf{a}) \\
\mathsf{f}^\natural(\mathsf{up}(x),\mathsf{block}(y)) & \to & \mathsf{up}(\mathsf{f}(x,y)) & & \mathsf{down}(\mathsf{f}(\mathsf{f}(x,y),z)) & \to & \mathsf{up}(\mathsf{a})
\end{array}$$

It can be observed that in the transformed TRS there are no rules that allow the symbol down to descend into a term. This is the case because for the anti-matching set S_L it holds that $S_L = \{\mathsf{a}\}$, such that no rules are created for the symbol down.

[1]This tool is available at `http://www.win.tue.nl/~mraffels/trafo.html`

The transformed TRS can easily be shown terminating within a short amount of time by all of the considered termination tools. For the next example, this is not the case anymore.

Example 7.2.36.

$$\mathsf{a} \;\to\; \mathsf{f}(\mathsf{a}) \qquad\qquad \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{f}(x))))) \;\to\; \mathsf{b}$$

Both AProVE and TTT2 can show termination of the transformed TRS, while Jambox fails to do so. What is also interesting is that TTT2 uses RFC Match Bounds to show this, whereas AProVE uses only Dependency Pairs and a large number of rewriting steps, but is able to find this proof much faster than TTT2.

The next example proved to be rather difficult for all of the considered tools. It is similar to the kind of problems generated by the technique proving productivity presented in Section 7.1, however it only generates an overflow symbol whenever the first argument is at least one.

Example 7.2.37.

$$\mathsf{from}(x) \;\to\; x : \mathsf{from}(\mathsf{s}(x)) \qquad\qquad \mathsf{s}(x) : xs \;\to\; \mathsf{overflow}$$

This example could only be proven terminating by the tool Jambox, both AProVE and TTT2 failed. However, the techniques used by Jambox to prove termination, namely semantic labelling and polynomial orders, are also implemented in both of the other tools. Hence, this clearly shows that proving termination is also strongly dependent on heuristics and/or search encodings.

In the examples considered so far, it was the case that always the right-hand side of the rule causing outermost ground termination was a ground term. This is different in the next example.

Example 7.2.38.

$$\mathsf{f}(\mathsf{f}(\mathsf{g}(x))) \;\to\; x \qquad\qquad \mathsf{g}(\mathsf{b}) \;\to\; \mathsf{f}(\mathsf{g}(\mathsf{b}))$$

The transformed TRS can be shown terminating by the tools TTT2 and Jambox, while AProVE fails.

In the example below, the right-hand sides are not always either growing or detecting a term that has grown too large.

Example 7.2.39.

$$\begin{array}{rcl} \mathsf{f}(\mathsf{f}(x,y),z) & \to & \mathsf{c} \\ \mathsf{f}(x,\mathsf{f}(y,z)) & \to & \mathsf{f}(\mathsf{f}(x,y),z) \\ \mathsf{a} & \to & \mathsf{f}(\mathsf{a},\mathsf{a}) \end{array}$$

For this example, termination of the transformed TRS can be shown terminating by both AProVE and Jambox, while TTT2 fails to show termination. If the first rule is changed to $\mathsf{f}(\mathsf{f}(x,y),z) \to \mathsf{f}(\mathsf{c},x)$, then only AProVE can show the transformed TRS to be terminating.

Next, the approach shall be compared with the tool Cariboo [FGK02, GK09], which uses a stand-alone approach to prove outermost termination. It is distributed with 6 examples of outermost termination. Of these 6 examples, 5 are left-linear and therefore they can be directly handled by the approach presented in this thesis. For all of these examples, outermost ground termination can be shown using Jambox as termination prover. The last example shall be considered in more detail below.

Example 7.2.40 (Outermost Example 6)**.**

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{f}(x,x) & \to & \mathsf{f}(\mathsf{i}(x),\mathsf{g}(\mathsf{g}(x))) & \mathsf{f}(x,\mathsf{i}(\mathsf{g}(x))) & \to & \mathsf{a} \\ \mathsf{f}(x,y) & \to & x & \mathsf{f}(x,\mathsf{i}(x)) & \to & \mathsf{f}(x,x) \\ \mathsf{g}(x) & \to & \mathsf{i}(x) & & & \end{array}$$

As can be seen above, this example has non-linear left-hand sides for the function symbol f. However, these left-hand sides are all instances of the left-hand side $\mathsf{f}(x,y)$, which makes this TRS quasi left-linear. Hence, only the set $L = \{\mathsf{f}(x,y),\mathsf{g}(x)\}$ of linear terms has to be considered, from which S_L is computed to be $S_L = \{\mathsf{a},\mathsf{i}(x)\}$. Using this anti-matching set, the transformation yields a finite TRS whose termination can be proven using any of the three considered tools.

Finally, the strength of the presented approach shall be compared against that of Cariboo. The following example is non-terminating for normal rewriting, since already the rule $\mathsf{h}(x) \to \mathsf{f}(\mathsf{h}(x))$ allows an infinite reduction.

Example 7.2.41.

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{f}(\mathsf{h}(x)) & \to & \mathsf{f}(\mathsf{i}(x)) & \mathsf{h}(x) & \to & \mathsf{f}(\mathsf{h}(x)) \\ \mathsf{f}(\mathsf{i}(x)) & \to & x & \mathsf{i}(x) & \to & \mathsf{h}(x) \end{array}$$

Cariboo is unable to prove outermost ground termination of the above TRS, while the transformed TRS $T(\mathcal{R})$ can be proven terminating by all considered tools. Also Example 7.2.38 and both variants of Example 7.2.39 cannot be proven outermost ground terminating by Cariboo.

There are also examples where Cariboo succeeds, whereas the presented transformational approach fails. First of all, Cariboo can also handle examples that are not quasi left-linear, while the transformation is not applicable in this case, since it would produce an infinite transformed TRS. But there are also quasi left-linear examples where Cariboo can prove outermost ground termination, but none of the considered tools can prove termination of the transformed TRS. Such an example is given below.

Example 7.2.42.

$$\mathsf{from}(x) \to x : \mathsf{from}(\mathsf{s}(x)) \qquad \mathsf{s}(\mathsf{s}(x)) : xs \to \mathsf{overflow}$$

This example can be shown terminating by Cariboo, whereas for all termination provers the transformed TRS is too hard. Please note that this is only a slightly modified version of Example 7.2.37, where instead of one s symbol now two such symbols are required.

In the termination competition of 2008 [Wal09], which included the outermost category, a combination of the presented transformations T and T_3, using both Jambox [End] and AProVE [GSKT06] as termination provers, was competing with an extension of the tool Jambox [End] by the context-sensitive transformation presented in [EH09], with the tool AProVE [GSKT06] that used both the transformation T presented here and the transformation presented in [Thi09], and the tool TTT2 [KSZM] which only proved outermost non-termination according to [TS09]. The results show that in general, for proving outermost termination, the context-sensitive approach of [EH09] is strongest, proving 72 of the total 291 considered examples outermost terminating. Using the transformation presented in this section, 46 examples could be proven outermost terminating, while the combination used in AProVE could prove

27 examples outermost terminating. It should however be remarked that there were examples for which the transformational approach presented here could be used to successfully prove outermost termination, whereas this was not possible with any of the competing automated approaches.

For proving outermost non-termination, the approach implemented in TTT2 outperformed all other techniques, proving a total of 158 examples to be outermost non-terminating. But there were also examples for which the transformation of [Thi09], implemented in the tool AProVE, was the only approach that could prove outermost non-termination. In total, AProVE could show outermost non-termination of 37 examples. Using the transformation T, which was presented here, outermost non-termination of 30 examples could be proven.

7.3 Summary

This chapter presented in Section 7.1 an alternative approach to prove productivity of orthogonal specifications by balanced outermost termination. The work presented is an extension of [ZR10b], where only stream specifications were considered. Instead, here also other data structures can be treated that fit into the format of orthogonal strictly proper specifications. The restrictions imposed onto these specifications are stricter than those considered in Chapter 6, here also the data arguments of constructors are required to be variables on left-hand sides of rules. This however, as was shown in the examples, can often be achieved by unfolding. Balanced outermost termination is similar to the outermost-fair reductions considered in the previous chapter, as they also require that no outermost redexes survive infintely long. However, in contrast to outermost-fair reductions, an infinite balanced outermost reduction only consists of outermost steps, which is not necessarily the case for an infinite outermost-fair reduction.

For the special case where no data rules exist and where every defined structure symbol has at most one structure argument, balanced outermost termination and outermost termination were shown to coincide. Even if there are more structure arguments, outermost termination will imply balanced outermost termination, however the contrary is not true anymore for these rewrite systems. This allows to use automated tools for proving outermost termination to check productivity. Note however that this technique is limited to orthogonal specifications. It was argued in the previous chapter that for the verification of cells, non-orthogonal specifications should be allowed to be able to express arbitrary input sequences. Thus, the technique presented in Section 7.1 can only be applied to check stabilization of cells for a limited set of input patterns. For this purpose, the corresponding specification is extended with a specification of the considered input patterns, the transformation is applied to this extended specification, and finally outermost termination of the transformed system is checked.

A technique to prove outermost termination was presented in Section 7.2. It works by transforming an outermost termination problem into a standard termination problem, which can then be treated by existing tools. For the transformation to result in a finite TRS, the original TRS whose outermost termination shall be analyzed must be quasi left-linear. This requires that all left-hand sides of the rules are an instance of some linear left-hand side. Then, it could be shown that a finite anti-matching set of terms can be computed, which matches exactly those ground terms not matched by any of the rules. In the basic transformation, this set is used to descend into terms whenever no rule can be applied above the current position. This transformation was furthermore proven to be complete, which is an extension over the original work presented in [RZ09]. Additionally, the section presented two transformations that are easier to check but are incomplete, i.e., they can only prove outermost termination, but not disprove it.

Section 7.2's technique to prove outermost termination was the first one that used existing termination provers. Previously, only the tool Cariboo [FGK02, GK09] existed that used a stand-alone approach to prove outermost termination. Therefore, this tool cannot make use of the tremendous improvements that have been made and still are being made in the area of termination analysis. Since the transformational technique of Section 7.2 was presented in [RZ09], several other techniques to prove outermost termination have been developed. The technique of Thiemann [Thi09] also uses a transformation of the outermost termination problem, where an innermost termination problem is created. Endrullis and Hendriks present in [EH09] a transformation from outermost termination to context-sensitive termination. In the termination competition of 2008 that included the outermost termination category [Wal09] this approach turned out to be the most powerful, proving the most examples outermost terminating. However, it should be remarked that the transformational approach presented in this thesis was able to prove outermost termination of examples for which all other approaches failed to do so.

For disproving outermost termination, direct techniques such as [TS09] have been developed. The results of the termination competition show that such an approach outperforms all other transformational approaches. Thus, it would be interesting to investigate whether also for proving outermost (ground) termination such direct approaches exist.

Another interesting topic for future research is to investigate whether balanced outermost termination can be checked automatically, which is introduced in Section 7.1 to determine productivity. At the moment, only techniques for checking general outermost termination are available. But it is often the case that for productive specifications containing symbols with more than one structure argument, such as the specification in Example 7.1.4 of the Thue Morse stream containing the function zip, outermost termination does not hold.

Chapter 8

Conclusion

This thesis presented techniques for the verification of cell libraries, which are used to implement a larger chip design from smaller standard components, called cells. Such cell libraries are usually provided by external parties, hence they should be verified to ensure that the chip design works as intended.

To aid in the development of larger designs, cell libraries contain a number of different views on the cells contained. These views describe different aspects of the cell, such as a functional simulation description, a transistor netlist description, different timing descriptions, layout information, etc. Hence, it is vital that these views correspond, i.e., that they describe the same behavior.

In this thesis, most emphasis was put on functional descriptions of cells. Chapter 3 presented a technique to verify that simulation descriptions in the commonly used subset VeriCell of the standardized language Verilog [IEE06] and the transistor netlist descriptions, usually given as SPICE netlists [NP73], implement the same functionality. There, it was discovered that even at this rather detailed level of description, non-determinism exists which can have an influence on the final result of a computation. For Verilog, the reason was found to be the User-Defined Primitives (UDPs), which are used to implement state-holding elements, i.e., memories.

Such non-determinism was investigated further in Chapter 4, both for Verilog and transistor netlists. In both cases, an efficient technique was developed to automatically identify cases where the result of a computation can differ due to this inherent non-determinism that results from the order of applying input changes. When the functional behavior is not affected by this order, the non-determinism can be used to optimize other design goals, such as for example power consumption. This was presented in Section 4.3, where the non-determinism analysis was extended by also taking an abstract measure of power into account. Then, one can enforce from functionally equivalent orders the order that consumes the minimal amount of power without altering the overall behavior of the cell. Also, during power characterization, the non-determinism analysis can be of use. There, the goal is to measure the concrete power consumed by different orders of applying inputs, in different states of the cell. Since this suffers from combinatorial explosion, the non-determinism analysis can be used to identify situations in which the functionality and the abstract power consumption do not differ, hence for such situations it is sufficient to only consider one of these equivalent orders.

Since non-determinism will usually be present in cells that implement useful functions, there exists further information that indicates the legal input of cells. Such information is to be respected by the environment to guarantee that the cell behaves as expected. The information about legal input is given by means of timing checks, describing time windows in which certain events (specific transitions of input signals) must not occur. Chapter 5 therefore extended the non-determinism analysis techniques presented in Chapter 4 to also take this information into account. In this way, it can be checked that a cell always behaves deterministically as long as the timing checks are respected by the environment. In the other direction, a situation in which non-determinism can occur is often an indication that a timing check is missing. Thus, the non-determinism analysis can be used to get an idea of the required timing checks that need to be added. Another timing specification that was considered in Chapter 5 are module paths, which are also known as timing arcs or delay arcs. Such a module path gives a delay that it takes an input change to propagate through the cell to an output. This of course is dependent on the functionality of the cell, thus a technique was presented that checked whether some specified module paths can actually occur in the functional implementation. Furthermore, a technique was presented to enumerate for a given cell all possible module paths.

For some of the analyses a next state function is required, i.e., a function that computes from the current state values and some input values the values in the next state. In a hardware implementation however, such a function is not immediate, instead numerous different functions are combined and re-evaluated a number of times until they (hopefully) stabilize in a next state. To analyze whether such a stable next state will always be reached, productivity analysis can be used. This notion is investigated in the setting of term rewrite systems, where productivity has been studied before. Productivity is the property that from a desired infinite object, any finite prefix can be computed by some given rules. When applying this to the hardware setting, it should be the case that when viewing wires as infinite streams of values, then any finite prefix (representing the history of the wire up to the current point in time) should be computable. A technique to analyze productivity based on context-sensitive termination was presented in Chapter 6. It creates from a given term rewrite system automatically another term rewrite system for which context-sensitive termination is analyzed. If context-sensitive termination can be shown, then it implies that the initial term rewrite system is productive. Since cells are interacting with the environment by means of input and output signals, these signals have to be abstracted. Hence, the idea is to allow arbitrary sequences of input values, which guarantees that the cell computes a stable next state in all possible environments. This abstraction required giving up the restriction to orthogonal specifications, which disallows such non-determinism. In the literature, only orthogonal specifications have been considered before, since it allows to also conclude a specification to describe a unique behavior. But in the analysis of hardware, presented for an example cell in Section 6.4, the two rules $\mathsf{rand} \to 0 : \mathsf{rand}$ and $\mathsf{rand} \to 1 : \mathsf{rand}$ already make the specification non-orthogonal. These rules implement the abstraction of input values to an arbitrary stream of Boolean input values and therefore non-orthogonal specifications need to be allowed. With the extension to non-orthogonal specifications it was illustrated that productivity, and hence stabilization, of hardware cells can be established for term rewrite systems corresponding to the implementation of cells.

Another technique for proving productivity, restricted to orthogonal specifications only, was presented in Chapter 7. There, the link with outermost rewriting, which was already used in Chapter 6, is made even more explicit by considering balanced outermost termination. It was proven that balanced outermost termination of an extended term rewrite system implies productivity of an initial term rewrite system.

For balanced outermost termination, no automated tools exist. However, there are special cases where outermost termination and balanced outermost termination coincide. Furthermore, when outermost termination can be proven, then balanced outermost termination also holds. To prove outermost termination, Chapter 7 also presented a transformation that allows to use termination provers for standard rewriting to prove outermost termination. This transformation was proven to be both sound and complete, i.e., termination of the transformed term rewrite system allows to conclude outermost termination of the initial system and from non-termination of the transformed term rewrite system it can be inferred that the initial term rewrite system is not terminating for outermost rewriting.

All of the techniques described in this thesis were implemented and evaluated experimentally on industrial cell libraries. Furthermore, the results were presented in a number of scientific publications [RMR$^+$09, RRM09, RZ09, RMS10, RMZ10, ZR10a, ZR10b, Raf11, RM11, RMZ11].

Future Work

A number of different views that describe cells contained in a cell library have been considered in this thesis. For these views it was verified that they correspond to each other, i.e., that they describe the same common behavior. However, there still are views that were not taken into account in this thesis and combinations that were not considered. Hence, it would be interesting to also investigate these, to be able to gain even more confidence that all views of a cell library describe the same cells. An important example are the different layout views, which describe, at different levels of detail, lithographic masks used for production. It is already common practice to extract from a detailed layout view a transistor netlist description which is then compared with the transistor netlist contained in the cell library. However, also the different layout views should be checked whether they correspond to each other or not. If for example a layout view that is used by a place-and-route tool to introduce connections between different cells in a larger chip design does not correspond with the other layout views, then the final chip design will be incorrect and non-functional.

The analysis techniques presented in this thesis made use of some links between hardware verification and term rewriting. One is the verification of stabilization described above. But also the non-determinism analysis is centered around a property that is known from term rewriting, called the commuting diamond property. There are also connections between these two fields that go in the other direction, one example is the use of SAT solving, that was initially developed to verify hardware circuits, for the purpose of searching well-founded orders. For this reason, it should be investigated whether more connections between these fields exist that allow such an exchange of knowledge.

Bibliography

[AEF$^+$08] B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving Context-Sensitive Dependency Pairs. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Artificial Intelligence*, pages 636–651. Springer-Verlag, 2008.

[AG00] T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[AGL06] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-Sensitive Dependency Pairs. In *Proceedings of the 26th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, volume 4337 of *Lecture Notes in Computer Science*, pages 297–308. Springer-Verlag, 2006.

[Aot01] T. Aoto. Solution to the Problem of Zantema on a Persistent Property of Term Rewrite Systems. *Journal of Functional and Logic Programming*, 2001(11):1–20, 2001.

[BBR96] A. Bogliolo, L. Benini, and B. Ricco. Power Estimation of Cell-Based CMOS Circuits. In *Proceedings of the 33rd annual Design Automation Conference (DAC'96)*, pages 433–438. ACM Press, 1996.

[BBR10] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2010.

[Ber99] G. Berry. The Constructive Semantics of Pure Esterel, 1999. Draft version 3, available from `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bro61] A. Brocot. Calcul des Rouages par Approximation, Nouvelle Méthode. *Revue Chonométrique*, 3:186–194, 1861.

[Bry87] R. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design*, 6(4):634–649, 1987.

[CCG$^+$02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer-Verlag, 2002. See also `http://nusmv.irst.itc.it`.

[CDE$^+$03] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 2003.

[CW00] N. Calkin and H. Wilf. Recounting the Rationals. *American Mathematical Monthly*, 107(4):360–363, 2000.

[DB95] A. J. Daga and W. P. Birmingham. A Symbolic-Simulation Approach to the Timing Verification of Interacting FSMs. In *Proceedings of IEEE International Conference on Computer Design (ICCD'95)*, pages 584–589. IEEE Computer Society Press, 1995.

[Dim01] J. Dimitrov. Operational semantics for Verilog. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 161–168. IEEE Computer Society Press, 2001.

[DKMW94] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Certified Timing Verification and the Transition Delay of a Logic Circuit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(3):333–342, 1994.

[EGH$^+$07] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of Stream Definitions. In *Proceedings of the Conference on Fundamentals of Computation Theory (FCT'07)*, volume 4639 of *Lecture Notes in Computer Science*, pages 274–287. Springer-Verlag, 2007.

[EGH08] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-oblivious Stream Productivity. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Computer Science*, pages 79–96. Springer-Verlag, 2008. Web interface tool: `http://infinity.few.vu.nl/productivity/`.

[EH09] J. Endrullis and D. Hendriks. From Outermost to Context-Sensitive Rewriting. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 2009.

[EH11] J. Endrullis and D. Hendriks. Lazy Productivity via Termination. *Theoretical Computer Science*, 412(28):3203–3225, 2011. Festschrift in Honour of Jan Bergstra.

[End] J. Endrullis. Jambox 2.0e. Downloadable from `http://joerg.endrullis.de`.

[End10] J. Endrullis. *Termination and Productivity*. PhD thesis, Vrije Universiteit Amsterdam, 2010.

[FD98] K. Futatsugi and R. Diaconescu, editors. *CafeOBJ Report*. World Scientific Publishing Company, 1998.

[FGK02] O. Fissore, I. Gnaedig, and H. Kirchner. System Presentation – CARIBOO: An Induction Based Proof Tool for Termination with Strategies. In *Proceedings of the 4th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP'02)*, pages 62–73. ACM Press, 2002.

[GB10] M. Geilen and T. Basten. Kahn Process Networks and a Reactive Extension. In *Handbook of Signal Processing Systems*, pages 967–1006. Springer-Verlag, 2010.

[GK09] I. Gnaedig and H. Kirchner. Termination of Rewriting under Strategies. *ACM Transactions on Computational Logic*, 10(2):10:1–10:52, 2009.

[GM04] J. Giesl and A. Middeldorp. Transformation Techniques for Context-Sensitive Rewrite Systems. *Journal of Functional Programming*, 14:329–427, 2004.

[Gor95] M. Gordon. The semantic challenge of Verilog HDL. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 136–145. IEEE Computer Society Press, 1995.

[GRSK$^+$11] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):7:1–7:39, 2011.

[GSKT06] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proceedings of the 3rd International Joint Conference on Automatic Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Computer Science*, pages 281–286. Springer-Verlag, 2006. Web interface tool: `http://aprove.informatik.rwth-aachen.de`.

[Ham50] R. W. Hamming. Error-detecting and error-correcting codes. *Bell System Technical Journal*, XXIX(2):147–160, 1950.

[Hay00] B. Hayes. On the Teeth of Wheels. *American Scientist*, 88(4):296, 2000.

[HBJ01] Z. Huibiao, J. Bowen, and H. Jifeng. From Operational Semantics to Denotational Semantics for Verilog. In *Proceedings of the 11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *Lecture Notes in Computer Science*, pages 449–464. Springer-Verlag, 2001.

[Hin08] R. Hinze. Functional Pearl: Streams and Unique Fixed Points. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 189–200. ACM Press, 2008.

[HKC00] M. Huang, R. Kwok, and S.-P. Chan. An Empirical Algorithm for Power Analysis in Deep Submicron Electronic Designs. *VLSI Design*, 14(2):219–227, 2000.

[HMMCM06] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip. In *Proceedings of the 6th International Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 171–178. IEEE Computer Society Press, 2006.

[IEE05] IEEE Std 1364.1-2005: Verilog Register Transfer Level Synthesis. IEEE Computer Society Press, 2005.

[IEE06] IEEE Std 1364-2005: IEEE Standard for Verilog Hardware Description Language. IEEE Computer Society Press, 2006.

[IEE09] IEEE-Std 1076-2008: IEEE Standard VHDL Language Reference Manual. IEEE Computer Society Press, 2009.

[Isi08] A. Isihara. Productivity of Algorithmic Systems. In *Proceedings of the Austrian-Japanese Workshop on Symbolic Computation in Software Science (SCSS'08)*, volume 08-08 of *RISC-Linz Report*, pages 81–95, 2008.

[Isi10] A. Isihara. *Algorithmic Term Rewriting Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2010.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Information Processing*, 74:471–475, 1974.

[Kel75] R. M. Keller. A fundamental theorem of asynchronous parallel computation. In *Proceedings of the Sagamore Computer Conference*, volume 24 of *Lecture Notes in Computer Science*, pages 102–112. Springer-Verlag, 1975.

[KGG08] S. Kundu, M. K. Ganai, and R. Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *Proceedings of the 45th annual Design Automation Conference (DAC'08)*, pages 936–941. ACM Press, 2008.

[KKM07] C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-Pattern Matching. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 110–124. Springer-Verlag, 2007.

[KSZM] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2 (TTT2). Downloadable from `http://colo6-c703.uibk.ac.at/ttt2`.

[L$^+$] S. Lucas et al. μ-Term. Web interface and download: http://zenon.dsic.upv.es/muterm/.

[LM87] J.-L. Lassez and K. Marriot. Explicit Representation of Terms Defined by Counter Examples. *Journal of Automated Reasoning*, 3(3):301–317, 1987.

[Luc98] S. Lucas. Context-Sensitive Computations in Functional and Functional Logic Programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.

[Luc02] S. Lucas. Context-Sensitive Rewrite Strategies. *Information and Computation*, 178(1):294–343, 2002.

[Luc06] S. Lucas. Proving Termination of Context-Sensitive Rewriting by Transformation. *Information and Computation*, 204(12):1782–1846, 2006.

[McM97] K. McMillan. A Compositional Rule for Hardware Design Refinement. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35. Springer-Verlag, 1997. See also http://www.kenmcmil.com/smv.html.

[Men08] Mentor Graphics Corp. ModelSim 6.3g, 2008. See http://www.model.com/.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[Mou09] M. R. Mousavi. Causality in the Semantics of Esterel: Revisited. In *Proceedings of the 5th International Workshop on Structural Operational Semantics (SOS'09)*, volume 18 of *Electronic Proceedings in Theoretical Computer Science*, pages 32–45, 2009.

[MZ07] C. Marché and H. Zantema. The Termination Competition. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA'07)*, volume 4533 of *Lecture Notes in Computer Science*, pages 303–313. Springer-Verlag, 2007. See also http://www.lri.fr/~marche/termination-competition and http://termcomp.uibk.ac.at.

[Nan08] Nangate Inc. Open Cell Library v2008_10 SP1, 2008. Downloadable from http://www.nangate.com/openlibrary/.

[NP73] L. W. Nagel and D. O. Pederson. SPICE (Simulation Program with Integrated Circuit Emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, 1973.

[PBE$^+$09] M. Palla, J. Bargfrede, S. Eggersglüß, W. Anheier, and R. Drechsler. Timing Arc Based Logic Analysis for False Noise Reduction. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD'09)*, pages 225–230. ACM Press, 2009.

[PBEB07] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer-Verlag, 2007.

[Pel98] D. Peled. Ten Years of Partial Order Reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer-Verlag, 1998.

[Pey03] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[Pix92] C. Pixley. A Theory and Implementation of Sequential Hardware Equivalence. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 11(12):1469–1478, 1992.

[PJB$^+$95] M. Pandey, A. Jain, R. Bryant, D. Beatty, G. York, and S. Jain. Extraction of finite state machines from transistor netlists by symbolic simulation. In *Proceedings of the International Conference on Computer Design (ICCD'95)*, pages 596–601. IEEE Computer Society Press, 1995.

[Pra07] Pragmatic C Software Corp. GPL Cver 2.12a, 2007. Downloadable from `http://www.pragmatic-c.com/gpl-cver/`.

[PSS97] S. E. Panitz and M. Schmidt-Schauss. TEA: Automatically proving Termination of Programs in a non-strict higher order Functional Language. In *Proceedings of the 4th International Symposium on Static Analysis (SAS'97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 345–360. Springer-Verlag, 1997.

[Raf11] M. Raffelsieper. Productivity of Non-Orthogonal Term Rewrite Systems. In *Informal Proceedings of the 10th International Workshop in Reduction Strategies in Rewriting and Programming (WRS'11)*, 2011. Extended Abstract.

[RDJ96] A. Raghunathan, S. Dey, and N. K. Jha. Glitch Analysis and Reduction in Register Transfer Level Power Optimization. In *Proceedings of the 33rd annual Design Automation Conference (DAC'96)*, pages 331–336. ACM Press, 1996.

[RM11] M. Raffelsieper and M. R. Mousavi. Symbolic Power Analysis of Cell Libraries. In *Proceedings of the 16th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'11)*, volume 6959 of *Lecture Notes in Computer Science*, pages 134–148. Springer-Verlag, 2011.

[RMR$^+$09] M. Raffelsieper, M. R. Mousavi, J.-W. Roorda, C. Strolenberg, and H. Zantema. Formal Analysis of Non-Determinism in Verilog Cell Library Simulation Models. In *Proceedings of 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'09)*, volume 5825 of *Lecture Notes in Computer Science*, pages 133–148. Springer-Verlag, 2009.

[RMS10] M. Raffelsieper, M. R. Mousavi, and C. Strolenberg. Checking and Deriving Module Paths in Verilog Cell Library Descriptions. In *Proceedings of the 13th Design, Automation, and Test in Europe Conference and Exposition (DATE'10)*, pages 1506–1511. European Design and Automation Association, 2010.

[RMZ10] M. Raffelsieper, M. R. Mousavi, and H. Zantema. Order-Independence of Vector-Based Transition Systems. In *Proceedings of the 10th International Conference on Application of Concurrency to System Design (ACSD'10)*, pages 115–123. IEEE Computer Society Press, 2010.

[RMZ11] M. Raffelsieper, M. R. Mousavi, and H. Zantema. Long-Run Order-Independence of Vector-Based Transition Systems. *IET Computers & Digital Techniques*, 2011. To appear.

[RRM09] M. Raffelsieper, J.-W. Roorda, and M. R. Mousavi. Model Checking Verilog Descriptions of Cell Libraries. In *Proceedings of the Ninth International Conference on Application of Concurrency to System Design (ACSD'09)*, pages 128–137. IEEE Computer Society Press, 2009.

[RZ09] M. Raffelsieper and H. Zantema. A Transformational Approach to prove Outermost Termination Automatically. In *Proceedings of the 8th International Workshop in Reduction Strategies in Rewriting and Programming (WRS'08)*, volume 237 of *Electronic Notes in Theoretical Computer Science*, pages 3–21. Elsevier Science Publishers B. V. (North-Holland), 2009.

[Sch97] R. R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[SLL97] W.-Z. Shen, J.-Y. Lin, and J.-M. Lu. CB-Power: A Hierarchical Cell-Based Power Characterization and Estimation Environment for Static CMOS Circuits. In *Proceedings of 2nd Asia and South Pacific Design Automation Conference (ASP-DAC'97)*, pages 189–194. IEEE Computer Society Press, 1997.

[Sny08] W. Snyder. Verilator 3.681, 2008. Downloadable from `http://www.veripool.org/wiki/verilator`.

[Ste58] M. A. Stern. Ueber eine zahlentheoretische Funktion. *Journal für die reine und angewandte Mathematik*, 55:193–220, 1858.

[TdS05] O. Tardieu and R. de Simone. Loops in Esterel. *ACM Transactions on Embedded Computing Systems*, 4(4):708–750, 2005.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[Thi09] R. Thiemann. From Outermost Termination to Innermost Termination. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, volume 5404 of *Lecture Notes in Computer Science*, pages 533–545. Springer-Verlag, 2009.

[TS09] R. Thiemann and C. Sternagel. Loops under Strategies. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag, 2009.

[Tur36] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[Tur03] M. Turpin. The Dangers of Living with an X. SNUG Boston, 2003.

[vE00] C. A. J. van Eijk. Sequential Equivalence Checking Based on Structural Similarities. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 19(7):814–819, 2000.

[Wal09] J. Waldmann. Report on the Termination Competition 2008. In *Informal Proceedings of the 10th International Workshop on Termination (WST'09)*, 2009. Available from `http://www.imn.htwk-leipzig.de/~waldmann/talk/09/wst/paper.pdf`. See also `http://termcomp.uibk.ac.at`.

[Wel08] WellSpring Solutions. VeriWell 2.8.7, 2008. Downloadable from `http://sourceforge.net/projects/veriwell`.

[Wil07] S. Wilson. Icarus Verilog v0.8.6, 2007. Downloadable from `http://www.icarus.com/eda/verilog/`.

[WW98] P. Wohl and J. Waicukauski. Extracting Gate-Level Networks from Simulation Tables. In *Proceedings of the IEEE International Test Conference (TEST'98)*, pages 622–631. IEEE Computer Society Press, 1998.

[Zan08] H. Zantema. Normalization of Infinite Terms. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA'08)*, volume 5117 of *Lecture Notes in Computer Science*, pages 441–455. Springer-Verlag, 2008.

[Zan09] H. Zantema. Well-definedness of Streams by Termination. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 164–178. Springer-Verlag, 2009.

[ZR10a] H. Zantema and M. Raffelsieper. Proving Productivity in Infinite Data Structures. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA'10)*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 401–416. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.

[ZR10b] H. Zantema and M. Raffelsieper. Stream Productivity by Outermost Termination. In *Proceedings of the 9th International Workshop in Reduction Strategies in Rewriting and Programming (WRS'09)*, volume 15 of *Electronic Proceedings in Theoretical Computer Science*, 2010.

Summary

Digital electronic devices are often implemented using cell libraries to provide the basic logic elements, such as Boolean functions and on-chip memories. To be usable both during the development of chips, which is usually done in a hardware definition language, and for the final layout, which consists of lithographic masks, cells are described in multiple ways. Among these, there are multiple descriptions of the behavior of cells, for example one at the level of hardware definition languages, and another one in terms of transistors that are ultimately produced. Thus, correct functioning of the device depends also on the correctness of the cell library, requiring all views of a cell to correspond with each other.

In this thesis, techniques are presented to verify some of these correspondences in cell libraries. First, a technique is presented to check that the functional description in a hardware definition language and the transistor netlist description implement the same behavior. For this purpose, a semantics is defined for the commonly used subset of the hardware definition language Verilog. This semantics is encoded into Boolean equations, which can also be extracted from a transistor netlist. A model checker is then used to prove equivalence of these two descriptions, or to provide a counterexample showing that they are different.

Also in basic elements such as cells, there exists non-determinism reflecting internal behavior that cannot be controlled from the outside. It is however desired that such internal behavior does not lead to different externally observable behavior, i.e., to different computation results. This thesis presents a technique to efficiently check, both for hardware definition language descriptions and transistor netlist descriptions, whether non-determinism does have an effect on the observable computation or not.

Power consumption of chips has become a very important topic, especially since devices become mobile and therefore are battery powered. Thus, in order to predict and to maximize battery life, the power consumption of cells should be measured and reduced in an efficient way. To achieve these goals, this thesis also takes the power consumption into account when analyzing non-deterministic behavior. Then, on the one hand, behaviors consuming the same amount of power have to be measured only once. On the other hand, functionally equivalent computations can be forced to consume the least amount of power without affecting the externally observable behavior of the cell, for example by introducing appropriate delays.

A way to prevent externally observable non-deterministic behavior in practical hardware designs is by adding timing checks. These checks rule out certain input patterns which must not be generated by the environment of a cell. If an input pattern can be found that is not forbidden by any of the timing checks, yet allows non-deterministic behavior, then the cell's environment is not sufficiently restricted and hence this usually indicates a forgotten timing check. Therefore, the check for non-determinism is extended to also respect these timing checks and to consider only counterexamples that are not ruled out. If such a counterexample can be found, then it gives an indication what timing checks need to be added.

Because current hardware designs run at very high speeds, timing analysis of cells has become a very important issue. For this purpose, cell libraries include a description of the delay arcs present in a cell, giving an amount of time it takes for an input change to have propagated to the outputs of a cell. Also for these descriptions, it is desired that they reflect the actual behavior in the cell. On the one hand, a delay arc that never manifests itself may result in a clock frequency that is lower than necessary. On the other hand, a forgotten delay arc can cause the clock frequency being too high, impairing functioning of the final chip. To relate the functional description of a cell with its timing specification, this thesis presents techniques to check whether delay arcs are consistent with the functionality, and which list all possible delay arcs.

Computing new output values of a cell given some new input values requires all connections among the transistors in a cell to obtain stable values. Hitherto it was assumed that such a stable situation will always be reached eventually. To actually check this, a wire is abstracted into a sequence of stable values. Using this abstraction, checking whether stable situations are always reached is reduced to analyzing that an infinite sequence of such stable values exists. This is known in the term rewriting literature as productivity, the infinitary equivalent to termination. The final contribution in this thesis are techniques to automatically prove productivity. For this purpose, existing termination proving tools for term rewriting are re-used to benefit from their tremendous strength and their continuous improvements.

Curriculum Vitae

Matthias Raffelsieper was born on May 5th, 1981 in Recklinghausen, Germany. After finishing secondary school in 2000, Matthias completed the compulsory civilian service in 2001. In the same year, he started studying computer science at the RWTH Aachen University of Technology. Matthias received his Diplom in computer science, equivalent to a Master of Science in computer science, with distinction in 2007. His diploma thesis was titled “Improving efficiency and power of automated termination analysis for Haskell” and was supervised by Prof. Dr. Jürgen Giesl.

Immediately after graduation, Matthias started as a PhD student at the Eindhoven University of Technology, under the supervision of Prof. Dr. Hans Zantema and Prof. Dr. Jan Friso Groote. His research in the Valichip project focused on the verification of cell libraries, a problem identified in cooperation with the industrial partner Fenix Design Automation. Next to this, Matthias also maintained a strong interest in term rewriting. Both areas have led to a number of publications in conference proceedings and journals.

Appendix A

Nangate Open Cell Library License

The Open Cell Library is intended for use by universities, other research activities, educational programs and Si2.org members. However allowed, the Open Cell Library is not intended for commercial use. If you use the Open Cell Library for demonstration of commercial EDA tools it is required to mention, indicate that the library was developped by Nangate.

If you have questions or concerns then please contact us at openlibrary@nangate.com

The Open Cell Library is provided by Nangate under the following License:

Nangate Open Cell Library License, Version 1.0. February 20, 2008

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural