Using Enhanced Logic Programming Semantics for Extending and Optimizing Synchronous System Design

Dissertation

Vom Fachbereich Informatik der Technischen Universität Kaiserslautern zur Verleihung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.) genehmigte Dissertation

von

Marc Dahlem

Datum der wissenschaftlichen Aussprache	17.08.2021
Dekan	Prof. Dr. Jens Schmitt
Gutachter	Prof. Dr. Klaus Schneider
	Prof. Michael Mendler, PhD

D 386

Abstract

The semantics of programming languages assign a meaning to the written program syntax. Currently, the meaning of synchronous programming languages, which are especially designed to develop programs for reactive and embedded systems, is based on a formal semantics definition similar to Fitting's fixpoint semantics for logic programs. Nevertheless, it is possible to write a synchronous program code that does not evaluate to concrete values with the current semantics, which means those programs are currently seen to be *not constructive*. In the last decades, the theoretical knowledge and representation of semantics for logic programming has increased, but not all theoretical results and achievements have found their way to practice and application in system design.

This thesis, in a first part, focuses on extensions to the semantics of synchronous programming languages to an evaluation similar to a *well-founded semantics* as defined in logic programming by van Gelder, Ross and Schlipf and to the *stable model semantics* as defined by Gelfond and Lifschitz. Particularly, this allows an evaluation for some of the currently *not constructive* programs where the semantics based on Fitting's fixpoint fails. It is shown that the extension to well-founded semantics is a conservative extension of Fitting's semantics, so that the meaning for programs which were already *constructive* does not change. Finally, it is considered how one can still generate circuits that implement the considered synchronous programs with the well-founded semantics. Again, this is a conservative approach that does not modify the circuits generated by the so-far used synthesis procedures. Answer set programming and the underlying stable model semantics describe problems by constraints and the related answer set solvers give all solutions to that problem as so-called answers. This allows the formulation of searching and planning problems as well as efficient solutions without having the need to develop special and possibly error-prone algorithms for every single application. The semantics of the synchronous programming language Quartz is also extended to the *stable model semantics*. For this extension, two alternatives are discussed: First of all, a direct extension similar to the extension to well-founded semantics is discussed. Second, a transformation of synchronous programs to the available answer set programming languages is given, as this

allows to directly use answer set solvers for the synthesis and optimization of synchronous systems.

The second part of the thesis contains further examples of the use of answer set programming in system design to emphasis their benefits for system design in general. The first example is hereby the generation of *optimal/minimal interconnection-networks* which allow non-blocking connections between n sources and n targets in parallel. As a second example, the stable model semantics is used to build a complete compiler chain, which transforms a given program to an optimal assembler code (called *move code*) for the new *SCAD processor architecture* which was developed at the University of Kaiserslautern. As a final part, the lessons learned from the two examples are shown by the means of some enhancement ideas for the synchronous programming language paradigm.

Acknowledgement

First of all, I want to give special thanks to Prof. Dr. Klaus Schneider, who guided me in a perfectly helpful and constructive way through the years of my research. Not only but also in hectic phases he was able to calm me down, to recalibrate my focus and to point me out to new directions when needed. But also when having submission deadlines, urgent issues or when getting stuck, he always had time to support, to discuss, and to solve the issues. I also want to express my huge gratitude to my company Insiders Technologies GmbH and especially to Florian Morr for giving me the opportunity, support and free space to work on my research. But also I want to thank the complete MOBILE team at Insiders, who conceivably suffered the most from my absence during my research time, for their total appreciation and mental support. Further, I want to thank Prof. Michael Mendler for his review of the thesis. To all my colleagues at the university: it was a great pleasure to discuss all the different topics with you; and it wasn't just a single time that those discussions led to new ideas and input for the research... I also want to give huge thanks to my parents and to my siblings, who supported me all the years during my studies and during the time of my research. Last and most important I want to thank my lovely wife Christina for the limitless support, all the motivation and encouragement, and not least for the language review of my work.

October 3, 2021, Marc Dahlem

Contents

1. Introduction **1**
1.1. Contributions 2
1.2. Related Work 3
1.3. Outline 4

2. Background **5**
2.1. Semantics of Logic Programs: From Horn Models to Stable Models 5
2.2. Stable Model Semantics in Practice: Answer Set Programming 12
2.3. Semantics of Synchronous Languages 14

3. Extending Semantics of Synchronous Programs **23**
3.1. Well-founded Semantics 23
3.2. Stable Model Semantics 50

4. Optimized System Design with Stable Models **69**
4.1. Synthesis of Optimal Interconnection Networks 70
4.2. Optimal Code Generation for SCAD 74
4.3. Enhancing Synchronous Programs by Stable Models 92

5. Conclusions **101**

Bibliography **103**

A. Curriculum Vitae **113**

List of Figures

2.1. ABRO: module 15
2.2. ABRO: synchronous guarded actions 16

4.1. Switch configurations 70
4.2. An expression DAG with its levelized and planarized version, and the final level-planar expression DAG 76
4.3. Equation system for ABRO 79
4.4. ASP encoding of the ABRO example. 82
4.5. (a) An example basic block as DAG, (b) the same DAG as ASP-code 83
4.6. Three valid variable assignments and orderings that allow computation of the basic block in Figure 4.5 without any overhead 84
4.7. Combined DAG for Solutions 1 and 2 from Figure 4.6(a) and (b) 87
4.8. One example to minimize the amount of execution time and then find the minimal number of PUs needed to schedule the example basic block shown in Figure 4.5 88
4.9. One example out of 8088 (without symmetries: 1348) to minimize the amount of PUs with an upper bound to 4 execution steps from the example basic block shown in Figure 4.5 89
4.10. Example schedules with LSU constraints for Figure 4.5 90
4.11. (a) Average and maximum time required by SAT and ASP solvers to derive resource constrained schedules for programs of different sizes. (b) Average and maximum time taken by SMT and ASP solvers to derive time constrained schedules for programs of different sizes. 91

Chapter 1

Introduction

As the market of embedded and integrated systems is still increasing, programming languages which can be synthesized to hardware are more than ever of interest. Synchronous languages are developed to describe system reactions with parallel micro step actions that are scheduled in macro steps, like the computations of synchronous hardware circuits. As embedded systems are often used in safety-critical applications like automobiles and aircrafts, there exists a high demand for the verification of these systems to ensure a proven correct behaviour. The semantics of synchronous languages are given by formal definitions which enables the verification against given formal requirements.

The world of mathematical logic and logic programming also completely relies on formal semantics. Already in the 50th, Alfred Horn introduced a clause form which is named after him to structure and analyse logic terms [Horn51]. For such Horn clauses, unique and minimal models can be computed which was shown by Emden and Kowalski in the 70th [EmKo76]. When negation is introduced to such sets of Horn clauses, the first logic programming semantics can be defined. Such a negation was introduced by Keith Clark with the special 'negation by failure' which computes not(x) only if x cannot be proved [Clar77]. To this end, Clark also defined a completion of programs which constructs equation systems for the given logic terms. Nevertheless, Clark's semantics lacked the handling of negations with cyclic dependencies which was the base for Fitting's fixpoint semantics [Fitt85] who constructed a three-valued interpretation for logic programs. Especially the still not optimal part of negations in Fitting's fixpoint semantics was enhanced by van Gelder, Ross and Schlipf in the well-founded semantics for logic programs [GeRS91]. Gelfond and Lifschitz [GeLi88] generalized all those ideas in the stable model semantics by the observation that every model which is stable in its application has a cause for every term to hold. This stable model semantics found its application in the language of answer set programming [EiIK09; CFGI13] which is especially useful to define search and optimization problems efficiently with the help of constraints, facts, and logical rules.

However, not all knowledge about logic programming semantics has so far been transformed into the area of synchronous languages. In particular, there

are synchronous programs which evaluate some of their values to $\perp$ (unknown) with the current semantics, although this program would evaluate with the well-founded semantics or the stable model semantics to concrete values. Further, answer set programming as application of the stable model semantics contains some useful enhanced features like optimization statements or choice statements from which synchronous languages could benefit. Especially, the system design with synchronous programs could switch from a possible error-prone algorithmic and computational approach to a constraint and specification driven perspective by describing the solution's parameters instead of writing concrete algorithms. Also answer set programming can benefit from such an extension of synchronous languages, because synchronous programming languages can provide a high-level access to the theory-based and formal language of answer set programs. For instance, user-friendly statements like if-then-else, abortions, a possibility to separate programs into steps using pause statements, and the capability to define modules would be convenient for programmers who do not want to directly access the low-level description of answer set programming with the stable model semantics.

1.1. Contributions

The current semantics of synchronous languages by the example of the synchronous programming languages Quartz and Esterel are essentially equivalent to Fitting's fixpoint semantics for logic programming: If a synchronous Quartz program is *constructive*, i.e., yields concrete and unique values for all variables, the corresponding logic program evaluated with Fitting's fixpoint semantics would result in the same result and vice-versa. However, there are programs which are not constructive with the current semantics of Quartz, while concrete and unique values could be computed with the well-founded semantics.

The first contribution of this thesis is therefore the extension of the current Quartz semantics in a way that these programs can be executed according to the well-founded semantics. This approach is discussed from two different perspectives: At first, a theoretical approach on how to integrate the well-founded semantics into a synchronous program interpreter and the semantics definition via structural operation semantics (SOS rules) is given. Then, a more practical way is shown which allows one to extend programs themselves in such a way that the current synchronous interpreter computes the same result as the well-founded semantics for logic programs would do. It is proven that this approach is conservative, and especially the interpretation of constructive programs in the current semantics based on Fitting's fixpoint computation does not change their interpretation with this extension, but more programs can be given a meaning. The main advantage of this approach is that it is practical in the sense that it allows a circuit generation for the considered programs with the well-founded semantics.

As a second contribution, the further extension of the semantics of synchronous programs to the stable model semantics is discussed. It will be shown

that a direct inclusion into the interpreter or into the program itself does not give a real benefit, especially because the program interpretation for the stable model semantics is non-monotonic and therefore needs special search tools for the solutions. However, a transformation into the standardized ASP-core-2 language format will be shown, which allows to use state-of-the-art solvers and heuristics of the answer set community to evaluate, synthesize and optimize synchronous programs. This translation adds an high-level access with synchronous programming languages to the theoretical constructs of ASP.

The third contribution are some real example solutions of system design problems, which show the huge power of the stable model semantics and ASP in system design. The first example is the generation of optimal and minimal non-blocking interconnection networks, which allow parallel connections of n source ports to n target ports. The second example builds a complete compiler chain on the base of the stable model semantics which allows to find optimal assembler code called move code for the new SCAD processor architecture, which was developed at the University of Kaiserslautern. In a last part, the lessons learned from the examples will be sketched by an idea to enhance synchronous programming languages with advanced features and statements that allow a rapid and flexible definition of search and optimization problems in synchronous languages by using their stable model interpretation with ASP.

1.2. Related Work

The formal semantics of synchronous programming languages has always been given special attention, as "[...]using a solid mathematical foundation is the ability to reason formally about the operation of the system" [BCEH03]. The Esterel synchronous language [BeGo92] introduced the *constructive causality* as a mathematical description of constructive synchronous programs in [Berr99]. That approach was adapted by Quartz [Schn09], after several studies on the semantics of synchronous programming languages and their causality [ScWe01; ScBS04b; BrSc08a].

The semantics of graphical representations of synchronous systems like Statecharts [Hare87], SyncCharts [Andr96; Andr95] or Argos [MaRe01; Mara91] was considered even earlier in many research papers [MeLu00; RoLM10; HPSS87; PnSh91; Mara92]. In particular, "What Is in a Step: New Perspectives on a Classical Question" summarizes the semantics of Statecharts and compares the corresponding Pnueli-Shalev step semantics [PnSh91] with game-theoretic semantics [AgMe11] or the synchronous semantics of Esterel, and proves that the Statecharts semantics as defined by Pnueli and Shalev 'coincides exactly with the so-called stable models introduced by Gelfond and Lifschitz' [RoLM10]. More recent work focused on the concurrent behaviour of synchronous programs and investigated denotational semantics for the synchronous programming paradigm [AMHF15].

The research in the field of logic programming semantics is quite impressive. It captures not only the general semantics definitions, like the well-founded semantics [GeRS91] or the stable model semantics [GeLi88; LiZh04b], but also

some comparisons as, e.g., [Dung92] who compared the stable model semantics with the well-founded semantics or [Lifs10], where thirteen different definitions of the stable model semantics are briefly discussed. More practical publications enhance the semantics according to their need, like, e.g., an adaption of the well-founded semantics for the case of program updates [BaAB04], or a new formal paradigm called *open-world logic programming* that enables the analysis of model parts in which some values are still unknown [JaBS13]. The stable model semantics have also been used to define reachability and deadlock properties in special Petri nets [Helj99]. Answer set programming as an implementation of the stable model semantics has been used in several applications, e.g. to plan robot coordination in [SNLG18]. A good overview on answer set applications is given in [ErGL16] and [FFST18]. Even the synthesis of embedded systems with ASP by exploiting the optimal design space is a topic of research in [NWSH17] and [NWSH18], although their work seems to be ongoing and is currently on an abstract level. But also in the domain of theoretical knowledge representation and reasoning the work goes on, as one can e.g. see in last year‘s publication about the reasoning on strong inconsistency in non-monotonic reasoning frameworks like ASP [MeMa20]. Very interesting is also the publication “Learning Dynamics with Synchronous, Asynchronous and General Semantics” [RFMR18] which tries to remove restricting semantics completely by replacing ‘will happen’ by ‘can happen’ semantics.

1.3. Outline

The thesis starts with a deeper look into the semantics of synchronous programs and logic programming in Chapter 2. Among others, the well-founded semantics and stable model semantics are presented, answer set programming as an application of the stable model semantics is introduced, and synchronous languages and especially their current semantics based on Fitting's fixpoint computation are shown.

Chapter 3 contains the main part of the thesis and shows the different approaches to extend the semantics of synchronous programming languages by the example of Quartz. In a first part, the semantics of Quartz is extended to a well-founded interpretation. In a second part, it is shown how Quartz programs can be evaluated also with the stable model semantics and how synchronous languages can be seen as a high-level access to the theory of stable models with the help of answer set programming.

The second half of the thesis focuses on two examples of the use of stable model semantics and ASP in system design in Chapter 4: First by a synthesis of optimal concentrator networks and second by a complete compiler tool chain from Quartz programs to assembler code for the SCAD architecture. Furthermore, this chapter adds an enhancement for synchronous languages by sketching how advanced ASP features could be modeled in the synchronous world.

The thesis closes in Chapter 5 with a short summary of the results.

Chapter 2

Background

Contents

2.1. Semantics of Logic Programs: From Horn Models to Stable Models 5
2.1.1. Horn Clauses 5
2.1.2. Fitting's Fixpoint Semantics 7
2.1.3. Well-founded Semantics 7
2.1.4. Beyond Well-founded Semantics: Stable Models 10
2.2. Stable Model Semantics in Practice: Answer Set Programming 12
2.3. Semantics of Synchronous Languages 14
2.3.1. Program Evaluation with Fitting's Fixpoint Semantics . 16

First of all, some background knowledge is introduced in this chapter. The first part sketch the history of logic programming semantics and introduce the concept of Answer Set Programming (ASP) which is based on the stable model semantics. The second part will then show the semantics of current synchronous languages, and how their interpretation compares with logic programming semantics.

2.1. Semantics of Logic Programs: From Horn Models to Stable Models

2.1.1. Horn Clauses

As the most basic definition, *Horn clauses* and their minimal models have to be mentioned. A *Horn clause* [Horn51] is a clause with at most one positive literal, i.e., $\{\neg x_1, \ldots, \neg x_n\}$ or $\{\neg x_1, \ldots, \neg x_n, y\}$. These clauses are named after the logician Alfred Horn and play an important role in logic programming and constructive logic. Since the literals of a clause are disjunctively connected, one may also write a Horn clause as an implication $x_1 \wedge \ldots \wedge x_n \rightarrow y$.

This implication written in the Prolog [Kowa74; CoRo96] and Quartz [Schn09] programming languages are shown in the following table.

`y :- x1,...,xn`	`module P1(event y,x1,...,xn) {`
	`  if(x1 and ... and xn) emit(y);`
	`}`

Van Emden and Kowalski showed that every set of definite clauses has a *unique minimal model* [EmKo76], and that an atomic formula is logically implied by a set of definite clauses if and only if it is true in its minimal model. The minimal model semantics of Horn clauses is the basis for the semantics of logic programs.

Hereby, the *minimality of a model* is defined in terms of the number of variables made true. If a satisfying assignment is associated with the set of variables it makes true, then minimality refers to minimal cardinality of these sets of variables. Thus, the least assignment is the one where all variables are false, and the greatest one is that one that makes all variables true. As already stated above, every set of Horn clauses has a minimal model. A minimal model of a set of Horn clauses can be constructed by the *marking algorithm* [DoGa84].

Introducing Negation and Completion of Horn Clauses

General clauses are more powerful than Horn Clauses, as they may include more complex formulas and especially negated atoms. To increase the expressiveness of Horn clauses, one would wish to introduce 'some kind' of negation in the Horn clauses. If this kind of negation was treated as the usual logical negation, one would no longer deal with a restricted set of clauses, thus with general propositional logic, and thus, the satisfiability problem would become NP-complete. To avoid these issues, some special forms of negation are preferred as the 'negation by failure' [Clar77] by Clark, which essentially states that `not(x)` is considered to be proved if and only if `x` cannot be proved. In the following, Horn clauses where negated literals are used are called *logic programs* to avoid confusion with sets of Horn clauses. Further, Clark [Clar77] introduced the 'negation by failure' in that he considered the *completion* of a logic program by computing for each variable y an equivalence $y \leftrightarrow \varphi_y$ where φ_y is the disjunction of all premises of clauses with conclusion y.

However, there are still some differences between Clark's completion and the semantics of usual logic programs (and also between the equation systems and the semantics of Quartz programs): for example, consider the following logic program and the corresponding Quartz program:

`p :- not(p)`	`module P2(event p) {`
	`  if(not(p)) emit(p);`
	`}`

For the single clause `p :- !p`, the completion `p <-> !p` is obtained, which has no two-valued models: so all formulas are implied by it. However, the logic

programming language Prolog answers 'no' to the query `p` and 'yes' to `!p`. The corresponding representation in Quartz on the right hand side is declared to be not constructive by the compiler which cannot assign a value different than $\perp$ to `p`. All in all, while Clark's completion does not give a complete characterization of the semantics of logic programs yet, the Quartz programs still behave in exactly the same way.

2.1.2. Fitting's Fixpoint Semantics

To overcome the issues of Clark's completion, another semantics has to be the corresponding semantics of logic programs. To characterize the semantics of logic programs, Fitting [Fitt85] suggested to use three-valued logic to denote whether a variable is true, false or unknown ($\perp$). To compare this iteration with *well-founded models*, two transformations of environments are formally introduced. To this end, an environment $\mathcal{I}$ is represented as a set of literals with the meaning that $x \in \mathcal{I}$ means that x is true, $\neg x \in \mathcal{I}$ means that x is false, and if neither is the case, then x is unknown.

For a given logic program P and an environment $\mathcal{I}$, two functions can be defined:

(1) $\mathcal{T}_P(\mathcal{I})$ is the set of variables y that have a rule $\ell_1, \ldots, \ell_n \to y$ in P such that all literals $\ell_1, \ldots, \ell_n$ are true in $\mathcal{I}$.

(2) $\mathcal{F}_P(\mathcal{I})$ is the set of variables y where all rules $\ell_1, \ldots, \ell_n \to y$ in P have at least one literal ℓ_i that is false in $\mathcal{I}$.

Note that (2) also applies if there is no rule with conclusion y. In [Fitt85], it has been shown that the minimal model of Horn clauses is the least fixpoint of the function $f_P(\mathcal{I}) := \mathcal{T}_P(\mathcal{I}) \cup \{\neg x \mid x \in \mathcal{F}_P(\mathcal{I})\}$. Moreover, [Fitt85] proved that a three-valued interpretation is a model of an equation system of a program P if and only if it is a fixpoint of f_P. For this reason, the meaning of a logic program P has been defined as the least fixpoint of this function f_P.

Finally, exactly the same kind of reasoning can be applied to synchronous programs: The above function $\mathcal{T}_P(\mathcal{I})$ corresponds with that part of the causality analysis that determines the 'must' actions of a program, i.e., those guarded actions (γ, α) where γ is true under $\mathcal{I}$. The function $\mathcal{F}_P(\mathcal{I})$ determines the variables that 'cannot' be assigned a value, since for all existing guarded actions (γ, α) where α could assign a value to them, the guard γ is false under $\mathcal{I}$. The causality analysis computes these two sets and then updates the environment by applying the above function $f_P(\mathcal{I})$. Thus, causality analysis of Quartz programs is exactly what is defined by Fitting as the meaning of a logic program.

2.1.3. Well-founded Semantics

Although Fitting's fixpoint semantics is already quite expressive, there are some programs for which this fixpoint cannot be computed. For this reason, some even more sophisticated semantics of logic programs were developed.

One of them is the so-called *well-founded semantics* which is described in the following. Van Gelder, Ross and Schlipf [GeRS91] introduced the *well-founded semantics of logic programs*, which is again based on a three-valued interpretation of variables as it was also the case for Fitting's fixpoint semantics. As above and in [GeRS91], these three-valued interpretations are represented by consistent sets of literals, i.e., sets that do not contain both a variable and its negation.

A canonical model in the well-founded semantics is also computed by a fixpoint computation. However, the function $g_P(\mathcal{I})$ used here instead of $f_P(\mathcal{I})$ is a stronger one, so that the well-founded semantics can be defined for programs where Fitting's fixpoint semantics cannot be defined. The part where the assignment of a variable is changed from unknown to true is thereby the same, i.e., function $\mathcal{T}_P(\mathcal{I})$ is also used, but the changes from unknown to false are done using a function $\mathcal{U}_P(\mathcal{I})$ instead of $\mathcal{F}_P(\mathcal{I})$. The definition of $\mathcal{U}_P(\mathcal{I})$ is the greatest unfounded set, which is defined as follows:

Definition 2.1 ⟨ Greatest Unfounded Set ⟩
Given a partial interpretation $\mathcal{I}$ and a logic program P, a set of variables A is called unfounded if for all variables $x \in A$ one of the following conditions holds for each rule with conclusion x:

- Some positive or negative subgoal x_i of the body is false in $\mathcal{I}$.
- Some positive subgoal of the body occurs in A.

Intuitively, an unfounded set of variables A is a set of variables that can be simultaneously made false based on a partial interpretation, i.e., changing $\mathcal{I}$ such that all variables in A will be assigned false is justified either by $\mathcal{I}$ or A.

The important observation is now that the union of unfounded sets is also an unfounded set, and therefore there is always a greatest unfounded set (which is the union of all unfounded sets). This greatest unfounded set of a program P w.r.t. a three-valued interpretation $\mathcal{I}$ is denoted as $\mathcal{U}_P(\mathcal{I})$ and can be computed as a greatest fixpoint as follows: First, all rules from P are removed where at least one subgoal is false in $\mathcal{I}$ (it is clear that these rules cannot fire since they are already disabled by the so-far determined interpretation $\mathcal{I}$). Let the remaining rules be the subset P' of P. Now, the greatest set of variables $\mathcal{U}_P(\mathcal{I})$ is sought such that x is in $\mathcal{U}_P(\mathcal{I})$ if and only if for each rule $x_1, \ldots, x_m, \neg y_1, \ldots, \neg y_n \rightarrow x$ one of the x_i is in A as well.

Thus, the procedure is started with the set of all variables $\mathcal{V}_0$ and successively removes all x from this set if there is a rule $x_1, \ldots, x_m, \neg y_1, \ldots, \neg y_n \rightarrow x$

either without positive subgoals x_i or where none of the positive subgoals is in the current set.

```
x1
x2 :- x4,not(x5)
x2 :- x5,not(x6)
x3 :- not(x2)
x4 :- x2
x5 :- x4
```

```
module P3(event x1,x2,x3,x4,x5,x6) {
    emit(x1);
    if(x4&!x5) emit(x2);
    if(x5&!x6) emit(x2);
    if(!x2) emit(x3);
    if(x2) emit(x4);
    if(x4) emit(x5);
}
```

For example, for $\mathcal{I} = \{\}$ and the above program $P3$, the greatest unfounded set $\mathcal{U}_P(\mathcal{I}) = \{\texttt{x2}, \texttt{x4}, \texttt{x5}, \texttt{x6}\}$ is computed, so that these variables could now be made false. Note that `x1` and `x3` are removed since these variables have a rule without positive subgoals.

For a fixed program P, let $\mathcal{U}_P(\mathcal{I})$ denote the greatest unfounded set and let $\mathcal{T}_P(\mathcal{I})$ denote the set of variables x whose truth values can be derived from the rules in P instantiated by the truth values of $\mathcal{I}$ (as in the fixpoint semantics). The well-founded model of a logic program is then obtained as the least fixpoint of the function $g_P(\mathcal{I}) := \mathcal{T}_P(\mathcal{I}) \cup \{\neg x \mid x \in \mathcal{U}_P(\mathcal{I})\}$, i.e., starting with $\mathcal{I} = \{\}$ and iterating with g_P yields in the limit the well-founded model of P.

For the above program, the following sets are obtained: $\mathcal{I}_0 = \{\}$, $\mathcal{I}_1 = \{\texttt{x1}, \neg\texttt{x2}, \neg\texttt{x4}, \neg\texttt{x5}, \neg\texttt{x6}\}$, and $\mathcal{I}_2 = \{\texttt{x1}, \neg\texttt{x2}, \texttt{x3}, \neg\texttt{x4}, \neg\texttt{x5}, \neg\texttt{x6}\}$. Fitting's fixpoint semantics, however, computes $\mathcal{I}_0 = \{\}$ and $\mathcal{I}_1 = \{\texttt{x1}, \neg\texttt{x6}\}$ which is also the result of the Quartz simulator for the corresponding program on the right hand side above.

The 2-valued models of the completion are $\mathcal{M}_1 = \{\texttt{x1}, \neg\texttt{x2}, \texttt{x3}, \neg\texttt{x4}, \neg\texttt{x5}, \neg\texttt{x6}\}$, $\mathcal{M}_2 = \{\texttt{x1}, \neg\texttt{x2}, \texttt{x3}, \neg\texttt{x4}, \neg\texttt{x5}, \texttt{x6}\}$, and $\mathcal{M}_3 = \{\texttt{x1}, \texttt{x2}, \neg\texttt{x3}, \texttt{x4}, \texttt{x5}, \neg\texttt{x6}\}$ so that the well-founded semantics determined the minimal model (in the sense that the fewest variables are made true)!

It is not difficult to see that the well-founded semantics always computes a minimal model since $\mathcal{U}_P(\mathcal{I})$ determines the largest set of variables that can be consistently made false (not true). Now consider the following extension of the above program with two further rules:

```
x1
x2 :- x4,not(x5)
x2 :- x5,not(x6)
x3 :- not(x2)
x4 :- x2
x5 :- x4
x7 :- not(x8)
x8 :- not(x7)
```

```
module P3'(event x1,x2,x3,x4,x5,x6,x7,x8){
  emit(x1);
  if(x4&!x5) emit(x2);
  if(x5&!x6) emit(x2);
  if(!x2) emit(x3);
  if(x2) emit(x4);
  if(x4) emit(x5);
  if(!x8) emit(x7);
  if(!x7) emit(x8);
}
```

The well-founded semantics computes $\mathcal{I}_0 = \{\}$, $\mathcal{I}_1 = \{\texttt{x1}, \neg\texttt{x2}, \neg\texttt{x4}, \neg\texttt{x5}, \neg\texttt{x6}\}$, and $\mathcal{I}_2 = \{\texttt{x1}, \neg\texttt{x2}, \texttt{x3}, \neg\texttt{x4}, \neg\texttt{x5}, \neg\texttt{x6}\}$ as before, and is therefore not able to determine values for `x7` and `x8` that depend on each other via negations. If the negations of the last two rules were omitted, the new variables `x7` and `x8` would be made false.

Since $f_P(\mathcal{I})$ is a subset of $g_P(\mathcal{I})$, it follows that the least fixpoint of f_P is a subset of the least fixpoint of g_P, thus the fixpoint semantics is weaker than the well-founded semantics. Moreover, it can be proved that the least fixpoint of g_P is also a fixpoint of f_P (although not necessarily the least one). Thus, it is a three-valued model of the equation system of P.

Again, it can be seen that the semantics of Quartz is equivalent to Fitting's fixpoint semantics, and therefore weaker than the well-founded semantics. It is therefore also possible to define a well-founded semantics of Quartz programs in the same sense as van Gelder, Ross and Schlipf [GeRS91] defined one for logic programs. Similar to the existing semantics, a canonical model that can be computed by fixpoints will be chosen this way, and programs like the one above which do not have a semantics with the current definitions, will be given a semantics with the well-founded approach.

From the computational perspective, the computation of the well-founded reaction is still polynomial in the size of the program, but requires the evaluation of an alternating fixpoint instead of a simple least one. It is therefore more expensive, i.e., it no longer runs in linear time, but it is still polynomial and therefore might scale well also for larger programs.

A generalization to non-boolean programs could be made such that as few as possible rules should be fired by the well-founded semantics.

2.1.4. Beyond Well-founded Semantics: Stable Models

The well-founded semantics is not quite the only semantics which is more powerful than Fitting's fixpoint semantics. Still, some programs can be constructed for which the well-founded semantics cannot deliver a solution as a variable assignment.

As shown by Kowalski [EmKo76], Horn clauses have a minimal model that is uniquely defined, and these models are computed by the above mentioned marking algorithm that is nothing else but a fixpoint computation as done by causality analysis. According to [Fitt85], this can be generalized to logic programs by using three-valued variable assignments so that again a minimal model semantics can be achieved. A generalization is obtained by well-founded models, and yet another approach to define the meaning of logic programs in terms of a canonical model has been taken by Gelfond and Lifschitz by the introduction of stable models [GeLi88]. In contrast to the fixpoint and the well-founded semantics, the stable model semantics does not always determine a unique model since even though also stable models are minimal models, they may not be unique since programs may have several stable models with the same number of variables made true. Especially, the introduction of negation leads to the characteristic non-determinism in the stable model semantics [SaZa90].

In contrast to Fitting's fixpoint and well-founded models, stable models are defined with two-valued logic. Thus, models are given as sets of variables with the meaning that the contained variables are exactly those that are true, see e.g., [Dung92; Fitt93].

The first and most basic definition of stable models uses the definition of the *reduct* of a logic program.

Definition 2.2 ⟨ Reduct ⟩
The reduct $\mathsf{RD}(P,\mathcal{I})$ *[GeLi88] of a logic program* P *with respect to a set* $\mathcal{I}$ *of variables is the set of Horn clauses obtained by the following two steps from* P*:*

- *Remove all rules* $\{x_1,\ldots,x_m,\neg y_1,\ldots,\neg y_n\} \to z$ *where at least one* y_i *belongs to* $\mathcal{I}$*.*
- *Replace the remaining rules* $\{x_1,\ldots,x_m,\neg y_1,\ldots,\neg y_n\} \to z$ *by the rule* $\{x_1,\ldots,x_m\} \to z$*.*

The concepts of Answer Set Programming (ASP) [EiIK09; GeLi88; CFGI13] are based on the stable model semantics. In ASP, stable models are seen as the solutions of a problem, and a logic program with several stable models will then just have several solutions, which is not considered a problem. For example, the different colorings of a graph may all be represented as stable models of a logic program describing the well-known graph coloring problem.

Furthermore, Francois Fages [Fage94] found a syntactic criterion for logic programs P so that all models of their completion are stable models of P. Fangzhen Lin and Yuting Zhao [LiZh04b] showed how to make the completion of a program P stronger by adding so-called loop formulas, so that all its non-stable models are eliminated. Those results could directly be used to compute stable models with satisfiability (SAT) solvers. The following example shows how this works in principle:

```
a :- b
b :- a
c :- not(d)
d :- not(c)
a :- not(c)
b :- d
```

```
module P4(event a,b,c,d) {
    if(b) emit(a);
    if(a) emit(b);
    if(!d) emit(c);
    if(!c) emit(d);
    if(!c) emit(a);
    if(d) emit(b);
}
```

This logic program P4 contains only one non-empty loop $L = \{a, b\}$ in the positive dependency graph of the program. The variables which support this loop L from external are $ExtSupp(L, P4) = \{\neg c, d\}$. With those sets, the disjunctive loop formula $LF_P(L)$ can be constructed by demanding that the

variables in the loop L can only hold if the external support $ExtSup$ is given: $LF_P(L) = (a \vee b) \longrightarrow (\neg c \vee d)$. Originally, Clark's completion of the program contains the following composition

```
Comp(P4) = {
    a ↔ (b ∨ ¬c),
    b ↔ (a ∨ d),
    c ↔ ¬ d,
    d ↔ ¬c
}
```

Considering only Clark's completion $Comp(P4)$, the classical model calculation would result in $M_1 = \{c\}$, $M_2 = \{a, b, d\}$ and $M_3 = \{a, b, c\}$. But when adding all loop formulas $LF(P)$ to the completion of the program, in our case $LF(P4) = \{(a \vee b) \longrightarrow (\neg c \vee d)\}$, exactly the stable models are determined. For program P4 this means that the classical models M_1 and M_2 are stable as they fulfill the loop formula, but the classical model M_3 fails and is therefore no stable model of the program $P4$. In general, finding all stable models as solutions to $Comp(P) \cup LF(P)$ can be performed by classical SAT solvers, which is the basic idea of the *assat* algorithm defined in [LiZh04b].

On top of those two basic definitions, there exist a bunch of other definitions of a stable model: Vladimir Lifschitz summarized them in the publication "Thirteen Definitions of a Stable Model" in [Lifs10]. For example, he sketches the relation between circumscription [FeLL07; FeLL11], the situation calculus [Reit01; McDe82a] or the equilibrium logic [Pear96] with the stable model semantics. But also the Pnueli-Shalev semantics to compute steps in Statecharts is seen as equivalent to the stable model semantics for logic programs [PnSh91; RoLM10].

2.2. Stable Model Semantics in Practice: Answer Set Programming

Based on the theoretical definition of stable models, a complete programming paradigm has been developed: Answer Set Programming (ASP). ASP [EiIK09; GeLi88; GeRS91; Fage94; LiZh04b] has its roots in the field of logic programming, knowledge representation and reasoning, deductive database querying, and constraint solving like SAT solving [Przy88; GeLi91; Lier17; Apt03]. It is seen as the practical implementation of the stable mode semantics and is especially capable of defining and solving all kinds of NP-hard search problems in a convenient way.

The most modern practical ASP solvers are implemented on the insights of Fangzhen Lin and Yuting Zhao [LiZh04b] and the loop formulas: A given program can be strengthened with additional formulas for variables which appear only in a dependency loop of each other, without an *external support* to give them a concrete value. [LiZh04b] showed that those enhanced programs exactly yield the stable model semantics for the corresponding original programs. This is normally implemented as a search algorithm by taking iteratively loop

formulas to the program until models can be found. Good insights into the practical implementation of ASP solvers can be found in [KLPS16b] and explain especially the difference of the two main steps *grounding* and solving and the conflict-driven approach to find stable models for a given program [GKNS07b; GeKS12].

ASP programs themselves are written in the standardized language format of the ASP-core-2 input language, defined as a first version in [CFGI13] and extended with additional constructs for version 2 in [CFGI20]. ASP programs consist of a set of rules, normally in the form `a :- b`, with a head `a`, which is deduced, and a body `b` which must hold to deduce `a`. A body can consist of multiple entries `a :-` $\texttt{b}_1$`;` $\texttt{b}_2$`;` $\ldots$`;` $\texttt{b}_n$, which all must hold, or if the body is completely omitted, then this can be written directly as *fact* with the form `a.` Further, atoms can be predicated with terms, e.g., $p(X_1, \ldots, X_m)$ such that it is possible to carry and argue about values the corresponding term X_i stands for.

For example, the following program computes all square numbers for all even X up to 10:

$$number(1..10).$$
$$square(X * X) \ :- \ number(X); X \backslash 2 == 0.$$

First of all, it captures all numbers from 1 to 10 in a predicated fact *number(i)* and then it formulates that if an X is a number and X is even (mod 2 == 0), then X*X is saved in the predicate square(i). An answer set solver computes exactly one solution for this program with the following content: number(1) number(2) number(3) number(4) number(5) number(6) number(7) number(8) number(9) number(10) square(4) square(16) square(36) square(64) square(100).

Version 2 of the ASP-core-2 input language [CFGI20] especially contains the definition of some advanced features like aggregate functions, choice rules and optimization statements.

Aggregate functions, like *sum*, *max*, *min* and *count*, allow one to reason about all terms of a given predicate. For example, the rule `up(X) :- X=#max {S : square(S)}.` returns the maximal term element in the already defined squares predicate; in our case it returns `up(100)`.

Using choice statements, predicates can be filtered and chosen according to a given criterion. These choice statements `N {p(X): q(X)} M` can be assigned how many at least (`N`) or at most (`M`) elements `p(X)` have to be chosen. For example, the choice statement `2 { upper(X): square(X),X>30 } 3` chooses `2` up to `3` out of our even square definition and assigns them to the predicate `upper`. This definition results in multiple answers, as there are multiple options to pick `2` or `3` square numbers from even numbers:

```
Answer: 1
upper(64) upper(100)
Answer: 2
upper(36) upper(64) upper(100)
Answer: 3
```

```
upper(36) upper(100)
Answer: 4
upper(36) upper(64)
```

Together with choice statements *optimization* statements [BrNT03; GeKS11] can be used in ASP in order to minimize or maximize the answer set regarding a given constraint. For example, the minimize statement `upperSum(X):-` → `X=#sum{U:upper(U)}. #minimize{X: upperSum(X)}.` asks the answer set solver to find the optimal answer out of the four answers from above, which has the least sum of square numbers in the upper predicate. It therefore picks the set as an only possible optimal answer, which contains 36 and 64 in the upper predicate as the sum is 100 and smaller than all other sums of the other answers:

```
Answer: 1
upper(36) upper(64)
Optimization: 100
OPTIMUM FOUND
```

There are other powerful extensions like disjunctive rule heads which allow one to formulate programs more conveniently such that the solver takes the task of finding solutions which fit either the one or the other predicate of the according rule head [LeRS97].

Ongoing research focuses on the power of the so-called *multishot-solving* with the help of iterative ASP programs [ObRS19; GKKS19], temporal operations and LTL formulas [ACPV11; CaDi11; CKSS18; CaSc19] or even theory propagation and solving, with the approach of *hybrid answer set programming* [KaSW17].

2.3. Semantics of Synchronous Languages

While different semantics for non- and monotonic reasoning arose, also different programming paradigms and languages came up to implement or use those semantics to their end. In order to program reactive systems and embedded systems, the paradigm of *synchronous programming languages*[BCEH03; Halb93] came up, which allows one to develop parallel embedded systems such that the hardware sensors can be monitored, read and controlled very efficiently. Classic synchronous languages divide their computation into so-called macro and micro steps where a macro step can be seen as a clock tick of a hardware circuit and every macro step is computed by a set of atomic actions called micro steps. Conventional sequential programming languages would suffer here very fast as their sequential nature hardly allows them to react, use and coordinate the parallel reaction to the sensor values and clock ticks in a way that is needed here.

As many embedded systems have very strict safety requirements, the verification of those systems plays an important role. Most synchronous languages are completely based on formal definitions which enables a formal verification of the written programs against the defined goals and requirements.

Popular representatives are the synchronous programming languages Esterel [BoSi91; Berr99], Lustre [HCRP91; CHPP87; Halb05], SIGNAL [GGBM91], and Quartz [Schn09]. Graphical representations are Statecharts [Hare87], SyncCharts [Andr96; Andr95] and Argos [MaRe01; Mara91]. The synchronous programming language Quartz [Schn09] has been designed at the University of Kaiserslautern, initially as a variant of Esterel [BoSi91]. Syntactically, macro steps in Quartz programs are therefore, as in Esterel, separated with **pause** statements. All actions in between two different **pause** statements build the micro steps of the corresponding macro step. All expressions and statements of Quartz are described by structural operational semantics (SOS) transition and reaction rules. As the current semantics are the base for the semantics extensions shown in this thesis, the semantics of Quartz and their evaluation equivalence to Fitting's fixpoint semantics of logic programs is shown in an own subsection (see Section 2.3.1). All available tools for the synchronous programming language Quartz are collected in the *Averest* [1] framework and contain, among others, compilation tools, synthesis tools, simulators, verification tools, and a collection of benchmark programs.

```
module ABRO(event bool ?a,?b,?r,!o) {
   loop
      abort {
         {wa: await(a); || wb: await(b);}
         o = true;
         wr: await(r);
      } when(r);
}
```

Figure 2.1.: *ABRO: module*

A typical Quartz program from the Averest benchmark suite called ABRO can be found in Figure 2.1. The signature of ABRO defines the input variables `a`, `b`, and `r` (indicated by the symbol `?`) and the output variable `o` (indicated by `!`). The variables `a`, `b`, and `r` are of the type boolean and define signals (keyword **event**) which means that they do not carry their values to the next macro step, but fire once.

The program body of ABRO creates a loop that resets as soon as the signal `r` appears. In the meantime, in a parallel statement the program waits for the signals `a` and `b` to hold. As soon as both signals have occurred, the output signal `o` fires and the system waits for `r` to restart the whole procedure.

The compilation of Quartz programs takes place in two separated steps: First, Quartz programs are compiled to the intermediate representation of the program which is a set of *synchronous guarded actions*. Second, those guarded actions can then be processed further up to the demand of the developer: they can be analyzed, simulated, verified, compiled to software and synthesised into hardware. Synchronous guarded actions, represented as $\gamma \Rightarrow \alpha$, serve as a uni-

[1] `http://www.averest.org`

fied intermediate representation for various synchronous languages [Schn09; BrSc11a; YBFH16; BGSS11; BGSS12]. Hereby, the guard γ describes the trigger condition that must hold such that the according atomic action α is executed. Typical atomic actions α are immediate assignments `x=`τ or delayed assignments **`next`**`(x)=`τ. Typical guards γ contain control flow labels, which describe the program state at which the action should appear. Especially every step-consuming a Quartz statement like **`pause`** or **`await`** gets assigned a unique label during compilation and special control flow actions, which describe the condition under which this label becomes active.

For example, the above mentioned ABRO program compiles to the following set of guarded actions:

```
control flow:
  true ⇒ next(running) = true
  !running ⇒ next(wa) = true
  !running ⇒ next(wb) = true
  (!r&wa&!a|r&(wr|wa|wb)) ⇒ next(wa) = true
  (!r&wb&!b|r&(wr|wa|wb)) ⇒ next(wb) = true
  (!r&(wr|a&wa&b&wb|!wa&b&wb|!wb&a&wa)) ⇒ next(wr) = true
data flow:
  (!r&(a&wa&b&wb|!wa&b&wb|!wb&a&wa)) ⇒ o = true
```

Figure 2.2.: *ABRO: synchronous guarded actions*

All not explicitly mentioned assignments, e.g., the assignments to false, are subsumed in their default reaction, often called the *reaction to absence* that takes place if no other guarded action is enabled for a variable.

All synchronous guarded actions of a program are seen to be executed and evaluated synchronously, i.e., in parallel, on the same environment. Here, the current semantics computed the least fixpoint according to a Fitting's fixpoint analysis, until all variables get their according values. For causally, correct programs in the current semantics, this fixpoint can be calculated in finitely many steps and assigns unique concrete values for all variables [Schn09].

2.3.1. Program Evaluation with Fitting's Fixpoint Semantics

This section will introduce the current semantics of synchronous languages and especially of the synchronous language Quartz in detail. Hereby it is shown that this semantics definition is equivalent to Fitting's fixpoint semantics for logic programs. Most of the results can also be found in our paper "Are Synchronous Programs Logic Programs?" [ScDa18].

The concrete semantics of Quartz and all statements is formally described by so-called structural operational semantics (SOS), which is a semantics description idea based on concepts first introduced by Plotkin [Plot81]. Furthermore, the interpretation of Quartz programs is split into two different aspects: to compute the concrete values of outputs for a given macro step, and to compute

the next control flow state. The latter is described by the so called SOS transition rules, whereas the computation of the variable environment of a single macro step is performed with the SOS reaction rules.

The SOS transition rules describe the movement of the control flow regarding the current variable assignments which were computed by the SOS reaction rules. Their idea is that a computation of the next control flow state is started with a set of statements S and a current environment $\mathcal{E}$ which provides a value $\mathcal{E}(x)$ for every variable x. As a result of the semantics description, it should compute the statements S' which have to be executed next, the actions $\mathcal{D}$ (assignments) performed by S in this computation step, and a termination flag $b \in \{\mathsf{true}, \mathsf{false}\}$ that indicates whether the statement's execution is finished or not. The general form of a SOS transition rule is given as follows:

Definition 2.3 ⟨ SOS Transition Rule ⟩

$$\langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, t \rangle$$

Of course, rules can also have preconditions φ_i: if they are fulfilled, the transition can be concluded.

Definition 2.4 ⟨ SOS Transition Rule with Assumptions ⟩

$$\frac{\varphi_1 \dots \varphi_n}{\langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, t \rangle}$$

As an example, the atomic statement **`nothing`** is taken which does not do anything.

Definition 2.5 ⟨ SOS Transition Rule for `nothing` ⟩

$$\langle \mathcal{E}, \mathtt{nothing} \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \mathtt{nothing}, \{\}, \mathsf{true} \rangle$$

In this definition one can see that no values changed and that no subsequent statement has to be executed. As a second example for a simple atomic statement, the assignment of the variable x as τ is considered next.

Definition 2.6 ⟨ SOS Transition Rule for Assignments x=τ ⟩

$$\langle \mathcal{E}, \texttt{x=}\tau \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\texttt{x=}\tau\}, \mathsf{true} \rangle$$

Here, the operational semantics collects the assignment itself in the set $\mathcal{D}$ of actions to be executed by this statement. More interesting is the definition of combined statements as for example the definition of a sequence of statements.

Definition 2.7 ⟨ SOS Transition Rules for Sequences ⟩

$$\frac{\langle \mathcal{E}, S_1 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, \mathsf{false} \rangle}{\langle \mathcal{E}, \{S_1; S_2\} \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \{S_1'; S_2\}, \mathcal{D}_1, \mathsf{false} \rangle}$$

$$\frac{\langle \mathcal{E}, S_1 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, \mathsf{true} \rangle \quad \langle \mathcal{E}, S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_2, t_2 \rangle}{\langle \mathcal{E}, \{S_1; S_2\} \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_1 \cup \mathcal{D}_2, t_2 \rangle}$$

For the sequence, two cases have to be considered. If the first set of statements S_1 is not immediate and therefore finishes the calculation of the current macro step, the sequence of $S_1; S_2$ needs to be evaluated to the remaining steps S_1' followed by S_2. Only the definitions/assignments $\mathcal{D}_1$ of the evaluation of S_1 have to be considered in this case and the sequence itself is also not immediate anymore (false). In the second case, the first set of statements S_1 can be evaluated completely within the macro step. Then, the sequence with S_2 will contain S_2' as next steps, and will contain all definitions $\mathcal{D}_1$ of S_1 and all definitions $\mathcal{D}_2$ of the immediate part of S_2. The combined sequence statement $S_1; S_2$ is then immediate iff S_2 is. All in all, the definition of the sequence of statements collects the definitions of S_1 and S_2 for all immediate statements.

In the same style, other expressions like conditional statements can be defined:

Definition 2.8 ⟨ SOS Transition Rules for Conditionals ⟩

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \ \textit{and} \ \langle \mathcal{E}, S_1 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, t_1 \rangle}{\langle \mathcal{E}, \texttt{if(}\sigma\texttt{)} \ \ S_1 \ \ \texttt{else} \ \ S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, t_1 \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \ \textit{and} \ \langle \mathcal{E}, S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_2, t_2 \rangle}{\langle \mathcal{E}, \texttt{if(}\sigma\texttt{)} \ \ S_1 \ \ \texttt{else} \ \ S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_2, t_2 \rangle}$$

For the conditional statement `if`(σ)..., the evaluation depends on the current value of the condition σ evaluated in the current environment $\mathcal{E}$. If σ evaluates to true, the statement in the if part has to be considered; if σ evaluates to false, the else part needs to be taken into account.

The above examples should be enough to show the general idea of the structural semantics definition. For a complete list of all statements of Quartz, please refer to [Schn09].

The SOS transition rules calculate the next internal control flow state, i.e., the statements which have to be executed in the next macro step. What remains is the evaluation of the variables themselves, i.e., to compute the values for all output variables given the statements, the input variables and the previous environment. That is the main task of the SOS reaction rules: they calculate the values according to the statements by the use of incomplete environments and fixpoint calculation in the sense of Fitting. To this end, the incomplete environment is filled step by step by an estimated set of assignments with the help of the so-called *must*- and *can*-sets, which are pessimistic and optimistic representations of assignments to a variable. At least all variable assignments in the must set and at most all assignments in the can set are executed. If a variable has no assignment in those sets, it is not considered in the current macro step meaning that no explicit assignment for that variable can happen when being given the current environment. To this end, so-called SOS reaction rules are introduced. Those take a statement S, an incomplete environment $\mathcal{E}$ and compute the sets $\mathcal{D}_{\mathsf{must}}$ of variables that must be given a value, $\mathcal{D}_{\mathsf{can}}$ of variables that can still be given a value, and the flags t_{must} and t_{can}, which indicate whether S must or can be executed instantaneously.

Definition 2.9 ⟨ SOS Reaction Rule ⟩

$$\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle$$

Please note that $\mathcal{D}_{\mathsf{must}} \subseteq \mathcal{D}_{\mathsf{can}}$ and $t_{\mathsf{must}} \rightarrow t_{\mathsf{can}}$ hold (see [Schn09]).

The following SOS reaction rules are defined for the same atomic actions described already by the SOS transition rules:

Definition 2.10 ⟨ SOS Reaction Rules for Some Atomic Actions ⟩

$$\langle \mathcal{E}, \texttt{nothing}\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \{\}, \{\}, \mathsf{true}, \mathsf{true}\rangle$$

$$\langle \mathcal{E}, \texttt{x=}\tau\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \{(x = \tau)\}, \{(x = \tau)\}, \mathsf{true}, \mathsf{true}\rangle$$

The `nothing` statement is instantaneously executed and no variable is added to the can- and must-sets. An assignment is also definitely executed, e.g., is in the must-set, but therefore it also can be executed. The assign statement itself is instantaneous.

Definition 2.11 ⟨ SOS Reaction Rule for Sequences ⟩

$$\frac{\langle \mathcal{E}, S_1\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{false}\rangle}{\langle \mathcal{E}, \{S_1\,;S_2\}\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{false}\rangle}$$

$$\frac{\langle \mathcal{E}, S_1\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{true}\rangle \quad \langle \mathcal{E}, S_2\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \{S_1\,;S_2\}\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, \mathsf{false}, t^2_{\mathsf{can}}\rangle}$$

$$\frac{\langle \mathcal{E}, S_1\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{true}, \mathsf{true}\rangle \quad \langle \mathcal{E}, S_2\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \{S_1\,;S_2\}\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}} \cup \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}\rangle}$$

For a *sequence* of statements $S_1;S_2$, three different rules have to be defined. If the statement S_1 is not instantaneous in the computation of the can (t^1_{can})- or must-set (t^1_{must}), the result of the sequence calculation is only the can ($\mathcal{D}^1_{\mathsf{can}}$- and must-set ($\mathcal{D}^1_{\mathsf{must}}$) of S_1 as the evaluation of the combined sequence statements must stop here in between. If the can-set calculation of S_1 is instantaneous, the can set of S_2 has to be added to the can set of S_1. Thirdly, if both the can- and must-sets of S_1 are computed instantaneously, both are unified with the can- and respectively must-set of S_2. The missing combination of the must set being computed instantaneously and the can-set not cannot happen, as it must hold that $t^1_{\mathsf{must}} \rightarrow t^1_{\mathsf{can}}$, as noted previously.

Definition 2.12 ⟨ SOS Reaction Rule for Conditionals ⟩

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \quad \langle \mathcal{E}, S_1\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{if(}\sigma\texttt{)}\ \ S_1\ \ \texttt{else}\ \ S_2\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \quad \langle \mathcal{E}, S_2\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{if(}\sigma\texttt{)}\ \ S_1\ \ \texttt{else}\ \ S_2\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \bot \quad \langle \mathcal{E}, S_1\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}}\rangle \quad \langle \mathcal{E}, S_2\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{if(}\sigma\texttt{)}\ \ S_1\ \ \texttt{else}\ \ S_2\rangle \not\rightarrow_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}} \cap \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, t^1_{\mathsf{must}} \wedge t^2_{\mathsf{must}}, t^1_{\mathsf{can}} \vee t^2_{\mathsf{can}}\rangle}$$

For the conditional statement, the evaluation of the condition σ in the current (mostly incomplete) environment determines the corresponding can - and

must-sets that are taken into account for the evaluation of the complete expression. If σ evaluates to true, the must- and can-sets of S_1 are taken; if σ evaluates to false, the sets of S_2 are taken. If the condition σ cannot be evaluated yet, e.g., evaluates to $\bot$, the can-sets of both sub-statements S_1 and S_2 are unified, as both can happen and the must-sets are intersected, as only the variables, which evaluate in both parts to a value, can be in the final must-set. In the same fashion, all other statements can be formulated in terms of SOS reaction rules (see [Schn09]).

Those SOS reaction rules can be used to calculate the least fixpoint and complete the incomplete environment step by step with new values, similar to the following code sketch.

Definition 2.13 ⟨ Computation of the System Reaction ⟩

```
function ComputeReaction(E, S, E_pre)
  do
    E_old := E;
    (D_must, D_can) := ReactSOS(E, S);
    E := UpdateEnv(D_must, D_can, E_old, E_pre)
  while E_old ≠ E
  return E
```

Hereby, the function $\mathsf{ReactSOS}(\ldots)$ is the evaluation of the SOS reaction rules as described before. The function $\mathsf{UpdateEnvs}(\ldots)$ takes the must- and can sets and proceeds as follows: Let $\mathcal{D}^{\mathtt{x}}_{\mathsf{must}}$ be all the assignments $\mathtt{x}{=}\tau$ of $\mathcal{D}_{\mathsf{must}}$ writing to $\mathtt{x}$ and $\mathcal{D}^{\mathtt{x}}_{\mathsf{can}}$ as the assignments $\mathtt{x}{=}\tau$ of $\mathcal{D}_{\mathsf{can}}$ writing to $\mathtt{x}$.

- For a variable x which has assignments in the must-set, e.g., $\mathcal{D}^{\mathtt{x}}_{\mathsf{must}} = \{\mathtt{x}{=}\tau_1, \ldots, \mathtt{x}{=}\tau_n\} \neq \{\}$, take all assignments $x = \tau_i$ and pick the supremum of the evaluation of all τ_i as new value for x, e.g., update $\mathcal{E}$ such that $\mathcal{E}(x) := \sup\{[\![x]\!]_{\mathcal{E}}, [\![\tau_1]\!]_{\mathcal{E}}, \ldots, [\![\tau_n]\!]_{\mathcal{E}}\}$
- For a variable x which has no assignments in the can-set, the so-called *reaction to absence* triggers, and sets the variable's value to its default value out of $\mathcal{E}_{\mathsf{pre}}$, e.g., false for a Boolean variable x: $\mathcal{D}^{\mathtt{x}}_{\mathsf{can}} = \{\} \Longrightarrow$ update $\mathcal{E}$ such that $\mathcal{E}(x) := \mathsf{false}$.

The definition of the function *ComputeReaction* given in Definition 2.13 shows that the environments are filled step by step until a fixpoint is reached, e.g., $\mathsf{while}\ \mathcal{E}_{\mathsf{old}} \neq \mathcal{E}$. As the function $\mathsf{UpdateEnvs}(\ldots)$ only updates values with their supremum, the update is monotonous and continuous and therefore the sketched function describes a least fixpoint computation (this directly follows from the well-known Tarski-Knasker Theorem [Tars55]).

By having the computation of the reaction for a complete macro step with the function $\mathsf{ComputeReaction}(\ldots)$ available, the SOS reaction rules and SOS

transition rules can be put together and a simplified interpreter for synchronous programs can be described as follows:

Definition 2.14 ⟨ **Quartz Interpreter** ⟩

function $InterpretQuartz(S)$
  $\mathcal{E}_{\mathsf{pre}} := \mathcal{E}_{\mathsf{def}}$; // *default values for all variables*
  do
    $\mathcal{E}_{\mathsf{in}} :=$ ReadInputs();
    $\mathcal{E}_{\mathsf{init}} :=$ UseInputs$(\mathcal{E}_{\mathsf{in}}, \mathcal{E}_{\mathsf{pre}})$;
    $\mathcal{E} :=$ ComputeReaction$(\mathcal{E}_{\mathsf{init}}, S, \mathcal{E}_{\mathsf{pre}})$;
    if $\exists x \in \mathcal{V}.\, \mathcal{E}(x) \in \{\bot, \top\}$ then fail;
    $(S', \mathcal{D}, t) :=$ TransSOS$(S, \mathcal{E})$;
    $S := S'$;
    $\mathcal{E}_{\mathsf{pre}} := \mathcal{E}$;
  while$(\neg t)$;

In every clock cycle, the inputs are read (ReadInputs()) and the current environment is filled with those input values (UseInputs(...)). Then, the new environment (concrete values for all output variables) is computed with the help of the SOS reaction rules (ComputeReaction(...)). After this, a check is performed to check whether the environment is complete. If either a value is $\bot$ or $\top$, the program was not causally correct, e.g., there were conflicting assignments to the same variable or the computation could not determine a value according to the semantics. With this complete environment, the SOS transition rules can be applied and the next macro step/remaining statements are computed (TransSOS(...)). These steps to interpret single macro steps and to get the remaining statements is repeatedly computed until no next statement remains.

All in all, this section described how the evaluation of synchronous programs is executed by current state of the art synchronous languages like Quartz. As mentioned in this section, this evaluation and especially the definition of the function UpdateEnv(...) with the contained description of the *reaction to absence* exactly describes the same semantics as Fitting's fixpoint semantics for logic programs. In the following, different approaches are shown to extend this semantics in a way such that programs can be interpreted with the well-founded semantics or the stable model semantics, which allows the interpretation of more programs as in the current state of the synchronous language definitions.

Chapter 3

Extending Semantics of Synchronous Programs

Contents

3.1. Well-founded Semantics 23
3.1.1. Extension of the SOS Rules 24
3.1.2. Direct Inclusion Into Synchronous Guarded Actions . . 41
3.1.3. Proofs 46
3.2. Stable Model Semantics 50
3.2.1. Direct Inclusion Into Quartz 50
3.2.2. Quartz as Frontend for Answer Set Programming 53

This section will introduce two extensions to the current synchronous programming semantics based on Fitting's fixpoint calculations. First of all, it is shown how the interpretation of synchronous programs can be performed with the well-founded semantics and which adaptions to the interpreter are needed. Second, another approach is shown which allows the interpretation of synchronous programs with the stable model semantics used by Answer Set Programming (ASP).

3.1. Well-founded Semantics

Well-founded semantics [GeRS91] sets the value of variables to false if they are contained in the so-called *unfounded set*. This is different to Fitting's fixpoint semantics and different to the equivalent concept of *reaction to absence* in synchronous languages which sets the value of a variable to false if its evaluation does definitely not lead to the value true.

This section examines an extension of the semantics of synchronous languages from Fitting's fixpoint semantics to well-founded semantics. The first part shows how one can extend synchronous programs to be evaluated with the

well-founded semantics by extending the SOS rules and the related simulation. The second part shows a way to reach this in practice by extending programs with new rules instead of changing the interpreter itself. This allows the interpretation with the available tools, so that all known analysis and hardware synthesis methods can still be used. The last part gives some proofs that the extension is conservative, e.g., all programs that had an interpretation with Fitting's fixpoint semantics yield the same interpretation.

3.1.1. Extension of the SOS Rules

When taking into consideration synchronous languages and their semantics in general, the so called SOS rules (see Section 2.3.1) will quickly come into play. They describe the structural semantics of synchronous languages from a semantics point of view and how the results/meanings of statements are calculated in general. Therefore, this section will focus on the changes needed to this semantics description in order to generalize the structural semantics from Fitting's fixpoint semantics up to well-founded semantics.

Example Evaluation with Current SOS Rules

First, the well-founded example P3 from Section 2.1.3 is considered with the current SOS rules for synchronous languages.

```
module P3(event x1,x2,x3,x4,x5,x6) {
    emit(x1);
    if(x4&!x5) emit(x2);
    if(x5&!x6) emit(x2);
    if(!x2) emit(x3);
    if(x2) emit(x4);
    if(x4) emit(x5);
}
```

If the interpretation procedure $InterpretQuartz(\mathsf{P3})$ is started, every value is assigned 0 for $\mathcal{E}_{\mathsf{pre}}$. The program code to be analyzed is the full program P3.

$\mathcal{E}_{\mathsf{pre}} = \{x1 = 0, x2 = 0, x3 = 0, x4 = 0, x5 = 0, x6 = 0\}$
`(S = Full program code of P3)`

The first step regarding the interpretation is to use the input variables $\mathcal{E}_{\mathsf{in}}$ (there are no input variables in our example) and add them to the initial environment for the fixpoint where every variable is assigned the value $\perp$ represented in the initial environment $\mathcal{E}_{\mathsf{init}}$.

$\mathcal{E}_{\mathsf{in}} = \varnothing$
$\mathcal{E}_{\mathsf{init}} = \{x1 = \perp, x2 = \perp, x3 = \perp, x4 = \perp, x5 = \perp, x6 = \perp\}$

Now, the reaction $\mathcal{E}$ can be computed for this setup:

$\mathcal{E} = \mathsf{ComputeReaction}($
$\{x1 = \perp, x2 = \perp, x3 = \perp, x4 = \perp, x5 = \perp, x6 = \perp\},$
$S,$
$\{x1 = 0, x2 = 0, x3 = 0, x4 = 0, x5 = 0, x6 = 0\})$

The first round to apply the SOS reaction rules takes six steps, as there are six lines of code in the original program.

```
// -----------------------------------------------
// Round 1:
// -----------------------------------------------
```

$\mathcal{E}_{\mathsf{old}} = \{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$

$(\mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}) := \mathsf{ReactSOS}($
$\quad \{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}, S)$

```
// -----------------------------------------------
// Round 1 - Step 1: (Sequence 3 + atomic assign)
// -----------------------------------------------
```

$\langle \mathcal{E}, \{\texttt{x1=1;S'}\}\rangle \looparrowright_{\mathbb{Q}} \langle \{x1 = 1\} \cup \mathcal{D}'_{\mathsf{must}}, \{x1 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}\rangle$

```
// -----------------------------------------------
// Round 1 - Step 2: (Sequence 3 + conditional 3)
// -----------------------------------------------
```

$\langle \mathcal{E}, \{\texttt{if(x4\&!x5) emit(x2); S'}\}\rangle \looparrowright_{\mathbb{Q}} \langle \mathcal{D}'_{\mathsf{must}}, \{x2 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}\rangle$

```
// -----------------------------------------------
// Round 1 - Step 3: (Sequence 3 + conditional 3)
// -----------------------------------------------
```

$\langle \mathcal{E}, \{\texttt{if(x5\&!x6) emit(x2); S'}\}\rangle \looparrowright_{\mathbb{Q}} \langle \mathcal{D}'_{\mathsf{must}}, \{x2 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}\rangle$

```
// -----------------------------------------------
// Round 1 - Step 4: (Sequence 3 + conditional 3)
// -----------------------------------------------
```

$\langle \mathcal{E}, \{\texttt{if(!x2) emit(x3); S'}\}\rangle \looparrowright_{\mathbb{Q}} \langle \mathcal{D}'_{\mathsf{must}}, \{x3 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}\rangle$

```
// -----------------------------------------------
// Round 1 - Step 5: (Sequence 3 + conditional 3)
// -----------------------------------------------
```

$\langle \mathcal{E}, \{\texttt{if(x2) emit(x4); S'}\}\rangle \looparrowright_{\mathbb{Q}} \langle \mathcal{D}'_{\mathsf{must}}, \{x4 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}\rangle$

```
// -----------------------------------------------
// Round 1 - Step 6: (Sequence 3 + conditional 3)
// -----------------------------------------------
```

$\langle \mathcal{E}, \{\texttt{if(x4) emit(x5); S'}\}\rangle \looparrowright_{\mathbb{Q}} \langle \mathcal{D}'_{\mathsf{must}}, \{x5 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}\rangle$

In the calculations above, all evaluations of σ in the `if`-statements were $\bot$ as all variables have the value $\bot$ in $\mathcal{E}_{\mathsf{old}}$. Therefore, all steps 2-6 added the corresponding assignment only to the can-set. The only variable that is in the must-set is the assignment **`emit(x1)`**.

Putting everything together, the combination results in the following must- and can set calculations for the first round.

$(\mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}) := (\{x1 = 1\}, \{x1 = 1, x2 = 1, x3 = 1, x4 = 1, x5 = 1\})$

With those must- and can-sets, the new environment for the next round can be computed as follows:

$\mathcal{E} :=$ UpdateEnv(
$\{x1 = 1\}$,
$\{x1 = 1, x2 = 1, x3 = 1, x4 = 1, x5 = 1\}$,
$\{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$,
$\{x1 = 0, x2 = 0, x3 = 0, x4 = 0, x5 = 0, x6 = 0\})$

Hereby, the function UpdateEnv will change the variables based on the can and must sets as follows:

- From $\mathcal{D}_{\mathsf{must}}$ it follows that $\mathcal{E}(x1) = 1 \longrightarrow \mathcal{E}'_1 = \{x1 = 1, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$
- From $\mathcal{D}_{\mathsf{can}}$ it follows:
  - $\mathcal{D}^{\mathsf{x2}}_{\mathsf{can}} \neq \varnothing \longrightarrow$ no changes
  - $\mathcal{D}^{\mathsf{x3}}_{\mathsf{can}} \neq \varnothing \longrightarrow$ no changes
  - $\mathcal{D}^{\mathsf{x4}}_{\mathsf{can}} \neq \varnothing \longrightarrow$ no changes
  - $\mathcal{D}^{\mathsf{x5}}_{\mathsf{can}} \neq \varnothing \longrightarrow$ no changes
  - $\mathcal{D}^{\mathsf{x6}}_{\mathsf{can}} = \varnothing \longrightarrow \mathcal{E}'_2 = \{x1 = 1, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = 0\}$

According to the algorithm ComputeReaction, the second round is started as $\mathcal{E}_{\mathsf{old}} = \{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$ is not equal to the newly calculated $\mathcal{E} = \mathcal{E}'_2 = \{x1 = 1, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = 0\}$. Nevertheless, the second iteration/round will result in the same result, i.e., $\mathcal{E} = \{x1 = 1, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = 0\} = \mathcal{E}_{\mathsf{old}}$, and the fixpoint iteration stops at this point. This is due to the fact that neither the new value of $x1 =$ true nor the new value of $x6 =$ false will make a change to the can- and must-set calculation as still no σ of the `if`-statements is computable/every σ will still evaluate to $\bot$ with the new $\mathcal{E}$. Therefore, for the example it is enough to have one round to calculate the result as demanded by Fitting's fixpoint semantics. The result is, as expected, that some variables still have assigned the value $\bot$, which means that Fitting's fixpoint semantics cannot calculate a valid result for the given program.

Changing the Interpretation

In order to determine concrete values for the undetermined variables in Fitting's fixpoint semantics, the unfounded set condition can be used. Hereby,

the idea is that if a variable is contained in the unfounded set, one can use that information to remove even more values from $\mathcal{D}_{\mathsf{can}}$, as this would fulfill the task of the unfounded set and would allow us to set more values to false than Fitting's fixpoint semantics. This can recursively lead to other values for variables, and finally to a fully evaluated environment according to the well-founded semantics.

Let's assume to have all unfounded variables according to the current environment $\mathcal{E}$ in a set $\mathcal{D}_{\mathsf{unf}}$. Then, let's update the definition of the function $\mathsf{UpdateEnvs}(\ldots)$ as follows: Let $\mathcal{D}^{\mathtt{x}}_{\mathsf{must}}$ be all the assignments $\mathtt{x}{=}\tau$ of $\mathcal{D}_{\mathsf{must}}$ writing to $\mathtt{x}$ and $\mathcal{D}^{\mathtt{x}}_{\mathsf{can}}$ as the assignments $\mathtt{x}{=}\tau$ of $\mathcal{D}_{\mathsf{can}}$ writing to $\mathtt{x}$. Additionally, define $t^{\mathtt{x}}_{\mathsf{unf}}$ as the predicate describing true if x is in the unfounded set $\mathcal{D}_{\mathsf{unf}}$ or false otherwise.

- For a variable x which has assignments in the must-set, i.e., $\mathcal{D}^{\mathtt{x}}_{\mathsf{must}} = \{\mathtt{x}{=}\tau_1, \ldots, \mathtt{x}{=}\tau_n\} \neq \{\}$, take all assignments $x = \tau_i$ and pick the supremum of the evaluation of all τ_i as new value for x, i.e., update $\mathcal{E}$ such that $\mathcal{E}(x) := \sup\{[\![x]\!]_{\mathcal{E}}, [\![\tau_1]\!]_{\mathcal{E}}, \ldots, [\![\tau_n]\!]_{\mathcal{E}}\}$.
- For a variable x which has no assignments in the can-set or which is in the unfounded set, the so-called *reaction to absence* triggers, and sets the variable's value to its default value out of $\mathcal{E}_{\mathsf{pre}}$, e.g., false for a Boolean variable x: $\mathcal{D}^{\mathtt{x}}_{\mathsf{can}} = \varnothing \vee t^{\mathtt{x}}_{\mathsf{unf}} \Longrightarrow$ update $\mathcal{E}$ such that $\mathcal{E}(x) := \mathsf{false}$.

The main idea is that the new *reaction to absence* fires not only in case the variable has no assignments in the original can-set $\mathcal{D}_{\mathsf{can}}$, but also if it is not even in the can-set reduced by the unfounded variables $\mathcal{D}_{\mathsf{can}} \setminus \mathcal{D}_{\mathsf{unf}}$. With this definition, all unfounded variables will be assigned their default value, as demanded by the well-founded semantics.

The crucial part now is the calculation of the (greatest) unfounded set. Let's recap the definition of the unfounded set.

Definition 3.1 ⟨ Greatest Unfounded Set ⟩
Given a partial interpretation $\mathcal{I}$ and a logic program P, a set of variables A is called unfounded, if for all variables $x \in A$ one of the following conditions holds for each rule with conclusion x:

- *Some positive or negative subgoal x_i of the body is false in $\mathcal{I}$.*
- *Some positive subgoal of the body occurs in A.*

In order to be able to analyze if a variable $\mathtt{x}$ is unfounded, those information are needed: what a rule, a positive subgoal and a negative subgoal of a variable $\mathtt{x}$ is regarding a partial environment $\mathcal{E}$.

In the case of the synchronous programming language Quartz, a rule is a set $\mathcal{D}_{\mathsf{C}}$ of conditions $\mathcal{C}$ that must be fulfilled by the partial environment $\mathcal{E}$

such that the variable assignment `x=`τ fires. There can be multiple rules, e.g., multiple different paths, that can trigger the variable to fire. This can be seen in the following example

```
module P(event y1,y2,x) {
    if(y1) emit(x);
    if(!y2) emit(x); }
```

which becomes the equation system `x = y1|!y2`. Here, two rules `r1=y1` and `r2=!y2` are contained. If at least one of them is fulfilled by a partial environment $\mathcal{E}$, the assignment fires and `x` gets the value true. The subgoals, the positives as well as the negatives, can be read out of the structure of the conditions $\mathcal{D}_\mathsf{C}$; in our example the rule `r1` has the positive subgoal `y1` and no negative subgoal, and the rule `r2` has no positive subgoal but the negative subgoal `y2`. For our example, the greatest unfounded set regarding e.g., the empty environment $\{y1 = \bot, y2 = \bot, x = \bot\}$ is, according to the definition of the greatest unfounded set, then the greatest fixpoint of the equation $u_x = (!y1 \vee u_{y1}) \wedge (y2) = (!y1 \vee \mathsf{true}) \wedge (y2) = y2 = \mathsf{false}$. The last step, e.g., $y2 = \mathsf{false}$ comes from the fact that the *reaction to absence* sets the value of $y2$ to false, as it is not contained in the can-set. Therefore, the unfounded set for the example becomes $\mathcal{D}_{\mathsf{unf}}^{example} = \{y1, y2\}$.

But how can the rules and the positive and negative subgoals be found with the help of the SOS rules in general? To this end, the SOS reaction rules will be extended to collect all conditions $\mathcal{C}$ for every variable assignment in a set $\mathcal{D}_\mathsf{C}$. This set can then be used to analyze the structure of the conditions and to find all rules and all positive and negative subgoals.

Definition 3.2 ⟨ SOS Reaction Rule, Well-founded ⟩

$$\langle \mathcal{E}, \mathcal{C}, S\rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}_\mathsf{must}, \mathcal{D}_\mathsf{can}, t_\mathsf{must}, t_\mathsf{can}, \mathcal{D}_\mathsf{C}\rangle$$

A SOS reaction rule for the well-founded extension knows at every step, which conditions $\mathcal{C}$ have been collected so far up to the current statement S, and can use these conditions to create a set of conditions $\mathcal{D}_\mathsf{C}$ for every subsequent assignment in S. The final result will then contain a list of conditions to be fulfilled mapped to every assignment $x = \tau$, no matter if this assignment is in the can- or must-set or in neither of both.

Definition 3.3 ⟨ SOS Reaction, Atomic Actions, Well-founded ⟩

$$\langle \mathcal{E}, \mathcal{C}, \texttt{nothing}\rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \{\}, \{\}, \mathsf{true}, \mathsf{true}, \varnothing\rangle$$

$$\langle \mathcal{E}, \mathcal{C}, \texttt{x=}\tau\rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \{(x = \tau)\}, \{(x = \tau)\}, \mathsf{true}, \mathsf{true}, \{(\mathcal{C}, x = \tau)\}\rangle$$

For an assignment it is stored that the conditions $\mathcal{C}$ were leading to this assignment. For an empty statement no mapping needs to be stored.

Definition 3.4 ⟨ SOS Reaction, Sequence, Well-founded ⟩

$$\frac{\langle \mathcal{E}, \mathcal{C}, S_1 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{false}, \mathcal{D}^1_{\mathsf{C}} \rangle}{\langle \mathcal{E}, \mathcal{C}, \{S_1 ; S_2\} \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{false}, \mathcal{D}^1_{\mathsf{C}} \rangle}$$

$$\frac{\begin{array}{c}\langle \mathcal{E}, \mathcal{C}, S_1 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{true}, \mathcal{D}^1_{\mathsf{C}} \rangle \\ \langle \mathcal{E}, \mathcal{C}, S_2 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}, \mathcal{D}^2_{\mathsf{C}} \rangle\end{array}}{\langle \mathcal{E}, \mathcal{C}, \{S_1 ; S_2\} \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, \mathsf{false}, t^2_{\mathsf{can}}, \mathcal{D}^1_{\mathsf{C}} \cup \mathcal{D}^2_{\mathsf{C}} \rangle}$$

$$\frac{\begin{array}{c}\langle \mathcal{E}, \mathcal{C}, S_1 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{true}, \mathsf{true}, \mathcal{D}^1_{\mathsf{C}} \rangle \\ \langle \mathcal{E}, \mathcal{C}, S_2 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}, \mathcal{D}^2_{\mathsf{C}} \rangle\end{array}}{\langle \mathcal{E}, \mathcal{C}, \{S_1 ; S_2\} \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}} \cup \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}, \mathcal{D}^1_{\mathsf{C}} \cup \mathcal{D}^2_{\mathsf{C}} \rangle}$$

If a sequence statement is reached, both sub-statements S_1 and S_2 are reached with the same conditions $\mathcal{C}$. When the first statement S_1 stops the execution and is not instantaneous, the assignments of S_2 will never be executed. That is why in this case only the mapping of S_1 to $\mathcal{D}^1_{\mathsf{C}}$ must be used as condition. In the case that S_1 is instantaneous in either the can execution or also in the must execution, both condition mappings $\mathcal{D}^1_{\mathsf{C}}$ and $\mathcal{D}^2_{\mathsf{C}}$ are collected and put together. With these two definitions, the sequence of `S=nothing; x=y1` would for example lead to $\mathcal{D}_{\mathsf{C}} = \varnothing \cup \{(\mathsf{true}, x = y1)\}$.

Definition 3.5 ⟨ SOS Reaction, Conditionals, Well-founded ⟩

$$\frac{\begin{array}{c}[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \\ \langle \mathcal{E}, \mathcal{C} \wedge \sigma, S_1 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}}, \mathcal{D}^1_{\mathsf{C}} \rangle \\ \langle \mathcal{E}, \mathcal{C} \wedge \neg\sigma, S_2 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}, \mathcal{D}^2_{\mathsf{C}} \rangle\end{array}}{\langle \mathcal{E}, \mathcal{C}, \texttt{if}(\sigma)\ \ S_1\ \ \texttt{else}\ \ S_2 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}}, \mathcal{D}^1_{\mathsf{C}} \cup \mathcal{D}^2_{\mathsf{C}} \rangle}$$

$$\frac{\begin{array}{c}[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \\ \langle \mathcal{E}, \mathcal{C} \wedge \sigma, S_1 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}}, \mathcal{D}^1_{\mathsf{C}} \rangle \\ \langle \mathcal{E}, \mathcal{C} \wedge \neg\sigma, S_2 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}, \mathcal{D}^2_{\mathsf{C}} \rangle\end{array}}{\langle \mathcal{E}, \mathcal{C}, \texttt{if}(\sigma)\ \ S_1\ \ \texttt{else}\ \ S_2 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}, \mathcal{D}^1_{\mathsf{C}} \cup \mathcal{D}^2_{\mathsf{C}} \rangle}$$

$$\frac{\begin{array}{c}[\![\sigma]\!]_{\mathcal{E}} = \bot \\ \langle \mathcal{E}, \mathcal{C} \wedge \sigma, S_1 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}}, \mathcal{D}^1_{\mathsf{C}} \rangle \\ \langle \mathcal{E}, \mathcal{C} \wedge \neg\sigma, S_2 \rangle \curvearrowright_{\mathbb{Q}_{WF}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}}, \mathcal{D}^2_{\mathsf{C}} \rangle\end{array}}{\begin{array}{c}\langle \mathcal{E}, \mathcal{C}, \texttt{if}(\sigma)\ \ S_1\ \ \texttt{else}\ \ S_2 \rangle \\ \curvearrowright_{\mathbb{Q}_{WF}} \\ \langle \mathcal{D}^1_{\mathsf{must}} \cap \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, t^1_{\mathsf{must}} \wedge t^2_{\mathsf{must}}, t^1_{\mathsf{can}} \vee t^2_{\mathsf{can}}, \mathcal{D}^1_{\mathsf{C}} \cup \mathcal{D}^2_{\mathsf{C}} \rangle\end{array}}$$

For conditional statements **if**$(\sigma)S_1$ **else** S_2, the condition σ is added to the conditions $\mathcal{C}$ that lead to an assignment in the substatements of S_1 or S_2.

All further statements, as the abort statements or different kind of loops, can be changed in the same matter, such that the condition of the current statements evaluation is given to the corresponding sub statements.

As a final result, if the adapted SOS reaction rules are executed for the complete program $\mathcal{P}$ respectively considering the current environment $\mathcal{E}$, a set of mappings $(\mathcal{C}, \mathtt{x}_i{=}\tau_i)$ is produced, which maps every occurring assignment to a variable to its execution conditions.

$$
\begin{aligned}
&\mathcal{D}^{\mathcal{P}}_{C,\mathcal{E}} = \{\\
&\quad (\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n, x = \tau_1),\\
&\quad \ldots,\\
&\quad (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m, y = \tau_2),\\
&\quad \ldots,\\
&\quad (\omega_1 \wedge \omega_2 \wedge \ldots \wedge \omega_p, x = \tau_3)\\
&\quad \ldots\\
&\}
\end{aligned}
$$

Especially, if a variable has assignments at different positions in the original program, all those assignments are included in the set, respectively all rules for a variable regarding the current environment $\mathcal{E}$ are available.

Let's define a new function ComputeUnfSet(...), which takes the computed mapping of conditions $\mathcal{D}_C$, the current partial environment $\mathcal{E}$ and calculates the set $\mathcal{D}_{\mathsf{unf}}$ containing all unfounded variables regarding $\mathcal{E}$.

Definition 3.6 ⟨ Computation of the Unfounded Set ⟩

```
function ComputeUnfSet(E, D_C)
  E_unf := {x := true | x ∈ A};
  D^x_C := {(C, y := τ) ∈ D_C | y == x};
  C^x := ComputeDNF(D^x_C);
  do
    E_unf,old := E_unf;
    ∀x ∈ A do
      t^x_unf = CheckUnfoundedCondition(C^x, E, E_unf);
    E_unf := {x := t^x_unf | x ∈ A};
  while E_unf,old ≠ E_unf;
  D_unf := {x ∈ A | t^x_unf == true};
  return D_unf;
```

The function starts by defining some starting conditions: according to the greatest fixpoint calculation, it is assumed that every variable is unfounded if not disproved and all conditions with a mapping to x are stored in a set $\mathcal{D}^x_{\mathsf{C}}$

for all x in the program. Furthermore, the conditions that have to be fulfilled to assign x=τ are normalized as DNF, as this allows the interpretation and separation as rules according to the definition of the unfounded set. e.g., the program

```
module P(event x1,x2,x3,x4,x5) {
    if (x1 & x2) if (x3) x4=τ;
    if (x3 & !x5) x4=τ;
}
```

results in the set $\mathcal{D}_{\mathsf{C}}^{x4} = \{((x1 \vee x2) \wedge x3, x4 = \tau), (x3 \wedge \neg x5, \texttt{x4=}\tau)\}$. This set is not directly interpretable as logical rules. But if this set is transformed to a disjunctive normal form (DNF) by concatenating all conditions $\mathcal{C}_i$ out of the mappings $(\mathcal{C}_i, \texttt{x4=}\tau)$ with a disjunction $\vee$ (every condition leads to the assignment means that only one condition needs to be fulfilled), the rules are easily cut out after the performance of the DNF transformation. In our example this means, the DNF transformation results in $\mathcal{C}^{\texttt{x4}} = (x1 \wedge x3) \vee (x2 \wedge x3) \vee (x3 \wedge \neg x5)$, which can easily be split into the three rules `r1=x1`$\wedge$`x3; r2=x2`$\wedge$`x3; r3=x3`$\wedge\neg$`x5;`. The special case of the set being empty for x, should return the trivial Boolean value $\mathcal{C}^{\texttt{x}} = \mathsf{false}$, as this means that the according variable x has no assignments in the program considering the current partial evaluation $\mathcal{E}$. Having those rules/normalized conditions $\mathcal{C}^{\texttt{x}}$ for every variable x in the program allows to calculate the greatest unfounded set $\mathcal{D}_{\mathsf{unf}}$ by a fixpoint calculation. To this end, the unfounded conditions are recursively checked for $\mathcal{C}^{\texttt{x}}$ with the (constant) environment $\mathcal{E}$ containing the current values for all variables and the (changing) environment $\mathcal{E}_{\mathsf{unf}}$, which contains the last calculated state of the fixpoint regarding the unfounded conditions of all variables.

The crucial part is now the function CheckUnfoundedCondition, which can be defined as follows:

Definition 3.7 ⟨ Check for Unfoundedness ⟩

function CheckUnfoundedCondition($\mathcal{C}^x, \mathcal{E}, \mathcal{E}_{\mathsf{unf}}$)
  $xIsUnfounded$:= true;
  $\forall conjTerm \in \mathcal{C}^x$ do
    $unfCond1 := [\![\neg conjTerm]\!]_{\mathcal{E}}$;
    $\mathsf{A}_{\mathsf{pos}}$:= GetPositiveVars($conjTerm$);
    $unfCond2 := \exists x \in \mathsf{A}_{\mathsf{pos}} \mid [\![x]\!]_{\mathcal{E}_{\mathsf{unf}}} = \mathsf{true}$;
    $xIsUnfounded := xIsUnfounded \wedge (unfCond1 \vee unfCond2)$;
  return $xIsUnfounded$;

As it can be seen, the two conditions of the unfounded set are checked for all rules, respectively all conjunction terms out of the mentioned DNF. To do so, the first condition evaluates the negation of the term/rule in the partial

environment $\mathcal{E}$, which means at least one positive or negative subgoal of that rule body evaluates to false under $\mathcal{E}$. Furthermore it is checked for all positive subgoals that at least one of it evaluates to true in $\mathcal{E}_{\mathsf{unf}}$, which means that this subgoal is recursively also a part of the unfounded set. Finally, the conjunction of all checks is returned as the conditions have to hold for all rules, respectively all conjunction terms of our formula. Above that, especially the edge cases are also considered correctly:

- for the trivial condition $\mathcal{C}^{\mathtt{x}}$=**true**, which means there is an assignment without preconditions, the function evaluates to false: there is one term `conjTerm:=`**true**. Its negation will lead to `unfCond1=`**false** and because it does not contain positive variables, also its second condition `unfCond2=`**false**.
- for the trivial condition $\mathcal{C}^{\mathtt{x}}$=**false**, which means that there was no assignment to that variable under the partial environment $\mathcal{E}$ at all, the function evaluates to true: there is only one `conjTerm:=`**false**. Its negation will lead to `unfCond1=`**true**.

All in all, the function turns the interpretation of the unfounded condition into the following scheme: If e.g., $\mathcal{C}^{x} = c_1 \vee c_2 \vee \ldots \vee c_{m_i}$ and every $c_k = x_{j_1} \wedge x_{j_2} \wedge \ldots \wedge x_{j_n} \wedge \neg x_{j_{n+1}} \wedge \neg x_{j_{n+2}} \ldots \wedge \neg x_{j_{n+z}}$, the unfounded-set condition for a rule c_k is interpreted as follows:

$t^{\mathtt{x}}_{\mathsf{unf}}$ =

```
// ------------------------------------------------
// unfounded set condition, rule k
// ------------------------------------------------
```

$\ldots \wedge ($

```
    // ------------------------------------------------
    // unfounded set condition part 1, false-evaluation
    // ------------------------------------------------
```

$(\neg x_{j_1} \vee \neg x_{j_2} \vee \ldots \vee \neg x_{j_n} \vee x_{j_{n+1}} \vee x_{j_{n+2}} \ldots \vee x_{j_{n+z}})$

$\vee$

```
    // ------------------------------------------------
    // unfounded set condition part 2, recursion
    // ------------------------------------------------
```

$(t^{\mathtt{x}_{j_1}}_{\mathsf{unf}} \vee t^{\mathtt{x}_{j_2}}_{\mathsf{unf}} \vee \ldots \vee t^{\mathtt{x}_{j_n}}_{\mathsf{unf}})$

$) \wedge \ldots$

For our small example from above with the three rules `r1=x1` $\wedge$ `x3; r2=x2` $\wedge$ `x3; r3=x3` $\wedge \neg$`x5;`, the function would lead to $t^{\mathtt{x4}}_{\mathsf{unf}} = (\neg x1 \vee \neg x3 \vee t^{\mathtt{x1}}_{\mathsf{unf}} \vee t^{\mathtt{x3}}_{\mathsf{unf}}) \wedge (\neg x2 \vee \neg x3 \vee t^{\mathtt{x2}}_{\mathsf{unf}} \vee t^{\mathtt{x3}}_{\mathsf{unf}}) \wedge (\neg x3 \vee x5 \vee \vee t^{\mathtt{x3}}_{\mathsf{unf}})$. This resulting formula would then be interpreted with the partial environments $\mathcal{E}$ for all program variables $\mathtt{x}_i$, and with the partial environment $\mathcal{E}_{unf}$ for all unfounded set conditions $t^{\mathtt{x}_i}_{\mathsf{unf}}$.

Having defined the computation of the greatest unfounded set regarding a partial environment $\mathcal{E}$ as described with the function ComputeUnfSet using the function CheckUnfoundedCondition, the complete program interpretation can be changed as follows if everything is put together:

Definition 3.8 ⟨ The Well-founded System Reaction ⟩

function ComputeReaction($\mathcal{E}, S, \mathcal{E}_{\text{pre}}$)
  $\mathcal{D}_{\text{unf}} := \varnothing$;
  do
    $\mathcal{E}_{\text{old}} := \mathcal{E}$;
    $\mathcal{D}_{\text{unf,old}} := \mathcal{D}_{\text{unf}}$;
    $(\mathcal{D}_{\text{must}}, \mathcal{D}_{\text{can}}, \mathcal{D}_C) :=$ ReactSOS$(\mathcal{E}, \text{true}, S)$;
    $\mathcal{D}_{\text{unf}} :=$ ComputeUnfSet$(\mathcal{E}, \mathcal{D}_C)$;
    $\mathcal{E} :=$ UpdateEnv$(\mathcal{D}_{\text{must}}, \mathcal{D}_{\text{can}}, \mathcal{E}_{\text{old}}, \mathcal{E}_{\text{pre}}, \mathcal{D}_{\text{unf}})$;
  while $\mathcal{E}_{\text{old}} \neq \mathcal{E} \vee \mathcal{D}_{\text{unf,old}} \neq \mathcal{D}_{\text{unf}}$
  return $\mathcal{E}$

As described, the system reaction depends on the unfounded set $\mathcal{D}_{\text{unf}}$, which is calculated by the conditions $\mathcal{D}_C$ that are collected with the SOS reaction rules. The starting condition for the SOS rule calculation is true, as all variable assignments without other conditions in the program code should be directly executed without any preconditions. The computed unfounded set $\mathcal{D}_{\text{unf}}$ is then used to update the environment as described in this chapter by especially setting all variables in the unfounded set also to its default value with an improved *reaction to absence* compared to the old fixpoint semantics. The iteration has to stop if both fixpoints $\mathcal{E}$ and $\mathcal{D}_{\text{unf}}$ are reached. Later it will be shown that these two fixpoints are alternation-free (see Section 3.1.1).

Example Evaluation with the Updated SOS Rules

Let's consider our well-founded example P3) from Section 2.1.3 again with the newly adapted SOS rules according to the well-founded definitions.

The beginning stays the same, every value is assigned false for $\mathcal{E}_{\text{pre}}$, the full program is analyzed S = P3, the initial environment is empty $\mathcal{E}_{\text{init}} = \{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$.

Now, the adapted reaction to get $\mathcal{D}_{must}$ and $\mathcal{D}_{can}$ (as before), and the new mapping of conditions $\mathcal{D}_C$ can be computed for this setup. The first round to apply the SOS reaction rules takes again six steps, as there are six lines of code in the original program.

```
// ------------------------------------------------
// Round 1:
// ------------------------------------------------
```

$\mathcal{E}_{\text{old}} = \{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$

$(\mathcal{D}_{\text{must}}, \mathcal{D}_{\text{can}}, \mathcal{D}_C) :=$ ReactSOS(
  $\{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$, true, S);

```
// ------------------------------------------------
// Round 1 - Step 1: (Sequence 3 + atomic assign)
// ------------------------------------------------
```
$\langle \mathcal{E}, 1, \{\texttt{x1=1;S'}\}\rangle$
$\quad \curvearrowright_{\mathbb{Q}_{WF}}$
$\quad\quad \langle \{x1 = 1\} \cup \mathcal{D}'_{\mathsf{must}}, \{x1 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(1, x1 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$

```
// ------------------------------------------------
// Round 1 - Step 2: (Sequence 3 + conditional 3)
// ------------------------------------------------
```
$\langle \mathcal{E}, 1, \{\texttt{if(x4\&!x5) emit(x2); S'}\}\rangle$
$\quad \curvearrowright_{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \{x2 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(x4\&!x5, x2 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$

```
// ------------------------------------------------
// Round 1 - Step 3: (Sequence 3 + conditional 3)
// ------------------------------------------------
```
$\langle \mathcal{E}, 1, \{\texttt{if(x5\&!x6) emit(x2); S'}\}\rangle$
$\quad \curvearrowright_{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \{x2 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(x5\&!x6, x2 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$

```
// ------------------------------------------------
// Round 1 - Step 4: (Sequence 3 + conditional 3)
// ------------------------------------------------
```
$\langle \mathcal{E}, 1, \{\texttt{if(!x2) emit(x3); S'}\}\rangle$
$\quad \curvearrowright_{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \{x3 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(!x2, x3 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$

```
// ------------------------------------------------
// Round 1 - Step 5: (Sequence 3 + conditional 3)
// ------------------------------------------------
```
$\langle \mathcal{E}, 1, \{\texttt{if(x2) emit(x4); S'}\}\rangle$
$\quad \curvearrowright_{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \{x4 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(x2, x4 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$

```
// ------------------------------------------------
// Round 1 - Step 6: (Sequence 3 + conditional 3)
// ------------------------------------------------
```
$\langle \mathcal{E}, 1, \{\texttt{if(x4) emit(x5); S'}\}\rangle$
$\quad \curvearrowright_{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \{x5 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(x4, x5 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$

The calculation of the must- and can-set does not change compared to the Fitting's fixpoint SOS reaction, therefore $\mathcal{D}_{\mathsf{must}} = \{x1 = 1\}$ and $\mathcal{D}_{\mathsf{can}} = \{x1 = 1, x2 = 1, x3 = 1, x4 = 1, x5 = 1\}$).

After that, the main change starts and the function ComputeUnfSet(...) executes with the partial environment $\mathcal{E}$ and with the map of conditions to assignments $\mathcal{D}_C$ from the SOS reaction rules calculation.

$$
\begin{aligned}
&\mathcal{D}_{unf} := \mathsf{ComputeUnfSet}(\\
&\quad \{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\},\\
&\quad \{\\
&\qquad (1, x1 = 1),\\
&\qquad (x4 \& !x5, x2 = 1),\\
&\qquad (x5 \& !x6, x2 = 1),\\
&\qquad (!x2, x3 = 1),\\
&\qquad (x2, x4 = 1),\\
&\qquad (x4, x5 = 1)\\
&\quad \})
\end{aligned}
$$

The first step of the function is the calculation of the DNF normalized conditions for the assignments:

$$
\begin{aligned}
\mathcal{C}^{x1} &= \mathsf{true}\\
\mathcal{C}^{x2} &= x4 \& !x5 \vee x5 \& !x6\\
\mathcal{C}^{x3} &= !x2\\
\mathcal{C}^{x4} &= x2\\
\mathcal{C}^{x5} &= x4\\
\mathcal{C}^{x6} &= \mathsf{false}
\end{aligned}
$$

Especially, the computation of the conditions for the variable $x6$ leads to false, as the set of mappings $\mathcal{D}_C^{x6} = \varnothing$, which means there is no assignment possible under the partial environment $\mathcal{E}_{old}$.

With those normalized conditions, the rules according to the unfounded set can be extracted and the unfounded condition can be checked recursively for every variable with CheckUnfoundedCondition(...) until the fixpoint is reached. This leads to the following computations with the starting environment $\mathcal{E}_{unf}(init) = \{t^{\mathsf{x1}}_{\mathsf{unf}} = 1, t^{\mathsf{x2}}_{\mathsf{unf}} = 1, t^{\mathsf{x3}}_{\mathsf{unf}} = 1, t^{\mathsf{x4}}_{\mathsf{unf}} = 1, t^{\mathsf{x5}}_{\mathsf{unf}} = 1, t^{\mathsf{x6}}_{\mathsf{unf}} = 1\}$.

```
// -----------------------------------------------
// Round 1/2 - Unfounded Set
// -----------------------------------------------
```

$$
\begin{aligned}
t^{\mathsf{x1}}_{\mathsf{unf}} &= \neg\mathsf{true} \Longrightarrow \left[\!\left[t^{\mathsf{x1}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{false}\\
t^{\mathsf{x2}}_{\mathsf{unf}} &= (\neg x4 \vee x5 \vee t^{\mathsf{x4}}_{\mathsf{unf}}) \wedge (\neg x5 \vee x6 \vee t^{\mathsf{x5}}_{\mathsf{unf}}) \Longrightarrow \left[\!\left[t^{\mathsf{x2}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{true}\\
t^{\mathsf{x3}}_{\mathsf{unf}} &= x2 \Longrightarrow \left[\!\left[t^{\mathsf{x3}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \bot\\
t^{\mathsf{x4}}_{\mathsf{unf}} &= \neg x2 \vee t^{\mathsf{x2}}_{\mathsf{unf}} \Longrightarrow \left[\!\left[t^{\mathsf{x4}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{true}\\
t^{\mathsf{x5}}_{\mathsf{unf}} &= \neg x4 \vee t^{\mathsf{x4}}_{\mathsf{unf}} \Longrightarrow \left[\!\left[t^{\mathsf{x5}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{true}\\
t^{\mathsf{x6}}_{\mathsf{unf}} &= \neg\mathsf{false} \Longrightarrow \left[\!\left[t^{\mathsf{x6}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{true}
\end{aligned}
$$

$\mathcal{E}_{unf}(round1/2) = \{t^{\mathsf{x1}}_{\mathsf{unf}} = 0, t^{\mathsf{x2}}_{\mathsf{unf}} = 1, t^{\mathsf{x3}}_{\mathsf{unf}} = \bot, t^{\mathsf{x4}}_{\mathsf{unf}} = 1, t^{\mathsf{x5}}_{\mathsf{unf}} = 1, t^{\mathsf{x6}}_{\mathsf{unf}} = 1\}$.
$\mathcal{D}_{unf}(round1/2) = \{x2, x4, x5, x6\}$.

After already one round, the fixpoint for the unfounded set calculation is reached for this example and it results in $\mathcal{D}_{unf} = \{x2, x4, x5, x6\}$. Now, this unfounded set together with the must- and can-sets can be used to update the environment $\mathcal{E}$ with the function UpdateEnv.

$\mathcal{E}$:= UpdateEnv(
$\{x1 = 1\}$,
$\{x1 = 1, x2 = 1, x3 = 1, x4 = 1, x5 = 1\}$,
$\{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$,
$\{x1 = 0, x2 = 0, x3 = 0, x4 = 0, x5 = 0, x6 = 0\}$,
$\{x2, x4, x5, x6\}$)

Hereby, the function UpdateEnv will change the variables based on the can- and must sets as before, but will also use the information from the unfounded set:

- From $\mathcal{D}_{\mathsf{must}}$ it follows, that $\mathcal{E}(x1) = 1 \longrightarrow \mathcal{E}'_1 = \{x1 = 1, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$
- From $\mathcal{D}_{\mathsf{can}}$ and $\mathcal{D}_{unf}$ it follows:
  - $\mathcal{D}^{\mathsf{x2}}_{\mathsf{can}} \neq \varnothing \vee t^{\mathsf{x2}}_{\mathsf{unf}} \longrightarrow \mathcal{E}'_2 = \{x1 = 1, x2 = 0, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$
  - $\mathcal{D}^{\mathsf{x3}}_{\mathsf{can}} \neq \varnothing \vee t^{\mathsf{x3}}_{\mathsf{unf}} \longrightarrow$ no changes
  - $\mathcal{D}^{\mathsf{x4}}_{\mathsf{can}} \neq \varnothing \vee t^{\mathsf{x4}}_{\mathsf{unf}} \longrightarrow \mathcal{E}'_3 = \{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = \bot, x6 = \bot\}$
  - $\mathcal{D}^{\mathsf{x5}}_{\mathsf{can}} \neq \varnothing \vee t^{\mathsf{x5}}_{\mathsf{unf}} \longrightarrow \mathcal{E}'_4 = \{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = 0, x6 = \bot\}$
  - $\mathcal{D}^{\mathsf{x6}}_{\mathsf{can}} = \varnothing \vee t^{\mathsf{x6}}_{\mathsf{unf}} \longrightarrow \mathcal{E}'_5 = \{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = 0, x6 = 0\}$

According to the algorithm ComputeReaction the second round is also started here as $\mathcal{E}_{\mathsf{old}} = \{x1 = \bot, x2 = \bot, x3 = \bot, x4 = \bot, x5 = \bot, x6 = \bot\}$ is not equal to the newly calculated $\mathcal{E} = \mathcal{E}'_5 = \{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = 0, x6 = 0\}$.

This time, the second iteration/round will not result in the same set as the first iteration, and differs here in comparison to the evaluation with Fitting's fixpoint semantics, and the fixpoint iteration continues. Therefore, let's take a look at the second iteration of the SOS reaction rules:

```
// ------------------------------------------------
// Round 2:
// ------------------------------------------------
```

$\mathcal{E}_{\mathsf{old}} = \{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = 0, x6 = 0\}$

$(\mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, \mathcal{D}_C)$:= ReactSOS(
$\{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = 0, x6 = 0\}$, true, S);

// --
// Round 2 - Step 1: (Sequence 3 + atomic assign)
// --
$\langle \mathcal{E}, 1, \{\texttt{x1=1;S'}\}\rangle$
$\quad \xrightarrow{\mathbb{Q}_{WF}}$
$\quad\quad \langle \{x1 = 1\} \cup \mathcal{D}'_{\mathsf{must}}, \{x1 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(1, x1 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$
// --
// Round 2 - Step 2: (Sequence 3 + conditional 2)
// --
$\langle \mathcal{E}, 1, \{\texttt{if(x4\&!x5) emit(x2); S'}\}\rangle$
$\quad \xrightarrow{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(x4\&!x5, x2 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$
// --
// Round 2 - Step 3: (Sequence 3 + conditional 2)
// --
$\langle \mathcal{E}, 1, \{\texttt{if(x5\&!x6) emit(x2); S'}\}\rangle$
$\quad \xrightarrow{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(x5\&!x6, x2 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$
// --
// Round 2 - Step 4: (Sequence 3 + conditional 1)
// --
$\langle \mathcal{E}, 1, \{\texttt{if(!x2) emit(x3); S'}\}\rangle$
$\quad \xrightarrow{\mathbb{Q}_{WF}}$
$\quad\quad \langle \{x3 = 1\} \cup \mathcal{D}'_{\mathsf{must}}, \{x3 = 1\} \cup \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(!x2, x3 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$
// --
// Round 2 - Step 5: (Sequence 3 + conditional 2)
// --
$\langle \mathcal{E}, 1, \{\texttt{if(x2) emit(x4); S'}\}\rangle$
$\quad \xrightarrow{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(x2, x4 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$
// --
// Round 2 - Step 6: (Sequence 3 + conditional 2)
// --
$\langle \mathcal{E}, 1, \{\texttt{if(x4) emit(x5); S'}\}\rangle$
$\quad \xrightarrow{\mathbb{Q}_{WF}}$
$\quad\quad \langle \mathcal{D}'_{\mathsf{must}}, \mathcal{D}'_{\mathsf{can}}, t'_{\mathsf{must}}, t'_{\mathsf{can}}, \{(x4, x5 = 1)\} \cup \mathcal{D}^2_{\mathsf{C}}\rangle$

With the new partial environment, all conditions σ of the example now evaluate to a concrete value instead of $\perp$ as it was in the first round and also in the Fitting's fixpoint evaluation. The second iteration of the SOS reaction rules result in $\mathcal{D}_{\mathsf{must}} = \{x1 = 1, x3 = 1\}$ and $\mathcal{D}_{\mathsf{can}} = \{x1 = 1, x3 = 1\}$. As one can see, all unfounded variables do not appear in the can set anymore, as their conditions to be emitted have all been turned to false. The only variable that has not been sure if it has been unfounded, `x3`, is now part of the must-set, as the evaluation of the corresponding σ=`!x2` turned positive with the new knowledge. The mappings of conditions for the assignment $\mathcal{D}_C$ are

unchanged, which comes from the fact that the evaluation of the σ is ignored by collecting the conditions.

With these results, the function ComputeUnfSet(...) is executed a second time with the partial environment $\mathcal{E}$ and with the map of conditions to assignments $\mathcal{D}_C$ from the SOS reaction rules calculation.

$$
\begin{aligned}
&\mathcal{D}_{unf} := \mathsf{ComputeUnfSet}(\\
&\quad \{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = 0, x6 = 0\},\\
&\quad \{\\
&\qquad (1, x1 = 1),\\
&\qquad (x4 \& !x5, x2 = 1),\\
&\qquad (x5 \& !x6, x2 = 1),\\
&\qquad (!x2, x3 = 1),\\
&\qquad (x2, x4 = 1),\\
&\qquad (x4, x5 = 1)\\
&\quad \})
\end{aligned}
$$

The calculation of the DNF normalized conditions does not change:

$$
\begin{aligned}
\mathcal{C}^{x1} &= \mathsf{true}\\
\mathcal{C}^{x2} &= x4 \& !x5 \vee x5 \& !x6\\
\mathcal{C}^{x3} &= !x2\\
\mathcal{C}^{x4} &= x2\\
\mathcal{C}^{x5} &= x4\\
\mathcal{C}^{x6} &= \mathsf{false}
\end{aligned}
$$

And also the rules extraction is the same, as the set $\mathcal{D}_C$ itself did not change. But the evaluation does, as the environment $\mathcal{E}$ is another one as in the first iteration:

```
// -----------------------------------------------
// Round 1/2 - Unfounded Set
// -----------------------------------------------
```

$$
\begin{aligned}
t^{\mathsf{x1}}_{\mathsf{unf}} &= \neg\mathsf{true} \Longrightarrow \left[\!\left[t^{\mathsf{x1}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{false}\\
t^{\mathsf{x2}}_{\mathsf{unf}} &= (\neg x4 \vee x5 \vee t^{\mathsf{x4}}_{\mathsf{unf}}) \wedge (\neg x5 \vee x6 \vee t^{\mathsf{x5}}_{\mathsf{unf}}) \Longrightarrow \left[\!\left[t^{\mathsf{x2}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{true}\\
t^{\mathsf{x3}}_{\mathsf{unf}} &= x2 \Longrightarrow \left[\!\left[t^{\mathsf{x3}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{false}\\
t^{\mathsf{x4}}_{\mathsf{unf}} &= \neg x2 \vee t^{\mathsf{x2}}_{\mathsf{unf}} \Longrightarrow \left[\!\left[t^{\mathsf{x4}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{true}\\
t^{\mathsf{x5}}_{\mathsf{unf}} &= \neg x4 \vee t^{\mathsf{x4}}_{\mathsf{unf}} \Longrightarrow \left[\!\left[t^{\mathsf{x5}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{true}\\
t^{\mathsf{x6}}_{\mathsf{unf}} &= \neg\mathsf{false} \Longrightarrow \left[\!\left[t^{\mathsf{x6}}_{\mathsf{unf}} \right]\!\right]^{\mathcal{E}_{unf}}_{\mathcal{E}} = \mathsf{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}_{unf}(round1/2) = \{t^{\mathsf{x1}}_{\mathsf{unf}} = 0, t^{\mathsf{x2}}_{\mathsf{unf}} = 1, t^{\mathsf{x3}}_{\mathsf{unf}} = \mathsf{false}, t^{\mathsf{x4}}_{\mathsf{unf}} = 1, t^{\mathsf{x5}}_{\mathsf{unf}} = 1, t^{\mathsf{x6}}_{\mathsf{unf}} = 1\}.\\
&\mathcal{D}_{unf}(round1/2) = \{x2, x4, x5, x6\}.
\end{aligned}
$$

The fixpoint for the unfounded set calculation is reached again after one round and results in a changed environment $\mathcal{E}_{unf}(round1/2)$ but in an unchanged

unfounded set $\mathcal{D}_{unf} = \{x2, x4, x5, x6\}$. This is the case in general, as the fixpoints are alternation-free, see Section 3.1.1.

Finally, the same unfounded set together with the must- and can-sets can be used to update the environment $\mathcal{E}$ with the function UpdateEnv.

$\mathcal{E}$:= UpdateEnv(
$\{x1 = 1, x3 = 1\}$,
$\{x1 = 1, x3 = 1\}$,
$\{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = 0, x6 = 0\}$,
$\{x1 = 0, x2 = 0, x3 = 0, x4 = 0, x5 = 0, x6 = 0\}$,
$\{x2, x4, x5, x6\}$)

The function UpdateEnv computes only small changes, because the changes are small in the can- and must-set and in the environment in comparison to the first iteration:

- From $\mathcal{D}_{\mathsf{must}}$ it follows that $\mathcal{E}(x3) = 1 \longrightarrow \mathcal{E}_1' = \{x1 = 1, x2 = 0, x3 = 1, x4 = 0, x5 = 0, x6 = 0\}$
- As the can-set $\mathcal{D}_{\mathsf{can}}$ was only reduced and the unfounded set $\mathcal{D}_{unf}$ was unchanged, both the sets have no more influence on the environment $\mathcal{E}_1'$

According to the algorithm ComputeReaction, the third round is started, as $\mathcal{E}_{\mathsf{old}} = \{x1 = 1, x2 = 0, x3 = \bot, x4 = 0, x5 = 0, x6 = 0\}$ is not equal to the newly calculated $\mathcal{E} = \mathcal{E}_1' = \{x1 = 1, x2 = 0, x3 = 1, x4 = 0, x5 = 0, x6 = 0\}$. But this iteration will end in the same environment $\mathcal{E}$ and also in the same $\mathcal{D}_{unf}$ and the fixpoint is reached. The program interpretation ends and is valid, as the resulting environment $\mathcal{E}$ is complete and does neither contain $\top$ nor $\bot$ values.

As a result one can see that the changed SOS reaction rules, the unfounded set computation and corresponding changes to update the environment allowed the interpretation of a program, which had no meaning before with Fitting's fixpoint semantics. That was reached by storing the paths/conditions to the variable assignments with the SOS reaction rules, and constructing logical rules for every variable assignment, which allowed to analyze them according to the unfounded set conditions. Finally, the variables that were found to be unfounded, could improve the *reaction to absence* and allowed to set more variables to their default values compared to Fitting's fixpoint interpretation without introducing conflicts.

Some Further Notes and Lemmas

Lemma 3.1 ($\mathcal{D}^{\mathtt{x}}_{\mathsf{must}} \neq \varnothing \longrightarrow t^{\mathtt{x}}_{\mathsf{unf}} = \mathsf{false}$)

Proof For a variable x in the must set $\mathcal{D}_{\mathsf{must}}$ it is known that all conditions $\mathcal{C}$ to assign at least one concrete expression $\mathtt{x}{=}\tau$ to this variable evaluated to true

in the current environment $\mathcal{E}$ during the application of the SOS reaction rules. This means, at least one path/one rule to that variable directly contradicts the condition 1 of the greatest unfounded set definition. Furthermore, if the whole rule evaluated to true, also all positive subgoals would evaluate to true. That finally means, these positive subgoals could not be unfounded as well. Therefore, it is clear that all variables x with an assignment in $\mathcal{D}_{\mathsf{must}}$, e.g., $\mathcal{D}^{\mathtt{x}}_{\mathsf{must}} \neq \emptyset$, x are not in the unfounded set.

Lemma 3.2 (Non-Alternation of the Fixpoints $\mathcal{E}$ and $\mathcal{D}_{unf}$) *If a variable* $\mathtt{x}$ *is set to its default value e.g.,* false *in* $\mathcal{E}$ *as* reaction to absence *because of* $t^{\mathtt{x}}_{\mathsf{unf}} = \mathsf{true}$*, that* $\mathtt{x}$ *cannot change the next iteration of the unfounded set* $\mathcal{D}_{unf}$ *computation, and the corresponding fixpoints are therefore alternation-free.*

Proof Assuming x is unfounded regarding a partial environment $\mathcal{E}$, for example **if**(y)**emit**(x); $\Longrightarrow \mathtt{t}^{\mathtt{x}}_{\mathsf{unf}}$=**true**=$\ldots \wedge (\neg \mathtt{y} \vee \mathtt{t}^{\mathtt{y}}_{\mathsf{unf}}) \wedge \ldots$. Furthermore, let's assume that for another variable x2 it could not be finally determined if it is unfounded with $\mathcal{E}$, because of a dependency to x : $t^{\mathtt{x2}}_{\mathsf{unf}} = \bot$. Now, the value of x is changed to false in the environment $\mathcal{E}_2$ because x was unfounded.

(1) Case 1: $t^{\mathtt{x2}}_{\mathsf{unf}}$ evaluates to true given $\mathcal{E}_2$ because of x

   (I) in order to evaluate to true with x = false, x must be negated in $t^{\mathtt{x2}}_{\mathsf{unf}} = \ldots \wedge \ldots \neg x \ldots \wedge \ldots$. But this means that x has to be positive in the rule $\mathcal{C}^{x2} = \ldots \wedge x \wedge \ldots$. According to the definition of the unfounded set condition 2, it must be checked, that all positive subgoals are also in the unfounded set, e.g., $t^{\mathtt{x2}}_{\mathsf{unf}} = \ldots \wedge \ldots \neg x \vee t^{\mathtt{x}}_{\mathsf{unf}} \ldots \wedge \ldots$. But as $t^{\mathtt{x}}_{\mathsf{unf}} = \mathsf{true}$, x = false cannot be the reason for $t^{\mathtt{x2}}_{\mathsf{unf}}$ to turn true. □.

(2) Case 2: $t^{\mathtt{x2}}_{\mathsf{unf}}$ evaluates to false given $\mathcal{E}_2$ because of x

   (I) $t^{\mathtt{x2}}_{\mathsf{unf}}$ = false has no influence on the unfounded set $\mathcal{D}_{unf}$ regarding $\mathcal{E}_2$, as only all positive occurrences $t^{\mathtt{x}_i}_{\mathsf{unf}}$ are included in the unfounded set.

   (II) If another $t^{\mathtt{x}_j}_{\mathsf{unf}}$ depends on $t^{\mathtt{x2}}_{\mathsf{unf}}$, and $t^{\mathtt{x2}}_{\mathsf{unf}}$ = false, also $t^{\mathtt{x}_j}_{\mathsf{unf}}$ can only become false but not true, as condition 2 of the unfounded set only demands positive checks of the set inclusion, e.g., $t^{\mathtt{x}_j}_{\mathsf{unf}} = \ldots \vee t^{\mathtt{x2}}_{\mathsf{unf}} \vee \ldots$. □.

(3) Case 3 (indirect influence): Another variable x3 evaluates to a concrete value in $\mathcal{E}_2$ because of a dependency to x and x3 has an impact to x2.

   (I) Case 3.1: x3 evaluates to true in $\mathcal{E}_2$ because of a dependency to x: e.g., **if**(!x)**emit**(x3);.

(A) $t^{\mathtt{x2}}_{\mathsf{unf}}$ evaluates to true given $\mathcal{E}_2$ because of x3.

(B) in order to evaluate to true with x3=**true**, x3 must be positive in $t^{\mathtt{x2}}_{\mathsf{unf}} = \ldots \wedge \ldots x \ldots \wedge \ldots$. But this means that x has to be negative in the rule $\mathcal{C}^{x2} = \ldots \wedge \neg x3 \wedge \ldots$. According to the definition of the unfounded set condition 2, it must be checked that all positive subgoals are also in the unfounded set, e.g., $t^{\mathtt{x2}}_{\mathsf{unf}} = \ldots \wedge \ldots \neg x \vee t^{\mathtt{x}}_{\mathsf{unf}} \ldots \wedge \ldots$. But as $t^{\mathtt{x}}_{\mathsf{unf}} = \mathsf{true}$, $x = \mathsf{false}$ cannot be the reason for $t^{\mathtt{x2}}_{\mathsf{unf}}$ to turn true. □.

3.1.2. Direct Inclusion Into Synchronous Guarded Actions

The last section showed how one can include the computation of the unfounded set condition theoretically into the formal semantics description of the SOS rules. Although this describes the changes on a theoretical level very well, in practice this would mean to change all available tools as the interpreter, the compiler, all verification and analysis tools, etc. Therefore, this section shows another approach which adds some extra statements to existing programs, such that the reduction of the can-set shown in the last section is still archived, but the interpretation can still be performed without changes to the available tools and interpreters. Especially, this allows to use the available hardware generation tools and also the interpretation with the well-founded semantics as it can be seen in the next lines. To this end, the general idea is to make the assignment explicit, e.g., add it to the must-set instead of using the implicit *reaction to absence* to set the default value. E.g., the assignment **if**(σ) x = **true**; **else nothing**; sets the value of x to false implicitly by the *reaction to absence* if σ evaluates to false; whereas **if**(σ) x = **true**; **else** x=**false**; explicitly adds x to the must set on the condition that σ evaluates to false.

According to the definition of the well-founded semantics, the part which changes the value of a variable from unknown to true is the same as in Fitting's fixpoint semantics, but the transformations from unknown to false are done by the 'unfounded set'. In order to interpret synchronous programs with the well-founded semantics, it is therefore enough to include a predicate describing the unfounded condition. To do so, for every variable x, a special variable u_x is added into a program, which should evaluate to true iff the corresponding variable x is in the unfounded set and false if it is not. This variable u_x can then be used to set the value of x explicitly to false if it evaluates to true. This is equivalent to being in the unfounded set.

```
module P3(event ...,x_i,x_{i+1},...) {
    event ...,u_{x_i},u_{x_{i+1}},...;
    // ------------------------------------------------
    // original program
    // ------------------------------------------------
    ...
    x_i = ...;
    x_{i+1} = ...;
```

```
    ...
    // ------------------------------------------------
    // reaction to absence triggered by unfounded set
    // ------------------------------------------------
    ...
    if(u_{x_i}) x_i = false;
    if(u_{x_{i+1}}) x_{i+1} = false;
    ...

}
```

The important part is now the definition of the unfounded set condition itself.

Recapping the definition of the greatest unfounded set, two conditions have to be modeled:

Definition 3.9 ⟨ Greatest Unfounded Set ⟩
Given a partial interpretation $\mathcal{I}$ and a logic program P, a set of variables A is called unfounded, if for all variables $x \in A$ one of the following conditions holds for each rule with conclusion x:

- *Some positive or negative subgoal x_i of the body is* false *in $\mathcal{I}$.*
- *Some positive subgoal of the body occurs in A.*

As mentioned in Section 2.3, synchronous programs can be transformed to an equational format, such that every variable is assigned to exactly one specific expression. Having the definition of variables in the equational format, the subgoals can be extracted directly from the different rules c_k out of the formula. If e.g., $x_i = c_1 \vee c_2 \vee \ldots \vee c_{m_i}$ and every $c_k = x_{j_1} \wedge x_{j_2} \wedge \ldots \wedge x_{j_n} \wedge \neg x_{j_{n+1}} \wedge \neg x_{j_{n+2}} \ldots \wedge \neg x_{j_{n+z}}$, the positive and negative subgoals can be extracted by taking the positive and negative occurrences of the depending variables.

All in all, the unfounded-set condition for a rule c_k can be formulated as follows:

```
u_{x_i} =
    // ------------------------------------------------
    // unfounded set condition, rule k
    // ------------------------------------------------
        ... ∧ (
            // ------------------------------------------------
            // unfounded set condition part 1, false-evaluation
            // ------------------------------------------------
                (¬x_{j_1} ∨ ¬x_{j_2} ∨ ... ∨ ¬x_{j_n} ∨ x_{j_{n+1}} ∨ x_{j_{n+2}} ... ∨ x_{j_{n+z}})
            ∨
```

```
            // ------------------------------------------------
            // unfounded set condition part 2, recursion
            // ------------------------------------------------
```
$$(u_{x_{j_1}} \vee u_{x_{j_2}} \vee \ldots \vee u_{x_{j_n}})$$
$$) \wedge \ldots$$

This is actually the same definition that had been used and defined in the last section for the unfounded set condition $t^{\mathbf{x}}_{\mathsf{unf}}$.

When taking a look into our introductory example, this can lead to the following definition, e.g., for u_2:

```
module P3(event x1,x2,x3,x4,x5,x6) {
    event u1,u2,u3,u4,u5,u6;
    // ------------------------------------------------
    // original program
    // ------------------------------------------------
    x1 = true;
    if(x4&!x5) x2 = true;
    if(x5&!x6) x2 = true;
    if(!x2) x3 = true;
    if(x2) x4 = true;
    if(x4) x5 = true;
    // ------------------------------------------------
    // reaction to absence triggered by unfounded set
    // ------------------------------------------------
    if(u1) x1 = false;
    if(u2) x2 = false;
    if(u3) x3 = false;
    if(u4) x4 = false;
    if(u5) x5 = false;
    if(u6) x6 = false;
    // ------------------------------------------------
    // computing the greatest unfounded set
    // ------------------------------------------------
    if(((x4&!x5) & !u4) | ((x5&!x6) & !u5)) u2 = false; else →
       u2 = true;
    ...
}
```

Evaluation

When taking the encoding from above the way it is, the simulation of programs which had no meaning in Fitting's fixpoint computation (some values had been $\perp$) and should have a meaning when evaluated with well-founded semantics, still evaluate to $\perp$ for most of those values. This comes from the fact that some of the rules c_k evaluated to $\perp$ for those variables, what means that some depending variables x_{j_l} could not have been evaluated before. But as the

same values are used by the condition of the unfounded set, the unfounded set condition for those variables also evaluates to $\perp$ and thus did not shift the interpretation to well-founded semantics yet. This can be seen clearly if one takes a look at our example:

The evaluation starts with all variables $\perp$: $\mathcal{E}_{\mathsf{init}} = \{x1 = \perp, x2 = \perp, x3 = \perp, x4 = \perp, x5 = \perp, x6 = \perp, u1 = \perp, u2 = \perp, u3 = \perp, u4 = \perp, u5 = \perp, u6 = \perp\}$.

Due to the current interpretation with the must and can sets, the variable x1 gets the value $x1 = \mathsf{true}$ and the variable x6 the value $x6 = \mathsf{false}$ because of to the reaction to absence. Further, the corresponding unfounded conditions are the trivial cases and therefore directly set as $u1 = \mathsf{false}$ and $u6 = \mathsf{true}$. But all other unfounded conditions still evaluate to $u_i = \perp$ as all conditions of their if-statement can still not be evaluated with these new variable values and the starting values $\perp$.

When considering the well-founded semantics definition in detail, it is important to see that the evaluation of the unfounded set must be executed by a greatest fixpoint analysis. Especially, all variables for which the unfounded set condition cannot be evaluated at the initial time have to be considered as being in the set, e.g., evaluating the condition to true.

Of course this could be reached by tagging the newly introduced expressions and conditions to be evaluated with a greatest fixpoint and changing the simulator and postponed steps of the compilation (hardware synthesis etc). But if one directly defines these fixpoints by unrolling the expressions, it is possible to use the existing eco-system without changes like synthesizing hardware or simulating the program.

Hence, the initial unfounded set condition of the unrolling $u_{x_i,0}$ of every variable x_i is set to true (unless it has assigned a constant value true in the original program, as then the unfounded set condition itself is false). Then, all intermediate calculation steps of the fixpoint unrolling $u_{x_i,l}$ refer to all values of the previous iteration $u_{x_j,l-1}$ and have the previous value $u_{x_i,l-1}$ as a fallback. As the maximal depth of the fixpoint, e.g., in how many steps the fixpoint can change one variable's value from true to false, is at maximum the amount of variables m in the program, the fixpoint can be unrolled by unrolling every single expression u_{x_i} up to $u_{x_i,m}$.

```
u_{x_i} = u_{x_i,m};
// ------------------------------------------------
// unfolding step m for x_i
// ------------------------------------------------
u_{x_i,m} =
    // ------------------------------------------------
    // unfounded set condition, rule k
    // ------------------------------------------------
        ( ... ∧ (
            // ------------------------------------------------
            // unfounded set condition part 1, false-evaluation
            // ------------------------------------------------
```

$$(\neg x_{j_1} \vee \neg x_{j_2} \vee \ldots \vee \neg x_{j_n} \vee x_{j_{n+1}} \vee x_{j_{n+2}} \ldots \vee x_{j_{n+z}})$$

$\vee$

```
// ------------------------------------------------
// unfounded set condition part 2, recursion.
// refers to previous unrolling m-1
// ------------------------------------------------
```

$$(u_{x_{j_1},m-1} \vee u_{x_{j_2},m-1} \vee \ldots \vee u_{x_{j_n},m-1})$$

$) \wedge \ldots)$

```
// ------------------------------------------------
// unfounded set, greatest fixpoint recursion itself
// ------------------------------------------------
```

$\vee \; u_{x_i,m-1}$

$\ldots$

```
// ------------------------------------------------
// unfolding step 1 for x_i
// ------------------------------------------------
```

$u_{x_i,0}$ `= true;`

For the example, this means:

```
// ------------------------------------------------
// computing the greatest unfounded set
// ------------------------------------------------
...
u2 = u2_6;
...
// ------------------------------------------------
// unfolding u2
// ------------------------------------------------
if(((x4&!x5) & !u4_5) | ((x5&!x6) & !u5_5)) u2_6 = false; →
   else u2_6 = u2_5;
if(((x4&!x5) & !u4_4) | ((x5&!x6) & !u5_4)) u2_5 = false; →
   else u2_5 = u2_4;
if(((x4&!x5) & !u4_3) | ((x5&!x6) & !u5_3)) u2_4 = false; →
   else u2_4 = u2_3;
if(((x4&!x5) & !u4_2) | ((x5&!x6) & !u5_2)) u2_3 = false; →
   else u2_3 = u2_2;
if(((x4&!x5) & !u4_1) | ((x5&!x6) & !u5_1)) u2_2 = false; →
   else u2_2 = u2_1;
if(((x4&!x5) & !u4_0) | ((x5&!x6) & !u5_0)) u2_1 = false; →
   else u2_1 = u2_0;
u2_0 = true;
```

With this trick, it is possible to transform every parallel program's interpretation to well-founded semantics without losing the capability to synthesize the program to hardware or to analyze it further with the existing tools. In

the following, the proofs will be given that the semantics of programs is only extended and especially does not change for programs for which a solution in the sense of Fitting has already existed.

The shown strategy introduces n^2 new formulas to a program, which is of course not optimal particularly for larger programs. Although the computation of the fixpoint is just linear in n because the current level computation of ux_l only influences the next level $l+1$, this representation can further be optimized in reality. One such optimization could be the calculation of the dependency depth of a variable in the dependency graph, as only subsequent data dependent variables x_j can influence the value of a variable x and therefore also the corresponding unfounded set conditions $ux_{j,l}$ the value of ux_{l+1}. But such optimizations do not change the general idea, and still allow the computation of a well-founded system's reaction with the help of the currently available rules and tools.

3.1.3. Proofs

This section will prove the conservative characteristics of the shown extension to well-founded semantics. The proofs will be performed by inductive proofs over the data dependency order. Therefore, the notation of a rule c_k in the case expression $\mathsf{true} \longrightarrow x = c_1 \vee c_2 \vee \ldots \vee c_m$ is enriched with a data dependency index j_y for every x in the rule as follows: $c_k = x_{j_1} \wedge x_{j_2} \wedge \ldots \wedge x_{j_n} \wedge \neg x_{j_{n+1}} \wedge \neg x_{j_{n+2}} \ldots \wedge \neg x_{j_{n+z}}$ which can be combined to $c_k = \bigwedge_{r_1=1..n} x_{j_{r_1}} \wedge \bigwedge_{r_2=n+1..n+z} \neg x_{j_{r_2}}$. Hereby, every x_{j_y} is computed in the data dependency order before x.

Furthermore, let's introduce two notations to clearly separate the interpretation with Fitting from the interpretation with well-founded semantics:

$I_F(x_i)$ is the interpretation of the variable x_i with Fitting's fixpoint semantic

$I_W(x_i)$ is the interpretation of the variable x_i with the extended semantics of well-founded models presented in the last sections

Basically there are to prove two different aspects:

- A variable made false because of the well-founded set/unfounded set condition was already evaluating to false or $\perp$ under the fixpoints semantics → the extension is conservative
- everything having evaluated to a concrete value under Fitting's fixpoint semantics has the same value under the newly introduced extension evaluating the well-founded semantics → conflict-free extension

Direct Influence of the Unfounded Set

Lemma 3.3 $u_{x_i} \longrightarrow I_F(x_i) = \mathsf{false}|\perp$

Proof The above lemma is proved with an inductive proof over the data dependency order. Since our program is derived from Quartz, it is known that these data dependencies exist.

(1) Induction Base: All constant variables are the base of the data dependencies. It is to prove that $u_{x_0} \longrightarrow I_F(x_0) = \mathsf{false}|\bot$. Let's assume $u_{x_0} = \mathsf{true}$. It directly follows from the definition of the extension that for constants only $x_0 = \mathsf{false}$ can lead to $u_{x_0} = \mathsf{true}$. The constant $x_0 = \mathsf{true}$ would lead to $u_{x_0} = \mathsf{false}$ □

(2) Induction Step: Assume the Lemma holds for all $j_y \leq i$: $u_{x_{j_y}} \longrightarrow I_F(x_{j_y}) = \mathsf{false}|\bot$ It is to prove that the Lemma also holds for $i+1$: $u_{x_{i+1}} \longrightarrow I_F(x_{i+1}) = \mathsf{false}|\bot$. Let's assume $u_{x_{i+1}} = \mathsf{true}$. As $u_{x_{i+1}} = u_{c_1} \wedge u_{c_2} \wedge \ldots \wedge u_{c_m}$, it is known that all u_{c_k} must be true. Let's take an arbitrary row k: $u_{c_k} = u^{c_k}_{x_{i+1},p_1} \vee u^{c_k}_{x_{i+1},p_2}$ can only be true if one of the two introduced well-founded conditions p_1 or p_2 is true.

(I) Case 1: $u^{c_k}_{x_{i+1},p_1} = \mathsf{true}$.

$$u^{c_k}_{x_{i+1},p_1} = \bigvee_{r_1=1..n_k} \neg x_{j_{r_1}} \vee \bigvee_{r_2=n_k+1..n_k+z_k} x_{j_{r_2}}$$

Therefore, either one $\neg x_{j_{r_1}}$ must be true or one $x_{j_{r_2}}$ for arbitrary $r*$.

(A) Case 1.1 $\neg x_{j_{r_1}} = \mathsf{true}$

This means, $x_{j_{r_1}}$ must be false. But then the corresponding case condition $c^{i+1}_k = \bigwedge_{r_1=1..n_k} x_{j_{r_1}} \wedge \ldots$ would never be able to evaluate to true. Therefore, if $\neg x_{j_{r_1}} = \mathsf{true}$ and an arbitrary k is chosen, $I_F(x_{i+1})$ must evaluate to $\mathsf{false}|\bot$ □

(B) Case 1.2 $x_{j_{r_2}} = \mathsf{true}$

This means, $\neg x_{j_{r_2}}$ must be false. But then the corresponding case condition $c^{i+1}_k = \ldots \wedge \bigwedge_{r_2=n_k+1..n_k+z_k} \neg x_{j_{r_2}}$ would never be able to evaluate to true. Therefore, if $x_{j_{r_2}} = \mathsf{true}$ and it was chosen an arbitrary k, $I_F(x_{i+1})$ must evaluate to $\mathsf{false}|\bot$ □

(II) Case 2: $u^{c_k}_{x_{i+1},p_2} = \mathsf{true}$

As $u^{c_k}_{x_{i+1},p_2} = \bigvee_{r_1=1..n_k} u_{x_{j_{r_1}}}$ at least one $u_{x_{j_{r_1}}}$ must be true. As for all $j_y \leq i$ it is given that $u_{x_{j_y}} \longrightarrow I_F(x_{j_y}) = \mathsf{false}|\bot$ from the induction assumption, it can follow that $I_F(x_{j_{r_1}}) = I_F(x_{j_y}) = \mathsf{false}|\bot$. But no matter if this positive subgoal evaluates to false or to $\bot$, the corresponding case condition c_k must evaluate to $\mathsf{false} \vee \bot$ too. As it were chosen arbitrary k and r, it directly follows that $I_F(x_{i+1})$ must evaluate to $\mathsf{false}|\bot$ then, too □

Indirect/Subsequent Influence of the Unfounded Set

It is known that $u_{x_i} = \mathsf{true}$ will lead to $I_F(x_i) = \mathsf{false} \vee \bot$. But can the variable x_i, which was made false from u_{x_i}, conflict Fitting's semantic? What exactly does conflict mean:

case 1 Fitting would normally say $I_F(x_i) = \mathsf{true}$, but one $I_W(x_{i-d}) = \mathsf{false}$ enforces x_i to be false.

case 2 Fitting would normally say $I_F(x_i) = \mathsf{false}$, but one $I_W(x_{i-d}) = \mathsf{false}$ enforces x_i to be true.

There is a third case in which Fitting would normally say $I_F(x_i) = \bot$, but $I_W(x_{i-d}) = \mathsf{false}$ enforces x_i to be true or false. This case 3 is no conflict; it is the wanted intention of the defined extension to well-founded semantics.

Well-founded Semantics Cannot Conflict true To be disproved: Fitting would normally say $I_F(x_i) = \mathsf{true}$ but one $I_W(x_{i-d}) = \mathsf{false}$ enforces x_i to be false.

As $I_F(x_i) = \mathsf{true} = c_1 \vee c_2 \vee \ldots \vee c_m$, it can be followed that at least one condition c_k fired and evaluated to true.

In order to change the value of x_i to false, one x_{i-d}, which was made false because of the unfounded set conditions, must change the value of the corresponding condition c_k to false.

In order to change $c_k = \bigwedge_{r_1=1..n_k} x_{j_{r_1}} \wedge \bigwedge_{r_2=n_k+1..n_k+z_k} \neg x_{j_{r_2}}$ from true to false with $I_W(x_{i-d}) = \mathsf{false}$ means that x_{i-d} is one of the positive subgoals $x_{j_{r_1}}$.

This means also that $x_{j_{r_1}}$ evaluated to true under Fitting's semantic: $I_F(x_{j_{r_1}}) = I_F(x_{i-d}) = \mathsf{true}$.

As $x_{i-d} = x_{j_{r_1}} = c_1^{r_1} \vee c_2^{r_1} \vee \ldots \vee c_{m_2}^{r_1} = \mathsf{true}$ under Fitting's semantics, at least one condition $c_{k_2}^{r_1} = \mathsf{true}$ with $c_{k_2}^{r_1} = \mathsf{true} = \bigwedge_{r_3=1..n_{k_2}} x_{j_{r_3}} \wedge \bigwedge_{r_4=n_{k_2}+1..n_{k_2}+z_{k_2}} \neg x_{j_{r_4}}$. Which means (1) all positive subgoals $x_{j_{r_3}}$ are true and (2) all negative subgoals $x_{j_{r_4}}$ are false.

At the same time, $u_{x_{i-d}} = u_{x_{j_{r_1}}} = u_{x_{c_1}}^{r_1} \wedge u_{x_{c_2}}^{r_1} \wedge \ldots \wedge u_{x_{c_{m_2}}}^{r_1} = \mathsf{true}$ means that the unfounded set condition is true for every case condition $u_{x_{c_{k_3}}}^{r_1}$, especially also for the case condition with the same index $k_3 = k_2$ that had been chosen for Fitting's semantic: $u_{x_{c_{k_2}}}^{r_1} = \mathsf{true}$. Because of $u_{x_{c_{k_2}}}^{r_1} = u_{x_{c_{k_2}},p_1}^{r_1} \vee u_{x_{c_{k_2}},p_2}^{r_1} = \mathsf{true}$, two different cases have to be considered:

(1) Case 1: $u_{x_{c_{k_2}},p_1}^{r_1} = \mathsf{true}$

The unfounded set definition p_1 demands at least one of the positive or negative subgoals to evaluate to false:

$$u_{x_{c_{k_2}},p_1}^{r_1} = \bigvee_{r_3=1..n_{k_2}} \neg x_{j_{r_3}} \vee \bigvee_{r_4=n_{k_2}+1..n_{k_2}+z_{k_2}} x_{j_{r_4}} = \mathsf{false}.$$

But this means at least one $x_{j_{r_3}}$ must be false or one $x_{j_{r_4}}$ must be true, which is both a contradiction to the above shown assumptions

(1) and respectively (2) that the variable x_{i-d} = true under Fitting's semantics. □

(2) Case 2: $u^{r_1}_{x_{c_{k_2}},p_2}$ = true

The unfounded set definition p_2 demands that at least one positive subgoal of the considered case condition $c^{r_1}_{k_2}$: $u^{r_1}_{x_{c_{k_2}},p_2} = \bigvee_{r_3=1..n_{k_2}} u_{x_{j_{r_3}}}$ = true. But if this $u_{x_{j_{r_3}}}$ = true, the corresponding variable $x_{j_{r_3}}$ must evaluates to false $\vee \perp$ under Fitting's semantics according to the proof shown in Section 3.1.3. This is a contradiction to the above shown assumption (1), that the variable $x_{j_{r_3}}$ = true given x_{i-d} = true under Fitting's semantics. □

Well-founded Semantics Cannot Conflict false To be disproved: Fitting would normally say $I_F(x_i)$ = false but $I_W(x_{i-d})$ = false enforces x_i to be true.

$I_F(x_i)$ = false = $c_1 \vee c_2 \vee \ldots \vee c_m$ means that all conditions c_k are false.

In order to change the value of x_i to true, one x_{i-d}, which was made false because of the unfounded set condition and is contained in at least one condition c_k, must change the value of the condition to true.

In order to change $c_k = \bigwedge_{r_1=1..n_k} x_{j_{r_1}} \wedge \bigwedge_{r_2=n_k+1..n_k+z_k} \neg x_{j_{r_2}}$ from false to true with $I_W(x_{i-d})$ = false means the x_{i-d} is one of the negative subgoals $\neg x_{j_{r_2}}$.

This also means that $x_{j_{r_2}}$ evaluated to true under Fitting's semantic: $I_F(x_{j_{r_2}}) = I_F(x_{i-d})$ = true. The only way how $I_W(x_{i-d})$ = false could happen under these circumstances is if x_{i-d} was in the unfounded set: $u_{x_{i-d}}$ = true.

As $x_{i-d} = x_{j_{r_2}} = c^{r_2}_1 \vee c^{r_2}_2 \vee \ldots \vee c_{m_2}$ = true under Fitting's semantics, at least one condition $c^{r_2}_{k_2}$ = true with $c^{r_2}_{k_2}$ = true = $\bigwedge_{r_3=1..n_{k_2}} x_{j_{r_3}} \wedge \bigwedge_{r_4=n_{k_2}+1..n_{k_2}+z_{k_2}} \neg x_{j_{r_4}}$. This means (1) all positive subgoals $x_{j_{r_3}}$ must evaluate to true and (2) all negative subgoals $x_{j_{r_4}}$ to false.

At the same time, $u_{x_{r_2}} = u^{r_2}_{x_{c_1}} \wedge u^{r_2}_{x_{c_2}} \wedge \ldots \wedge u^{r_2}_{x_{c_{m_2}}}$ = true means that the unfounded set condition is true for every case condition $u^{r_2}_{x_{c_{k_3}}}$, especially also for the case condition with the same index $k_3 = k_2$ that had been chosen for Fitting's semantic: $u^{r_2}_{x_{c_{k_2}}}$ = true. Because of $u^{r_2}_{x_{c_{k_2}}} = u^{r_2}_{x_{c_{k_2}},p_1} \vee u^{r_2}_{x_{c_{k_2}},p_2}$ = true, two different cases have to be considered:

(1) Case 1: $u^{r_2}_{x_{c_{k_2}},p_1}$ = true

The unfounded set definition p_1 demands at least one of the positive or negative subgoals to evaluate to false:

$u^{r_2}_{x_{c_{k_2}},p_1} = \bigvee_{r_3=1..n_{k_2}} \neg x_{j_{r_3}} \vee \bigvee_{r_4=n_{k_2}+1..n_{k_2}+z_{k_2}} x_{j_{r_4}}$ = false.

But this means at least one $x_{j_{r_3}}$ must be false or one $x_{j_{r_4}}$ must be true, which is both a contradiction to the above shown assumptions (1) and respectively (2), that the variable x_{i-d} = true under Fitting's semantics. □

(2) Case 2: $u^{r_2}_{x_{c_{k_2}},p_2} = \mathsf{true}$

The unfounded set definition p_2 demands that at least one positive subgoal of the considered case condition $c^{r_2}_{k_2}$: $u^{r_2}_{x_{c_{k_2}},p_2} = \bigvee_{r_3=1..n_{k_2}} u_{x_{j_{r_3}}} = \mathsf{true}$. But if this $u_{x_{j_{r_3}}} = \mathsf{true}$, the corresponding variable $x_{j_{r_3}}$ must be false according to the definition of the unfounded set extension. This is a contradiction to the above shown assumption (1), that the variable $x_{i-d} = \mathsf{true}$ under Fitting's semantics, which is even more underlined by the proof shown in Section 3.1.3 that $u_{x_i} \longrightarrow I_F(x_i) = \mathsf{false}|\bot$. □

3.2. Stable Model Semantics

3.2.1. Direct Inclusion Into Quartz

Based on the concepts of the unfounded set, the stable model semantics can be defined. The idea behind stable models is that every variable needs to have a reason to become true. Thus, there must be a rule for this variable. In contrast to well-founded semantics, stable model semantics can have multiple solutions. Furthermore, it is known that all *total* well-founded models, e.g., in which no variable has the value $\bot$, are the stable models of a program. Thus, the solution of the well-founded semantics is a subset of all stable models. Therefore, it would be enough to select appropriate values for all variables that have the value $\bot$ in the well-founded semantics.

Considering those facts, it is not surprising that the original definition of a stable model with the *reduct* of a program uses the unfounded condition to describe the properties of the stable models (see Section 2.1.4).

Nevertheless, a lot of knowledge on the stable model semantics exists; the most advanced is the connection and presentability with the help of loop formulas. Hereby, the information of the program variable loops is collected, and even more important are the rules which are excluded and therefore supporting the loop variables. This leads to the possibility of reducing the check to stable models to a default SAT problem, by putting together Clark's completion of the program with the associated supporting loop formulas. This idea has been presented in Section 2.1.4 in the example program P4. In order to evaluate Quartz programs with the stable model semantics, it seems close to just add the corresponding loop formulas to the program as it was the case for the well-founded semantics and its unfounded set condition in Section 3.1.2.

```
module P4'(event a,b,c,d) {
    event ua,ub,uc,ud,ua_0,ua_1,ua_2,ua_3, →
        ua_4,ub_0,ub_1,ub_2,ub_3,ub_4, →
        uc_0,uc_1,uc_2,uc_3,uc_4, ud_0,ud_1,ud_2,ud_3,ud_4;

    // ------------------------------------------------
    // original program
    // ------------------------------------------------
    if(a) b=true;
```

```
if(b) a=true;
if (!c) a=true;
if(!d) c=true;
if(!c) d=true;
if(d) b=true;

// ------------------------------------------------
// loop formulas
// ------------------------------------------------
if(!d & c) a=false;
if(!d & c) b=false;

// ------------------------------------------------
// reaction to absence triggered by unfounded set
// ------------------------------------------------
if(ua) a = false;
if(ub) b = false;
if(uc) c = false;
if(ud) d = false;

// ------------------------------------------------
// computing the greatest unfounded set
// ------------------------------------------------
ua = ua_4;
ub = ub_4;
uc = uc_4;
ud = ud_4;

// ------------------------------------------------
// unfolding ua
// ------------------------------------------------
if((b & !ub_3) | ((!d & c) & !uc_3) | !c) ua_4 = false; →
   else ua_4 = ua_3;
if((b & !ub_2) | ((!d & c) & !uc_2) | !c) ua_3 = false; →
   else ua_3 = ua_2;
if((b & !ub_1) | ((!d & c) & !uc_1) | !c) ua_2 = false; →
   else ua_2 = ua_1;
if((b & !ub_0) | ((!d & c) & !uc_0) | !c) ua_1 = false; →
   else ua_1 = ua_0;
ua_0 = true;
// ------------------------------------------------
// unfolding ub
// ------------------------------------------------
if((a & !ua_3) | ((!d & c) & !uc_3) | (d & !ud_3)) ub_4 = →
   false; else ub_4 = ub_3;
```

```
    if((a & !ua_2) | ((!d & c) & !uc_2) | (d & !ud_2)) ub_3 = →
       false; else ub_3 = ub_2;
    if((a & !ua_1) | ((!d & c) & !uc_1) | (d & !ud_1)) ub_2 = →
       false; else ub_2 = ub_1;
    if((a & !ua_0) | ((!d & c) & !uc_0) | (d & !ud_0)) ub_1 = →
       false; else ub_1 = ub_0;
    ub_0 = true;
    // ------------------------------------------------
    // unfolding uc
    // ------------------------------------------------
    if(!d) uc_4 = false; else uc_4 = uc_3;
    if(!d) uc_3 = false; else uc_3 = uc_2;
    if(!d) uc_2 = false; else uc_2 = uc_1;
    if(!d) uc_1 = false; else uc_1 = uc_0;
    uc_0 = true;
    // ------------------------------------------------
    // unfolding ud
    // ------------------------------------------------
    if(!c) ud_4 = false; else ud_4 = ud_3;
    if(!c) ud_3 = false; else ud_3 = ud_2;
    if(!c) ud_2 = false; else ud_2 = ud_1;
    if(!c) ud_1 = false; else ud_1 = ud_0;
    ud_0 = true;
}
drivenby t0 {
    pause;
}
```

But the example extended with the loop formula and also additionally with the unfounded set condition as shown in $P4'$ is still not constructive, respectively has still some values left $\perp$ after the interpretation with the current Quartz interpreter.

When taking a deeper look into the example, one can realize that with the current Quartz interpreter, the monotonic fixpoint computation does not lead to a constructive result. As all variables somehow depend on the others, all variables are in the can-set, but no value is concretely assigned a value with the must-set. Only the starting values for the unfounded set computation ua_0, ub_0, uc_0, ud_0 evaluate to concrete values true and the second iteration variable ub_1 too, because it is the only variable whose unfounded set condition of the iteration two can be computed just by the starting values in the first iteration.

But all other values, not only the unfounded set conditions, but all variables, recursively depend in loops to each other. This will let a monotonic fixpoint fail during computation. The only way to overcome this issue would be to *choose* variable values and make the computation non-monotonic and from a semantics point of view especially also non-deterministic. Of course it would somehow be possible to encode this *choose* operations and an accord-

ing backtracking if a value contradicting the other rules in Quartz itself was chosen; but that would finally mean to re-implement a SAT solver or Answer Set Solver in Quartz.

3.2.2. Quartz as Frontend for Answer Set Programming

Instead of bringing the computation of stable models into the synchronous world, the more practical approach is to use the available tools, knowledge and optimizations from state-of-the-art answer set solvers directly. On the one hand, as answer set programming is based on the concept of stable models, this would allow to interpret every synchronous program directly with stable model semantics. On the other hand, this allows to give a high-level language access to the much more theoretical language of ASP systems. To this end, this section describes two different approaches to translate synchronous programs directly to the standardized answer set programming input language format ASP-Core-2. The first option will show, how one can translate synchronous programs into the default ASP-Core-2 language format. And the second option will use the latest enhancements of answer set and translates them into an inductive definition, in which every step of the original program is then described as an induction step in ASP.

Translation to Default ASP-Core-2 Language

Synchronous languages can be transformed into the intermediate common representation of synchronous guarded actions. Those guarded actions $\gamma \Rightarrow \alpha$ consist of a guard γ which must evaluate to `true` in the current environment, such that the action α fires. This representation is already very close to the representation in ASP, where a program consists of a set of rules $\mathcal{H} \Leftarrow \mathcal{B}$, which are composed of a head $\mathcal{H}$ and a body $\mathcal{B}$ and which can be read as if the body $\mathcal{B}$ can be deduced, then also the head $\mathcal{H}$.

The naive approach would now take every guarded action $\gamma \Rightarrow \alpha$ and transforms it to the rule for ASP with $\alpha \Leftarrow \gamma$, such that the head $\mathcal{H} = \alpha$ and the body $\mathcal{B} = \gamma$. But as always, the devil's in the details.

A (normal) action in the language of synchronous guarded actions is either an immediate assignment `x=`τ or a delayed assignment **`next`**`(x)=`τ. Hereby, the expression τ is typed with the type of `x`. The guard γ itself is always a Boolean expression over the set of variables and constants in the program. At the same time, in ASP a head is a literal and the body itself consists of a conjunction of other literals. This hinders the direct transformation from general guarded actions to ASP, as the expressions of synchronous actions are not necessarily conjunctions. Further, for some special types like arrays or tuples, corresponding representations in the ASP-core-2 syntax have to be defined and the connection from the action α and the guard γ has to be considered as well. Nevertheless, the basic idea of translating every synchronous guarded action by itself into the syntax of the ASP-Core-2 language stays the same.

In detail, every guarded action $\gamma \Rightarrow$ `x|`**`next`**`(x)=`τ can be transformed to a rule `x(V)|`**`next`**`(x(V))` $\Leftarrow \gamma; \tau$`(V).` in ASP. It is to read as follows: if the

condition γ is fulfilled and if then the expression τ evaluates to the concrete `V`, the variable `x` also evaluates to the value `V`. For Boolean expressions, the value `V` can be omitted and then the rule gets `x|next(x)` $\Leftarrow \gamma; \tau$.

The important part now is the translation of the expressions themselves.

Boolean Expressions Consider e.g., the following Boolean expression: `next(x)=case(x2|x23:true, x3:true, default:false)`, which can appear in the intermediate representation as a guarded action in the system of Averest. This expression cannot directly be translated to ASP, as ASP neither has a case statement, nor it is possible to have (nested) disjunctions in rule bodies directly.

Therefore, one option would be to translate the Boolean expression first into a disjunctive normal form $x = \phi_1 \vee \phi_2 \vee \ldots \phi_n$ first, and then every disjunctive part ϕ_i into an own rule in ASP:

$$\begin{aligned} x &:- \phi_1. \\ x &:- \phi_2. \\ x &:- \ldots. \\ x &:- \phi_n. \end{aligned}$$

But calculating the minimal DNF is on the one hand NP-complete and on the other hand can lead to an exponential explosion of the formula. Therefore, a simple trick can be used to omit this step. Instead of calculating the DNF, all sub-expressions can be given unique names, such that the focus of the translation procedure can just lie on single operations instead of complete formulas. The formula `next(x)=case(x2|x23:true, x3:true, default:false)` can e.g., lead to the following rules:

$$\begin{aligned} next(x) &:- x2\_or\_x23. \\ next(x) &:- x3. \\ x2\_or\_x23 &:- x2. \\ x2\_or\_x23 &:- x23. \end{aligned}$$

Hereby, a new variable `x2_or_x23` is introduced for the only non-trivial sub-expression `x2|x23`. Then, this sub-expression itself is also transformed to ASP.

In the following, the translation for the different operations of Boolean expressions are shown, whereas it is accordingly assumed that every expression is given a unique name x. A variable access in a Boolean expression would lead to the following two possibilities, dependent on the definition type of the variable x.

AIF expression	ASP rules
`x = BoolVar qn` (immediate)	x :- v_qn.
`next(x)= BoolVar qn` (delayed)	next(x) :- v_qn.

The Boolean constants (**`true`** and **`false`**) in the AIF type system can be translated to the ASP constants of the type bool, #true and #false.

AIF expression	ASP rules
`x = BoolConst true`	x :- #true.
`x = BoolConst false`	x :- #false.

This step could be simplified and replaced in ASP by the fact `x.` for the constant `true` and the constraint `:- x.` for the constant `false`, but the extensions defined in the next sections, especially for arrays, are easier to define if every rule has left and right-hand sides in ASP.

Next, let's define the default Boolean expressions of Boolean negation, conjunction, disjunction, implication and the if-then-else statement.

AIF expression	ASP rules
`x = BoolNeg bexp`	x :- not bexp.
`x = BoolConj(bexp1,bexp2)`	x :- bexp1; bexp2.
`x = BoolDisj(bexp1,bexp2)`	x :- bexp1. x :- bexp2.
`x = BoolImpl(bexp1,bexp2)`	x :- not bexp1. x :- bexp2.
`x = BoolIte(bExp,bExpIf,bExpElse)`	x :- bexp; bExpIf. x :- not bexp; bExpElse.

Those expressions are recursively defined over the according sub-expressions, as for the negation `BoolNeg` the sub-expression `bexp`. On the ASP side of the translation, of course these sub-expressions must be translated accordingly and as mentioned they must be given a unique name in order to avoid building the full DNF for the (ground) formula.

The conjunction of two expressions can be defined with one ASP rule: both sub-expressions `bexp1` and `bexp2` must be fulfilled in order to satisfy the current variable `x`. The disjunction can be defined accordingly with two rules: the first rule states that `x` holds if the first sub-expression `bexp1` is `true`, whereas the second rule is defined in a same matter for the second sub-expression `bexp2`. The implication uses the Boolean transformation $\texttt{a} \longrightarrow \texttt{b} \equiv \texttt{!a} \vee \texttt{b}$. And the If-then-else expression includes the condition accordingly to the if- or else-part of the expression $\textbf{if}\ \texttt{a}\ \textbf{then}\ \texttt{b}\ \textbf{else}\ \texttt{c} \equiv (\texttt{a} \wedge \texttt{b}) \vee (\texttt{!a} \wedge \texttt{c})$.

Furthermore, the AIF system contains a case expression of the form `BoolCase (caselist,default_val)`. Hereby, the `caselist` represents a list of pairs of Boolean conditions on the one side, and a concrete Boolean value expression on the other `caselist = [(cond_1, bexp_1), ... , (cond_n, → bexp_n)]`. The default value `default_val` represents the value for the expression, if none of the case conditions is evaluated to `true` in the current environment. The Boolean case expression can be handled in the translation to ASP similar to the disjunction, where every entry in the caselist represents a disjunction term by combining the condition and the attached Boolean expression for every entry in the list. And finally, the `default_val` expression is

attached with a conjunction of all negated case conditions to explicitly show that the expression evaluates to this value in case that every case condition evaluates to false.

AIF expression	ASP rules
`x = BoolCase (`	
`[`	
`(cond_1,bexp_1),`	x :- cond_1; bexp_1.
...	...
`(cond_n,bexp_n)`	x :- cond_n; bexp_n.
`],`	
`default_val`	x :- not cond_1; ...; not cond_n; default_val.
`)`	

With those translations, a new look into the example formula **`next(x)=`** → **`case(`**`x2|x23:`**`true`**`, x3:`**`true`**`, `**`default:false`**`)` can be taken.

$$
\begin{aligned}
v\_asp\_1 &:- v\_x2.\\
v\_asp\_1 &:- v\_x23.\\
v\_asp\_2 &:- \#\mathsf{true}.\\
next(v\_x) &:- v\_asp\_1; v\_asp\_2.\\
v\_asp\_3 &:- v\_x3.\\
next(v\_x) &:- v\_asp\_3; v\_asp\_2.\\
v\_asp\_4 &:- \#\mathsf{false}.\\
next(v\_x) &:- not\ v\_asp\_1; not\ v\_asp\_3; v\_asp\_4.
\end{aligned}
$$

This is the result of a prototypical compiler, that has been written and is more precisely described in Section 3.2.2.

In contrast to the previously introduced simplified example solution (see next listing), the following changes can be determined.

$$
\begin{aligned}
next(x) &:- x2\_or\_x23.\\
next(x) &:- x3.\\
x2\_or\_x23 &:- x2.\\
x2\_or\_x23 &:- x23.
\end{aligned}
$$

- all newly introduced variable names during the compilation are prefixed with *asp*
- the constants #true and #false are added
- the default case statement is made explicit by negating all case conditions. This is especially needed if the default value itself is an expression like the so-called *carrier variables* in Averest
- the expression on the right-hand side of every case statement is added, as this must not always be the constant true, but a full expression `bexp_i`

- asp-variable names are reused, when a new variable name is introduced during the compilation and the same expression can be found on multiple places in the Averest program.

Number Expressions In the AIF system Averest supports expressions of other types than Boolean. Especially number expressions can be defined in Quartz programs and they are translated to according synchronous guarded actions. Numbers can be of the concrete type integer, natural number or real number. The ASP-Core-2 language supports out of the box integers and natural number expressions. Those expressions can be translated from the AIF system into ASP-core-2 language rules, similar to the Boolean expressions.

Instead of defining Boolean variables `x` as basic Boolean literals `v_x`, number expressions must also hold their value `V` and can be expressed by simple function terms `x(V)` with the meaning that `x` evaluates to the value `V` if all other conditions of the rules are fulfilled. Especially, the rule contains according constraints to `V` such that the original meaning is obtained.

AIF expression	ASP rules
`x = NatConst const`	x(const) :-#true.
`x = NatVar (qn,_)`	x(V) :- v_qn(V).
`x = NatAdd (ln,rn)`	x(X+Y) :- ln(X); rn(Y).
`x = NatMul (ln,rn)`	x(X*Y) :- ln(X); rn(Y).
`x=NatIte(bExp,nIf,nElse)`	x(V) :- bexp; nIf(V). x(V) :- not bexp; nElse(V).
`x = NatSub (ln,rn)`	x(X-Y) :- ln(X); rn(Y).
`x = NatDiv (ln,rn)`	x(X/Y) :- ln(X); rn(Y).
`x = NatMod (ln,rn)`	x(X\Y) :- ln(X); rn(Y).
`x = NatExp (ln,rn)`	x(X**Y) :- ln(X); rn(Y).

`x = NatCase (`
`[`
`(con_1, nexp_1),`
`...`
`(con_n, nexp_n)`
`],`
`nExp`
`)`
	
	x(V) :- con_1; nexp_1(V).
	...
	x(V) :- con_n; nexp_n(V).
	
	x(V) :- not con_1; ...; not con_n; nExp(V).

The default expressions on natural numbers like for example the subtraction `NatSub(ln,rn)` can be directly translated into ASP by defining the resulting value V according to the corresponding arithmetic expression in ASP, e.g., `V=X-Y` for the subtraction if the expression on the left evaluates in the current computation to the concrete value `X` in `ln(X)` and the right-hand side expression to `Y` in `ln(Y)`. Constants can directly be expressed by facts, e.g., the assignment `x=5` will lead to the fact `x(5).` in ASP. Again, the constant #true is added to the ASP body of the corresponding rule, so that it is easier in the

following to define array expressions. The case-expression and the if-then-else expression can be defined quite similar to the Boolean corresponding expressions; only the concrete value is now of type number instead of bool, which means the concrete value `V` has to be added.

By allowing number expressions to be translated from AIF to ASP, also the corresponding equality checks should be translated. Therefore, the following translation schemes for the additional Boolean expressions are introduced:

AIF expression	ASP rules
`x = NatEqu (ln,rn)`	x :- ln(V); rn(V).
`x = NatLes (ln,rn)`	x :- ln(X); rn(Y); X<Y.
`x = NatLeq (ln,rn)`	x :- ln(X); rn(Y); X<=Y.

The equality of two number expressions `ln` and `rn` can be expressed in ASP by defining that the equality is only `true` if both expressions evaluate to the same value `V` in the concrete answer. The less and less-or-equal expressions can be translated by using the corresponding comparators in ASP < and <=.

Having defined those expressions, the following number expression can e.g., be translated **`next`**`(x)=`**`case`**`(x10:x1+1, x8:x1+1, `**`default`**`:x1)`.

$$
\begin{aligned}
v\_asp\_1(1) &:- \#\mathsf{true}.\\
v\_asp\_2(X+Y) &:- v\_x1(X); v\_asp\_1(Y).\\
next(x(V)) &:- v\_x10; v\_asp\_2(V).\\
next(x(V)) &:- v\_x8; v\_asp\_2(V).\\
next(x(V)) &:- not\ v\_x10; not\ v\_x8; v\_x1(V).
\end{aligned}
$$

For this example, two new variables are introduced for the corresponding non-trivial sub-expressions. The first variable `v_asp_1` stands for the constant value 1. The second variable `v_asp_2` is the addition of `x1` with this constant 1. With those variables, the case statement can be expressed, whereas both case statements result in an own rule each and the default statement with all negated conditions in a third rule. The variable `x` should now contain exactly one concrete value for the next step if the original program was constructive also in Quartz/Averest.

Array Expressions As a last step, the translation of array expressions is discussed. This is especially useful as arrays are not directly supported in the original language of ASP-Core-2. Therefore, having this translation, besides all other statements in Quartz like if-then-else constructs or abort and loop statements, gives a huge benefit for programmers who want to access the power of ASP without the need to deeply dive into the theoretical parts of ASP, logic programming and logic rules. It enables them to use an abstract, more programmer-friendly language, such as the synchronous language Quartz to describe problems more easily.

Arrays in the AIF system are always of a known, fixed and constant size. The basic idea for the translation of array expressions to ASP-Core-2 languages

is to add a position to the already known variable definition `x(V, POS)`. This can be read as *the array x evaluates to value `V` at position `POS`*. Having the according additional constraints on `POS`, enables to define possibly different values for every position.

The assignment to a complete array variable can be defined as follows:

AIF expression	ASP rules
`x = ArrVar (qn,(size,Qbool))`	x(POS) :- v_qn(POS).
`x = ArrVar (qn,(size,Qnat))`	x(V, POS) :- v_qn(V, POS).
`x = ArrVar (qn,(size,Qarr))`	x(V, POS) :- v_qn(V, POS).

Hereby, the current variable `x` is assigned the same value `V` for every position `POS` of the assigned array variable with the name `qn`. The value `V` can again be omitted if the type of the values in the array is Boolean.

Arrays can also be defined as list of expressions of the arrays value type. Such a list can be transformed into single rules describing the arrays value at every position 0 up to the size of the list n.

AIF expression	ASP rules
`x = ArrOfExprL (`	
`[`	
`exp_0,`	x(0, V) :- exp_0(V).
`...`	...
`exp_n`	x(n, V) :- exp_n(V).
`])`	

If the expressions are of type bool, the concrete value `V` can be ignored when translating an array being a list of expressions.

Similar to the other types, array expressions are possible to appear as case statements and if-then-else statements. Again, the position of the values has to be added to the according expressions in ASP.

AIF expression	ASP rules
`x=ArrIte(bExp,aIf,aElse)`	x(V, POS) :- bexp; aIf(V, POS). x(V, POS) :- not bexp; aElse(V, POS).
`x = ArrCase (`	
`[`	
`(cond_1,aexp_0),`	x(V, POS) :- cond_1; aexp_0(V, POS).
`...`	...
`(cond_n,aexp_n)`	x(V, POS) :- cond_n; aexp_n(V, POS).
`],`	
`default_val`	x(V, POS) :- not cond_1; ...; not cond_n; default_val(V, POS).
`)`	

Arrays cannot only be assigned to variables at once; the most important part is the access to specific array positions like `x = a[POS]` itself. The type of an array access depends on the concrete type of the elements in the array. For Boolean array, the type of `x` is bool and therefore also the access operation `BoolArrAcc(aExp, nExp)` is a Boolean expression. All array access operations `<Type>ArrAcc(aExp, nExp)` have the array to be accessed (`aExp`) and a natural number expression `nExp` describing the position of the access as parameters.

AIF expression	ASP rules
`x = <Type>ArrAcc(aExp,nExp)`	x(V) :- aExp(V, POS), nExp(POS).

Such a typed array expression can be formulated in ASP by stating that the variable `x` evaluates to a value `V` if the given array evaluates to `V` at the position `POS`. Furthermore, the given natural number expression has to evaluate to that position `POS` in the current computation.

Arrays can also be nested, e.g., `x = a[1][2];`. Therefore, a representation for nested arrays has to be defined in ASP. The chosen representation adds nested positions for nested array expressions to the according ASP expression: `x(V, (POS1, (... (POSN-1, POSN)...)))`. This especially means that an array variable `x(V, P)` can stand for a position `P` as a natural number expression `P=N` or as a nested access `P=(POS1, (..., POSN))`. The access to such nested expressions can be defined as follows:

AIF expression	ASP rules
`x=ArrArrAcc(aExp,nExp)`	x(V,P) :- aExp(V,(POS0,P)), nExp(POS0).

The array access which returns an array can be read like explained in the following: If the given index expression `nExp` evaluates to the concrete value `POS0` then collect all `V` which are available at `POS0` in the nested array (collect all sub-positions `P`) and construct the new array, while the position `P` is then the index of the accessed array.

Instead of accessing concrete positions, also concrete positions can be assigned a value, like `a[2][1] = 19;`. For those expressions, the index expressions from the left-hand-side must be considered in the translation to an ASP rule as well.

AIF expression	`x[nExp0]...[nExpn] = <typed>Exp`

ASP rules	`x(V, (POS0, (..., (POSN-1, POSN)...)))`
	`:- <typed>Exp(V), nExp0(POS0), nExp1(POS1), ...,`
	`nExpN-1(POSN-1), nExpN(POSN).`

With those translation schemes, all array operations can be translated to ASP. For example the assignment expression `a[2][1] = 19;` will lead to:

$$\begin{aligned} v\_asp\_1(1) &:- \#\text{true}. \\ v\_asp\_2(2) &:- \#\text{true}. \\ v\_asp\_3(19) &:- \#\text{true}. \\ v\_a(X,(POS0,POS1)) &:- v\_asp\_3(X); v\_asp\_2(POS0); v\_asp\_1(POS1). \end{aligned}$$

Here, the constant numbers 1, 2, and 19 are defined first. Then, the variable v_a is assigned the value X at the nested position, where the constant 2 and constant 1 respectively are available. The access to an array position `x = → a[1][2];` will end in:

$$\begin{aligned} v\_asp\_1(1) &:- \#\text{true}. \\ v\_asp\_2(2) &:- \#\text{true}. \\ v\_asp\_3(X,POS) &:- v\_a(X,POS). \\ v\_asp\_4(X,POS1) &:- v\_asp\_1(POS0); v\_asp\_3(X,(POS0,POS1)). \\ v\_asp\_5(X) &:- v\_asp\_2(POS); v\_asp\_4(X,POS). \\ v\_x(X) &:- v\_asp\_5(X). \end{aligned}$$

Here, the intermediate variable v_asp_4 stands for the outer access at position 1 and returns the corresponding inner array as the result. The intermediate variable v_asp_5 accesses this array and stands for the inner access at position 1. It returns the final value X, which is finally assigned to the variable v_x.

Evaluation

All in all, the shown translation scheme converting AIF expressions to ASP-core-2 expressions allows to evaluate corresponding Quartz programs with the stable model semantics by using state-of-the-art ASP solvers. An accordingly translated ASP-Program allows to compute one system reaction if a current environment, including the program state and the input variables, is given.

Definition 3.10 ⟨ Quartz Interpreter with ASP ⟩

function $InterpretQuartzWithASP(S)$
  $\mathcal{E}_{\text{pre}} := \{\mathcal{E}_{\text{def}}\}$; *// default values for all variables*
  $\mathcal{S}_{\text{pre}} := \{S\}$;
  do
    $\mathcal{E}_{\text{in}} :=$ ReadInputs();
    $\mathcal{E}_{\text{init}} :=$ UseInputs$(\mathcal{E}_{\text{in}}, \mathcal{E}_{\text{pre}})$;
    $\{\mathcal{E}\} :=$ ComputeReactionsAsAnswerSets$(\mathcal{E}_{\text{init}}, \mathcal{S}_{\text{pre}}, \mathcal{E}_{\text{pre}})$;
    if $\exists x \in \mathcal{V}.\, \mathcal{E}(x) \in \{\bot, \top\}$ then fail;
    $\{(S', \mathcal{D}, t)\} :=$ TransSOSASP$(\mathcal{S}_{\text{pre}}, \{\mathcal{E}\})$;
    $\mathcal{S}_{\text{pre}} := \{S'\}$;
    $\mathcal{E}_{\text{pre}} := \{\mathcal{E}\}$;
  while$(\exists t.\ \neg t)$;

Our according interpreter could be enhanced by using the ASP solver to compute the reaction (see pseudo-code above). As answer set programming can lead to potentially multiple correct answers, every answer can be taken into consideration by itself. This can be done by wrapping all computations into sets of results, as shown in the pseudo-code. Here, especially the set of states $\mathcal{S}_{\mathsf{pre}}$ represents all currently reachable states with the computed answer sets and the set of according environments is stored in the variable $\mathcal{E}_{\mathsf{pre}}$. The interpreter can stop if all termination flags t of all computations are reached. The function TransSOSASP(...) hereby stands for the default TransSOS(...) applied to each element given in the set of states and environments.

The function ComputeReactionsAsAnswerSets(...) uses the translated program $\mathcal{P}_{ASP}$ and enriches it with the information about the input variables and the currently examined state $S \in \mathcal{S}_{\mathsf{pre}}$. As all values of the environments and states have a known value V, all according variables can be added to the ASP program as ASP facts $y_i(C_i)$, while C_i stands for the according constant value for the variable:

```
// Translated Program
v_asp_1(X) :- ....
...
v_asp_N(X) :- ....
x_1(V) :- ....
...
x_n(V) :- ....
next(y_1(V)) :- ....
...
next(y_m(V)) :- ....

// Current state
y_1(C_1).
....
y_m(C_m).

// Current inputs
i_1(C2_1).
....
i_l(C2_l).
```

The translation scheme shown in the last section has been implemented in a prototypical compiler. The compiler takes an (equalized) AIF program and compiles it to an equivalent ASP encoding.

Let‘s take as an example the following program from the background chapter:

```
module P3‘(event x1,x2,x3,x4,x5,x6,x7,x8){
   emit(x1);
   if(x4&!x5) emit(x2);
   if(x5&!x6) emit(x2);
   if(!x2) emit(x3);
   if(x2) emit(x4);
   if(x4) emit(x5);
   if(!x8) emit(x7);
   if(!x7) emit(x8);
 }
```

It compiles to the following intermediate representation of synchronous guarded actions:

```
system P3‘:
   abbreviations:
      true => __lvar000 = x4&!x5
      true => __lvar001 = x5&!x6
   control flow:
      true => next(____running000) = true
   data flow:
      true => x1 = case(!____running000: true, default: false)
      true => x2 = case(__lvar001&!____running000: true, →
          __lvar000&!____running000: true, default: false)
      true => x3 = case(!x2&!____running000: true, default: false)
      true => x4 = case(x2&!____running000: true, default: false)
      true => x5 = case(x4&!____running000: true, default: false)
      true => x6 = false
      true => x7 = case(!x8&!____running000: true, default: false)
      true => x8 = case(!x7&!____running000: true, default: false)
```

The corresponding translation to ASP results in the following rules regarding the translation scheme from the last section:

```
%%%%% ASP Program Code %%%%%
% Abbreviations

   asp_lvar_0 :- not v_x5.
   v___lvar000 :- v_x4; asp_lvar_0.
   asp_lvar_1 :- not v_x6.
   v___lvar001 :- v_x5; asp_lvar_1.

% Control Flow

   next(v_____running000) :- #true.

% Data Flow

   asp_lvar_2 :- not v_____running000.
   asp_lvar_3 :- #true.
   v_x1 :- asp_lvar_2;asp_lvar_3.
   asp_lvar_4 :- #false.
   v_x1 :- not asp_lvar_2; asp_lvar_4.
   asp_lvar_5 :- v___lvar001; asp_lvar_2.
   v_x2 :- asp_lvar_5;asp_lvar_3.
   asp_lvar_6 :- v___lvar000; asp_lvar_2.
   v_x2 :- asp_lvar_6;asp_lvar_3.
   v_x2 :- not asp_lvar_5; not asp_lvar_6; asp_lvar_4.
   asp_lvar_8 :- not v_x2.
   asp_lvar_7 :- asp_lvar_8; asp_lvar_2.
   v_x3 :- asp_lvar_7;asp_lvar_3.
   v_x3 :- not asp_lvar_7; asp_lvar_4.
   asp_lvar_9 :- v_x2; asp_lvar_2.
   v_x4 :- asp_lvar_9;asp_lvar_3.
   v_x4 :- not asp_lvar_9; asp_lvar_4.
   asp_lvar_10 :- v_x4; asp_lvar_2.
   v_x5 :- asp_lvar_10;asp_lvar_3.
   v_x5 :- not asp_lvar_10; asp_lvar_4.
   v_x6 :- #false.
   asp_lvar_12 :- not v_x8.
   asp_lvar_11 :- asp_lvar_12; asp_lvar_2.
   v_x7 :- asp_lvar_11;asp_lvar_3.
   v_x7 :- not asp_lvar_11; asp_lvar_4.
   asp_lvar_14 :- not v_x7.
   asp_lvar_13 :- asp_lvar_14; asp_lvar_2.
   v_x8 :- asp_lvar_13;asp_lvar_3.
   v_x8 :- not asp_lvar_13; asp_lvar_4.
```

This example does neither have a solution in Fitting's fixpoint semantics nor in the well-founded semantics. But calling the ASP suite clingo results to the following two different answers, as it was the initial intention of the translation from synchronous programs to ASP:

```
Answer: 1
asp_lvar_3
asp_lvar_2
asp_lvar_1
asp_lvar_0
next(v_____running000)
v_x1
asp_lvar_8
asp_lvar_7
v_x3
asp_lvar_12
v_x7
asp_lvar_11
Answer: 2
asp_lvar_3
asp_lvar_2
asp_lvar_1
asp_lvar_0
next(v_____running000)
v_x1
asp_lvar_8
asp_lvar_7
v_x3
v_x8
asp_lvar_14
asp_lvar_13
SATISFIABLE

Models    : 2
Calls     : 1
Time      : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time  : 0.000s
```

The ASP solver takes just 0.002s to compute the result. It assigns the first answer as $A1 = \{x1, x3, x7\}$ and the second answer $A2 = \{x1, x3, x8\}$.

The prototypical compiler has been run on several more test examples from the Averest benchmark suite. Especially included were the ABRO example for Boolean expressions, the Speed and Malik examples for natural number expressions and the Rivest and the Sudoku example for array expressions.

The compilation results for those programs can be found in the following table. Hereby, the compiler, which was written in the programming language F#, has been running on a Windows 10 machine incl. 48GB of RAM and an Intel i5-6600k processor (4 x 3,5GHz):
Table 3.1 shows the program name in the first column, and in the other columns some compilation information. It contains the compilation time taken to translate the AIF program to the ASP program, then the amount of synchronous guarded actions in the AIF system for which the compilation was started, the overall amount of resulting ASP rules, and the amount of newly introduced abbreviation variables in ASP of the form `asp_lvar_i`. It can be observed that the compilation for small programs is very fast, all far under one second. The amount of guarded actions doubles until triples when compiled to ASP rules for Boolean and number programs. Hereby, approximately

Experimental Compilation Results				
Program Name	Compilation Time	#GAs	#ASP rules	#New Vars
ABRO	0.202411s	29	54	11
Speed	0.206204s	24	49	12
Malik	0.225320s	17	37	10
Rivest	0.261346s	14	93	47
Sudoku	69.841374s	5428	53523	17396
P3‘	0.234527s	11	34	15

Table 3.1.: *Experimental compilation results*

1/3 of the ASP rules are newly introduced abbreviation variables having been introduced in order to create the full expression tree. But it is also visible, that the picture seems to be a bit different for the compilation of programs containing arrays (Rivest and Sudoku). The amount of guarded actions has nearly increased tenfold from 5429 guarded actions to 53523 ASP rules in the Sudoku example. This can be explained by the deeply nested arrays contained here. Further, arrays are unrolled during compilation from guarded actions to ASP resulting in a quite immense blowup the deeper the arrays are nested. This blowup also explains the huge increase in compilation time of the Sudoku example. Nevertheless, this unrolling of arrays does not have a huge influence on the actual program simulation and result calculation, as it is indicated in the following paragraph.

The compiled ASP programs have been used to be interpreted on some useful environments, like for the ABRO example with the state w_a = `true` and the input variable b = `true`. The following table contains the interpretation/simulation times for those examples using the answer set suite clingo 4.5.4 to compute the system reaction on the same machine as the compiler was running on:

Simulation Result		
Program Name	#ASP rules	ASP Answer in
ABRO	54	0.015s
Speed	49	0.020s
Malik	37	0.011s
Rivest	93	0.003s
Sudoku	53523	2.400s
P3‘	34	0.002s

Table 3.2.: *Experimental simulation results*

Table 3.2 shows the feasibility to use ASP as a simulator for synchronous programs. Although the Sudoku was quite huge with 53523 ASP rules, solving the Sudoku with a given predefinition of the grid just took 2.4 seconds with ASP, whereas the available simulator in Averest takes 15.147s to find the solution. All other examples took just a few ms to calculate the results and it is not even realized as a delay when computing the simulation result for those.

Full Simulation

So far, translated programs in ASP allow to interpret one macro step of synchronous programs given a current environment. This can be extended to a full simulation/ interpreter by introducing a counter representing the current execution step.

This can be archived by representing all immediate expressions/assignments `exp(V)` by `exp(V,N)`, with `N` representing the current execution step. Furthermore, delayed assignments `next(exp(V))` can be represented by `exp(V,N+1)` to state that the expression has to hold in the next execution step `N+1`.

On top of that, the ASP solver clingo contains latest research results via the so-called multishot solving. The idea of multishot solving is that problems, which can be separated into several steps, can be solved iteratively. Like in an inductive proof, the definition of such multi-shot programs contains an induction base containing the initial state. Further, a definition of an induction step has to be given. And finally, a check function is part of such a program, to define conditions when to end the solving process.

For our translation, the base program can contain constants/facts, like for example the maximal execution step counter or a driver for the current program:

```
#include <incmode>.
#program base.
max_prog_counter(150).
// driver
a(140).
b(145).
r(147).
```

The induction step part contains the program translation as described in the latest sections, defined in dependency to the current execution step k.

```
#program step(k).
//control flow :
____running003(k+1).
wa(k+1) :- not ____running003(k).
wb(k+1) :- not ____running003(k).
wa(k+1) :- not r(k), wa(k), not a(k).

wa(k+1) :- r(k), wr(k).
wa(k+1) :- r(k), wa(k).
wa(k+1) :- r(k), wb(k).

...

// data flow :
o(k) :- not r(k), a(k), wa(k), b(k), wb(k).
o(k) :- not r(k), not wa(k), b(k), wb(k).
o(k) :- not r(k), not wb(k), a(k), wa(k).
```

And finally, the check part allows to add an ending condition. In general cases, it could be added for example, that the maximum defined program counter is not exceeded.

```
#program check(k).
:- query(k), k<L; max_prog_counter(L).
```

In Averest, it is possible to define assertions and assumptions as Boolean expressions on the program variables. Those assertions can also be added to the check part in order to stop simulating/running as soon as an assertion fails.

```
#program check(k).
// assertions
new_assert_var_1(k) :- query(k); ....
:- not new_assert_var_1(k); query(k).
...

new_assert_var_n :- query(k); ....
:- not new_assert_var_n(k); query(k).

// max counter
:- query(k), k<L; max_prog_counter(L).
```

All in all, this extension to the multi-shot-solving in ASP allows a full simulation of a synchronous program with the stable model semantics, and can even be enriched with LTL formulas, when using the newest research results from [ACPV11; CaDi11; CKSS18; CaSc19].

Chapter 4

Optimized System Design with Stable Models

Contents

4.1. Synthesis of Optimal Interconnection Networks 70
4.1.1. Experimental Results 72
4.2. Optimal Code Generation for SCAD 74
4.2.1. aif2lp 77
4.2.2. lp2mc 83
4.2.3. Experimental Results 90
4.3. Enhancing Synchronous Programs by Stable Models 92
4.3.1. Disjunctive Programming 94
4.3.2. Constraints 94
4.3.3. Choices 96
4.3.4. Aggregates 97
4.3.5. Optimizations 98

The last chapter showed how to interpret synchronous programs with stable model semantics. In this chapter the focus lies on the application of stable model semantics by the means of two larger examples written for answer set programming. Both examples have been developed to solve (optimization) problems in the area of system design and especially show the descriptive nature of ASP programs: the first example solves the problem of finding an interconnection network with a concentrator property while being minimal in the sense of used switches in the network. The second example is a complete compiler from Quartz to move code for the SCAD architecture, optimizing the produced code in the sense of finding the best distribution to all SCAD processing units.

4.1. Synthesis of Optimal Interconnection Networks

In general, an interconnection network connects input ports and output ports by a network circuit of so-called switches, which can either route the one way or the other (see Figure 4.1). The idea is, that such a network transmits data from the inputs to the outputs, whereby every switch in the network has got a concrete position and thereby influences, where the corresponding input is transmitted to, and finally, which input ports are routed to which output ports.

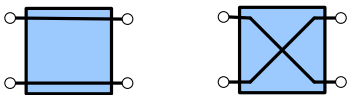


Figure 4.1.: *Switch configurations*

Interconnection networks can be categorized with the help of special properties, e.g. a network is *sorting* if the network can sort every combination of values at the input ports to a sorted output at the output ports.

The property to be a (N, M)-concentrator [Pins73] circuit with N inputs and $M \leq N$ outputs is defined as being able to route any given number $K \leq M$ of valid inputs to K of its M outputs. Hereby, it is not important which of the K out of the M outputs are selected and how the inputs are mapped to the output ports. But no matter which $K \leq M$ inputs are valid, it must be possible to map them to the same K of the M outputs. Every sorting network is of course also a concentrating network, however the practically best known sorting networks [Batc68; Parb92] require $O(\log(N)^2)$ depth and $O(N \log(N)^2)$ size (in terms of comparators) and therefore do not lead to competitive concentrator designs. The best known practically used concentrator implementations are the so-called permutation networks (see references in [JaSc16]) which can have size $O(N \log(N))$ and depth $O(\log(N))$. Still, it is known that those permutation networks contain switches not needed for obtaining the concentrator property ([QuWi05]), which leads to the question how absolutely minimal concentrators look like.

As interconnection networks are often needed for chip and system implementation, the real size has a lot of impact to the final chip size. This optimization problem, to find a minimal concentrator network in the sense of needed switches or minimum network levels, is presented in the following as an ASP program. Those results have been published in [DJSG17].

The ASP definition of the problem starts with the overall goal definition to minimize the switches in the network with an according optimization statement:

$$\#minimize\{N : num\_switches(N)\}.$$

Furthermore, the program can be configured with three constants: the amount of inputs N, the amount of outputs M to be used for the concentrat-

ing property and the constant *max_switches* for the upper bound of switches to limit the search space.

$$\begin{aligned} \#const\ N &= 3. \\ \#const\ M &= 2. \\ \#const\ max\_switches &= 5. \end{aligned}$$

The next part of the program defines the basic nodes as N inputs and output ports of the network. Additionally, all last M of the outputs are defined to be used for concentrating. All those concentrating target outputs must later on be reachable from all possible $K \leq M$ out of N inputs. It is sufficient to find a concentrator for $K = M$, because this concentrator maintains the concentrator property for all $K \leq M$.

$$\begin{aligned} in(1..N). \\ out(1..N). \\ used\_output(Out) \ :- \ out(Out), Out > (N - M). \end{aligned}$$

As a next step, the amount of switches are chosen with an according choice statement. For every answer in the answer set, the ASP solver chooses exactly one amount of switches and together with the overall minimizing goal, this amount is minimal regarding all other constraints of the program.

$$\begin{aligned} &possible\_switch\_amount(0..max\_switches). \\ &1\ \{num\_switches(N) : possible\_switch\_amount(N)\}\ 1. \\ &switch(1..N)\ \texttt{:-}\ num\_switches(N),\ N > 0. \end{aligned}$$

So far, the definitions have been given. Without more constraints, this network would minimize to size 0.

Let's consider the concentrator property: it must be defined that for every combination of picking $k = M$ out of the N inputs it must be possible to find an according routing or more precise to find an according set of switch configurations which maps those inputs to the outputs marked as being used for concentrating.

To this end, first it is assumed that all combinations of picking the $k = M$ inputs are collected in a variable C and that a predicate $contains(In, C)$ is available, which is only `true` if the given input port is in the chosen input combination C. With that help, for every combination C the according goals can be defined by mapping those chosen inputs to an output marked to be used for concentrating:

$$\begin{aligned} &1\ \{goal(C, In, Out) : used\_output(Out)\}\ 1\ \texttt{:-}\ contains(In, C). \\ &\texttt{:-}\ goal(C, In, Out), goal(C, In2, Out), In \neq In2. \end{aligned}$$

The first line states that for every input In contained in an combination C, exactly one *used_output* must be chosen. The second line restricts that chosen goal with an according constraint, such that all inputs In of a combination C must be mapped to different outputs Out by disallowing the output Out to be the same in the according goals, when the inputs In and $In2$ differ. All in

all, there are $\binom{N}{M}$ combinations of picking M inputs out of I available inputs, which finally sums up to $\binom{N}{M} \cdot M$ goals.

Each of these $\binom{N}{M} \cdot M$ goals must be fulfilled by the network searched as answer set. Furthermore, every combination C can lead to a different switch configuration of the network. Therefore, the ASP solver must find $\binom{N}{M}$ different switch configurations for the network.

This leads to the following two restrictions, which describe that for every goal the outputs that are defined in the goals must be reachable with the given network. Hereby, the first line demands that only one output is reachable from every input and the second line demands that this output is the output which is defined in the according goal for the input, by considering the according combination C.

$$:- \quad goal(C, In, O), \#count\{1, Out : reachable(C, in(In), out(Out))\} = N, \\ N \neq 1.$$
$$:- \quad goal(C, In, Out), reachable(C, in(In), out(Out2)), Out2 \neq Out.$$

The last needed definition is the predicate *reachable*. This predicate is dependent on the chosen input combination C as another switch configuration of the network is possible for every combination. To this end, the following recursive definition is chosen to define the reachability: A node D is reachable from an input In for a combination C if

- either D is directly connected to the input In
- or the predecessor of D is a switch S already reachable: the input port IV of S is already reachable from In and IV routes for the combination C (with switch position SV) to the node D

This definition can be expressed with the following ASP code:

$$\begin{aligned} reachable(combination(ID), in(In), D) \quad &:- connection(in(In), D), \\ &\quad\ combination(ID). \\ reachable(C, In, D) \quad &:- connection(switch(S), SV, D), \\ &\quad\ switchConfig(C, S, IV, SV), \\ &\quad\ reachable(C, In, switch(S, IV)). \end{aligned}$$

With those definitions, the minimal concentrating networks can be searched with ASP. Additionally, one could add some more constraints to ease the search and help the solver by adding additional constraints, like that every node S cannot be reached from different inputs:

$$:- \quad reachable(C, In, S), reachable(C, In2, S), In \neq In2.$$

4.1.1. Experimental Results

As shown in [DJSG17], an experiment with the above ASP encoding has been executed. As the ASP solver, clingo 4.5.4 has been used on a machine running Ubuntu 16.04.1 LTS with a i5-6600@3.30GHz CPU and 16 GB memory

installed. The process used approximately 2GB of memory during the experiment. For the execution of the experiment, the ASP solver clingo had been called repeatedly with different constant numbers for N and M. The generated minimal concentrators can be found in Table 4.2 for different N and M and Table 4.1 completes the experimental results with the required runtime. It is differentiated into the total time, the time needed to prove the optimality and time needed until the first concentrator of that size was found.

N/M	total	optimality	model
5/1	0.26	0.198	0.015
5/2	145.79	145.245	0.354
5/3	303.70	302.949	0.415
5/4	0.88	0.394	0.365
6/1	24.11	23.850	0.087
6/2	33042.85	33031.465	8.439
6/3	317913.72	317906.224	5.799
6/4	60621.53	60594.077	11.865
6/5	113.61	112.970	0.208
7/1	5618.63	5618.438	0.038
7/2	1870617.88 (aborted)	N/A	3.392

Table 4.1.: *Runtime in seconds required for generating optimal concentrators (runtime for $N < 5$ are negligible).*

As it can be observed in the result table, the ASP solver needs nearly no time for all $N \leq 4$. For larger concentrators with $N \geq 3$, an immense blowup of the time needed can be seen. For example, to find a minimal concentrating network for $N = 6$ and $M = 3$ already took more than three days and the computation for $N = 7$ and $M = 2$ was cancelled after three weeks without finding an optimal model (but it had found a possible non-optimal model already after three seconds).

Indeed, the search space of all possible graphs having to be considered as potential concentrators is enormously large: Assume, the final minimal network has $M \leq max\_switches$ many switches. Without further constraints, there exist $2M + N$ input ports. As every input port of a node must be reached, there exist $(2M + N)!$ many possibilities to construct a network with M switches and N inputs and outputs. For every $i \leq max\_switches$, the network must be checked if it is minimal, this sums up to the immense sum of $\sum_{i=0}^{max\_switches} (2i + N)!$.

For $N = 7$ and $max\_switches = 8$, this then yields a total of $2.59 \cdot 10^{22}$ possible networks to be checked. In general, it is sufficient to check less networks, because if one minimal model with M switches exists (computed in X steps with the solving capabilities of ASP solvers), the sum of the networks to be checked for unsat the problem is then $X + \sum_{i=0}^{M-1} (2i + N)!$. Furthermore, one network (invalid or valid) must be checked in the worst case for all $\binom{N}{K}$

possibilities to choose K out of the N inputs, which results in an total upper bound of $\binom{N}{K} \cdot (X + \sum_{i=0}^{M-1} (2i+N)!)$.

The evaluation results on the one hand show the feasibility of the approach and proved that the minimal size of concentrator network is lower than the minimal size of sorting networks [MoSc11]. On the other hand, the results show that due to the large amount of networks to be checked even for small numbers of N inputs, the computation time is quite enormous. When comparing the computation time with the SAT-encoding for minimal sorting networks in [MoSc11], the time needed to find the solution is a bit higher.

Besides showing how minimal networks with a special property, in this case the concentrator property, can be found with and formulated as ASP problem, the shown example especially gives a feeling about the descriptive nature of ASP programming. Instead of writing down an algorithm to find or construct networks, the shown ASP encoding defined the properties that the network must fulfill with the help of descriptive rules and the solver did the work to construct the network according to the given rules.

4.2. Optimal Code Generation for SCAD

The second example tackles the optimal code generation problem for a newly proposed architecture developed in the Embedded Systems Group of the University of Kaiserslautern called SCAD (Synchronous Control Asynchronous Data). SCAD is a dataflow driven architecture, which moves values from output buffers of processing units to input buffers while bypassing registers completely - this allows a highly parallel and data driven computation approach.

A simple code generation for the so-called *move code* can be performed by a breadth-first traversal over its syntax tree [FeEr81]. With the breadth-first traversal, it can be ensured that operands required for the execution of operations at one level of the tree appear in the correct order for the next level of the graph at the head of the input queues.

Often, basic blocks of programs are represented as directed acyclic graphs (DAGs) which can be processed similar to syntax trees. Nevertheless in order to generate move code for a DAG, additional *duplicate* and *swap* operations have to be introduced as overhead, in order to create a level-planar graph [ScLY02] as shown in Figure 4.2

This approach, which is based on code generation for queue machines, was described in [BhJS16], and it was especially observed that every basic block can be executed on a SCAD machine without the need to have those overhead operations. This is due to the fact, that in contrast to queue machines, the SCAD machine has multiple buffers which are associated with multiple PUs, while queue machines are only attached to a single buffer. Therefore, the additional swap and duplicate operations can be omitted in the SCAD machine when having available enough PUs and buffers. This fact finally leads to the question how to compute the best move code for a basic block in the sense of

	N=2	N=3	N=4	N=5	N=6	N=7
M=1						
M=2						(aborted)
M=3						(N/A)
M=4						(N/A)
M=5						(N/A)

Table 4.2.: *Optimal-size concentrators generated by ASP (the outputs with a circle are those where the valid inputs will be routed to).*

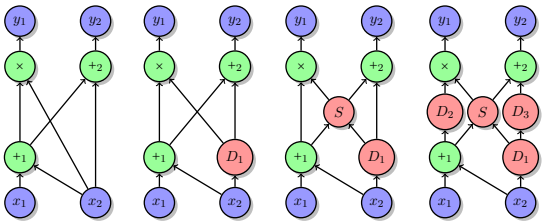


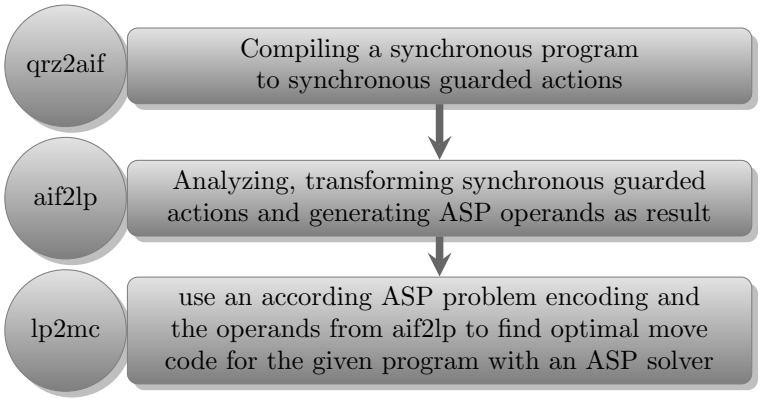
Figure 4.2.: *An expression DAG with its levelized and planarized version, and the final level-planar expression DAG*

maximizing the instruction level parallelism. Furthermore, the question is how large a SCAD machine must be scaled in PUs and buffer size such that a given basic block can be scheduled on this machine without overhead operations.

In [BhSc16] a first SAT encoding of the problem to find a minimal SCAD machine and its move code was shown. This was extended to an SMT encoding in [BhSc17]. Both encodings [BhSc16; BhSc17] lacked automated optimization and especially describing resource constraints quickly became unhandy. Such resource constraints could be to find the minimal runtime for a given number of PUs or nested minimization constraints like as a first criterion to minimize the overall runtime and as a second criterion to reduce the amount of needed PUs.

To this end, an ASP encoding has been defined in [DaBS18b], which allows to automate the optimization problem of finding the most parallel move code by taking into account the available resources. This encoding further allows to add additional nested constraints on the resources. This solution of finding optimal move code for a single basic block had been extended to a full compiler from synchronous programs to optimal move code for SCAD in [DaSc20]. Those two results are shown in this section.

The following overall pipeline for a compiler from synchronous program to optimal SCAD move code had been chosen:

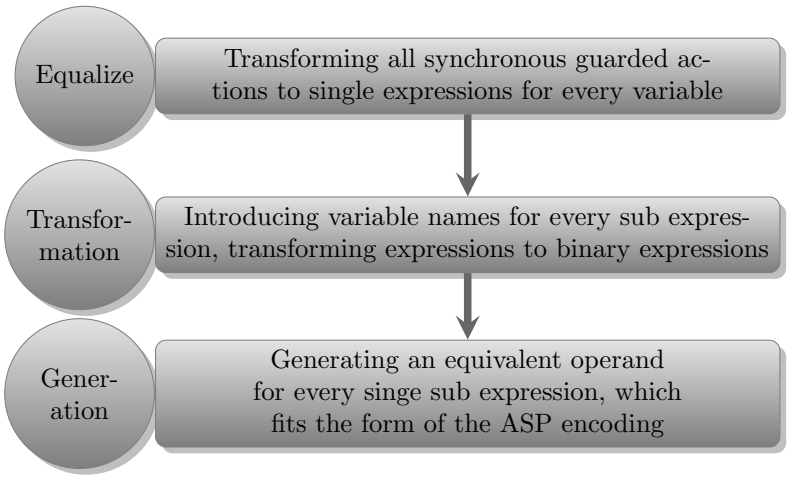


The first step *qrz2aif* is to take a synchronous program in Quartz and compile it to the intermediate language of synchronous guarded actions. As it is known that for constructive and causally correct Quartz programs, a fixpoint

in the sense of Fitting [Fitt85] exists, the *reaction* of the complete program/evaluation of all guarded actions can be computed completely without case distinctions or non-interpreted values. This fact leads to the concrete idea to interpret all synchronous guarded actions together as a single basic block, which describes all possible reactions of the corresponding program in all program states. Such a basic block can also be seen as the complete hardware synthesis of the program and can be used as an input for the ASP encoding described in [DaBS18b]. This translation from the synchronous guarded actions to the corresponding ASP encoding is substituted under *aif2lp*. The task of *aif2lp* especially is to generate 2-address code out of all synchronous guarded actions. This then allows a uniform input for the last step *lp2mc*, which finally finds optimal move code for a given program. Further, *lp2mc* includes the transformation from the ASP answer set to the move code syntax of the SCAD machine.

4.2.1. aif2lp

The task of the first step *aif2lp* is to transform synchronous guarded actions to a form interpretable as a single basic block which can be used as an input for the next step *lp2mc* for the optimal move code generation. It is proposed to execute this task with the following three steps, which are described in detail within the next paragraphs:



Equalize On the one hand, the guard γ of a guarded action $\gamma \Rightarrow \alpha$ represents a *label* which stands for the current control flow position of the program.

$$label1 \rightarrow y = z \& x$$
$$label2 \rightarrow y = z | x_2$$

On the other hand, the actions α of the guarded actions assign values to variables if the label is true in the current environment. As there can be multiple assigning actions α for the same variable x_i under different program conditions, this format cannot directly be used to generate a directed acyclic graph;

which is needed to be interpreted as a basic block. For the example above this would mean that one could generate two operands similar to $operand(y, z, x)$ or $operand(y, z, x_2)$ for the same variable y, which does not represent a DAG anymore. Such an operand form should be read as y can be generated by using z and x or z and x_2. In order to reach a form which allows the interpretation as a DAG, it is required to get a unique single expression similar to $operand(y, A, B)$ for every variable x_i. In Averest, such a form can be archived by the *equalize* operation, which transforms a program given as a set of synchronous guarded actions to an equivalent set of guarded actions, in which every variable gets a unique single action consisting of a case expression on the right hand side of the assignment.

The two expressions from the example above will be transformed to the following single guarded action containing only one case statement:

```
y =
  case
    label1: z & x;
    label2: x | x2;
  default false
```

In the background chapter, the example program ABRO has been introduced in Figure 2.1. The program describes a system which waits for two signals `a` and `b` to appear on the sensors and trigger the output `o` as soon as both the input sensors fired. The procedure can be reset at any time with a reset signal `r`.

This ABRO example compiles to the synchronous guarded actions as shown in Figure 2.2. Hereby, the different program states are encoded in the labels wa, wb, wr and in some further conditions on the input variables. As mentioned before, this representation itself is not directly suited to be transformed into a DAG, as e.g. different assignments to *wa* occur in different guarded actions.

Therefore, the ABRO example can be transformed by the equalize transformation in a corresponding system containing only case statements as unique assignments for every variable. This result is shown in Figure 4.3. Note that here, the reaction to absence was made explicit by using the 'if and only if' semantics of equations.

Transformation The *equalize* transformation produces a set of equations, in which every right hand side of every assignment is a case statement. The translation of those case statements to a form which can be used for the shown *operand*s requires every case statement to be rewritten and transformed like shown in the following. Especially, this means that every sub-expression must be transformed to a binary expression in a way that a uniform representation is achieved, which can directly be read as a directed acyclic graph.

Consider that the expression only consists of Boolean expressions and variables. Then, the transformation to binary operations / 2-address-code can be archived as follows. Hereby, the following property of the given equalize

```
control flow:
  next(running) = true
  next(wa) = case
               !running: true;
               !r&wa&!a|r&(wr|wa|wb): true;
             default: false;
  next(wb) = case
               !running: true;
               !r&wb&!b|r&(wr|wa|wb): true;
             default: false;
  next(wr) = case
               !r&(wr|a&wa&b&wb|!wa&b&wb|!wb&a&wa): true;
            default: false;
data flow:
    o = lvar018 //abbreviation variable computed by the compiler
```

Figure 4.3.: *Equation system for ABRO*

algorithms has been used: the produced default reactions/default values are always the Boolean constant **false** and all other cases in the expression assign only the constant value **true** to the variable if the corresponding case condition holds. With those facts in mind, the above case statement from the example was originally given as:

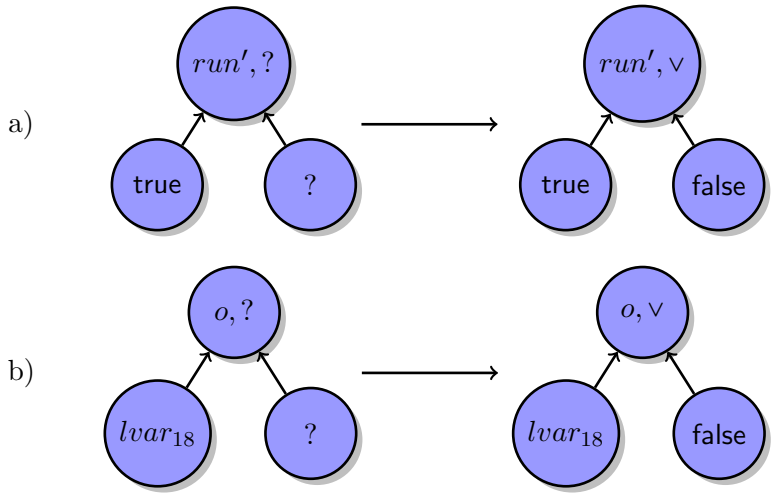
```
y = case
      label1&(z&x) : true
      label2&(z|x2) : true
    default: false
```

For a full compiler, let's take all available guarded actions of the dataflow, control flow and the abbreviations together. For those guarded actions, the goal is to generate an acyclic dataflow graph including all the information from the guarded actions themselves. Additionally, all nodes in the graph will be labeled so that every sub-expression can be given a concrete name. This enables a direct generation of the required operand structure from this graph.

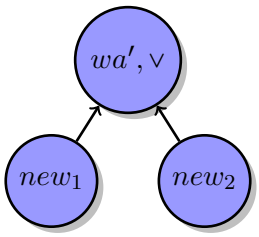
When taking into account Boolean case statements, the idea is to convert them into equivalent disjunctions: Every statement **case** $\alpha_1 : \beta_1; \ldots; \alpha_n : \beta_n;$**default**: ω; can be converted to the disjunction $\bigvee_{i=1}^{n} \alpha_i \wedge \beta_i$ (as already mentioned, remember that ω is false for Boolean event types).

By looking into the details, some corner cases have to be considered. Recap again our example in Figure 4.3: It consists of four case statements for the control flow and one case statement for the data flow.

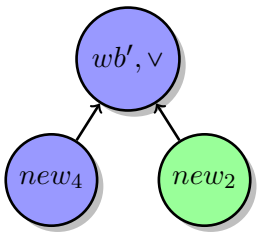
If e.g. no case statement is given at all, as in the example a) for the variable `running` in the control flow and b) for the data flow variable `o`, a new binary expression can be introduced for its right-hand side φ. To this end, the variable φ is replaced by the equivalent binary expression $\varphi \vee$ **false**:



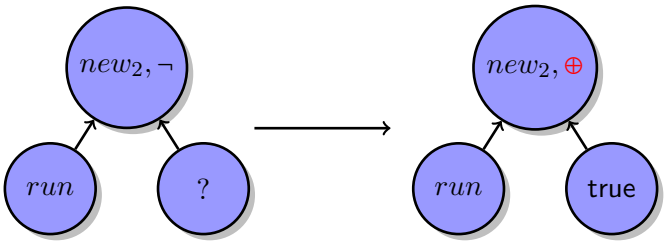
To be able to translate nested expressions into a DAG, new variables have to be introduced for non-binary sub-expressions to build a binary tree. In the ABRO example this is e.g. needed for all first case conditions in the control flow. Therefore, to express for example **`next(wa)`** (shortly written as `wa'`) as a binary expression, all case conditions of `wa` must introduce new variables.



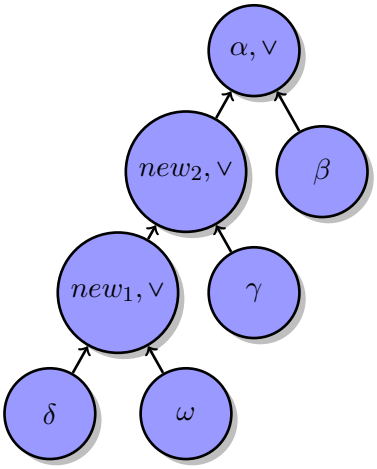
It is important to note that for duplicate expressions no new variable names should be introduced - as otherwise the same expression would evaluate twice on the running processor. In the ABRO example, the sub-expression `!running` had been given the name $\texttt{new}_2$ in the case statement of `wa'`. Therefore, this new variable name should be reused in the DAG representation of `wb'`:



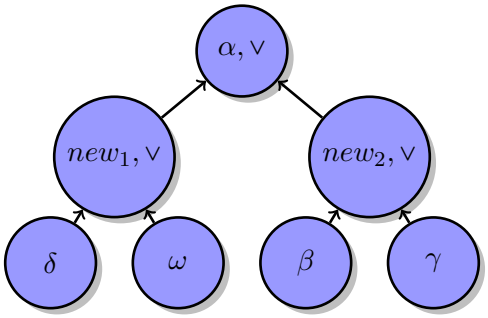
Looking more carefully at the exact definition of the introduced variable $\texttt{new}_2$, it can be realized that this expression initially was a negation and therefore also was not directly binary. To work around this, negations can be transformed into an appropriate expression as into a xor expression with the constant `true`. In our ABRO example, this could lead to a node for the variable $\texttt{new}_2$ as follows:



Another transformation has to be performed. Consider that the list of case conditions contains more than two case statements, then additional variables have to be introduced to allow building a binary tree. Hereby, the order of the construction defines how many steps need to be executed to evaluate the complete case statement. If we take the case statement α = **case** β : **true**; γ : **true**; δ : **true**; ω : **true**; **default** : **false**, the simplest way would be to create a binary tree by recursively folding the list of case conditions into own case statements; e.g. with a disjunction of the last two cases. This would lead to the following binary tree:



Such an unbalanced binary tree is of course not optimal and introduces additional computation steps. This comes from the fact that new data dependencies are introduced, which are not needed for transforming the case statement to a DAG. Instead, the proposed compiler should build a balanced tree if it translates a list of case conditions as disjunctions recursively.



If everything is combined, an expression dataflow graph as DAG, which defines all variable computation dependencies, is constructed. Furthermore, every node in this DAG is labeled with a unique variable name and all sub-expressions appearing on different program locations only appear as one node in the resulting graph.

Code Generation The task now is to generate a unique representation of the so computed binary expression tree with the ASP syntax. As such a representation, a set of *operands(X,Y,Z)* is chosen for every variable X, which state the data dependencies of X by indicating that X is computed by Y and by Z. The concrete operation out of the expression node of X can be omitted, as it is assumed to have only universal processing units in the SCAD machine, which means that the operation does not influence the compilation result regarding the optimality. Still, in order to compute the concrete move code, the developed compiler prototype stores the operation together with the operands as comment besides the according ASP rule.

For the discussed example program ABRO, this compilation technique results in the following ASP operands as shown in Figure 4.4.

```
% Abbreviations
   operand("asp_lvar_0", "a", "TRUE"). % xor
   operand("lvar_000", "wa", "asp_lvar_0"). % and
   operand("asp_lvar_1", "r", "TRUE"). % xor
   operand("lvar001", "lvar000", "asp_lvar_1"). % and
   operand("lvar002", "lvar001", "asp_lvar_1"). % and
   operand("lvar003", "a", "wa"). % and
   operand("asp_lvar_2", "b", "TRUE"). % xor
   operand("lvar004", "wb", "asp_lvar_2"). % and
   ...
   operand("lvar009", "lvar003", "lvar007"). % and
   operand("asp_lvar_3", "wa", "TRUE"). % xor
   operand("lvar010", "lvar007", "asp_lvar_3"). % and
   operand("asp_lvar_4", "wb", "TRUE"). % xor
   operand("lvar011", "lvar003", "asp_lvar_4"). % and
   ...
% Control Flow
   operand("asp_lvar_5", "lvar002", "lvar023"). % or
   operand("asp_lvar_6", "running003", "TRUE"). % xor
   operand("next_wa", "asp_lvar_5", "asp_lvar_6"). % or
   operand("asp_lvar_7", "lvar016", "lvar019"). % or
   operand("next_wr", "asp_lvar_7", "FALSE"). % or
   operand("asp_lvar_8", "lvar006", "lvar023"). % or
   operand("next_wb", "asp_lvar_8", "asp_lvar_6"). % or
   operand("next_running003", "TRUE", "TRUE"). % or
% Data Flow
   operand("o", "lvar018", "FALSE"). % or
```

Figure 4.4.: *ASP encoding of the ABRO example.*

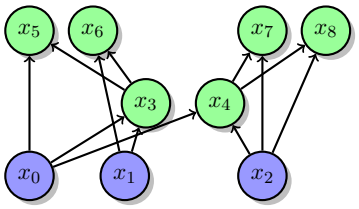
This representation of a program as *operand*s is used as the inputs for the next step *lp2mc*, which is the actual definition of the search problem with ASP to find the optimal move code.

4.2.2. lp2mc

To define the actual search problem of the optimal code is part of the prototypical implementation in *lp2mc*. The results shown in this section are based on the publication in [DaBS18b].

Default Problem Encoding As a first step, the scheduling of basic blocks itself is modeled similar to the encoding in [BhSc16], which was used as a base encoding idea also for the shown ASP encoding. In this first step, the question is to be answered about how many numbers of PUs are required to schedule a given basic block B on a SCAD machine without the introduction of the overhead operations duplicate and swap. Additionally, with the given ASP encoding, the amount of such minimal schedules can be counted and every schedule's structure can be analyzed further as *all* optimal schedules can be computed.

The outcome of the last step *aif2lp* was a basic block for a complete synchronous program. Let's consider the following example of a basic block as shown in Figure 4.5.



$operand(x_3, x_0, x_1).$
$operand(x_4, x_0, x_2).$
$operand(x_5, x_0, x_3).$
$operand(x_6, x_1, x_3).$
$operand(x_7, x_4, x_2).$
$operand(x_8, x_4, x_2).$

Figure 4.5.: *(a) An example basic block as DAG, (b) the same DAG as ASP-code*

From such basic blocks it is desired to produce schedules for a SCAD machine. Such a schedule consists of two different aspects to be answered:

- *Assignment:* which variable is produced by which PU?
- *Ordering:* in which order are the variables produced, when they are assigned to the same PU?

An assignment is the mapping of variables to PUs and an ordering on such a PU must enable the generation of move code for a SCAD machine without producing overhead operations. To be more concrete, such an assignment and ordering must allow to move values from the (FIFO) output buffers to (FIFO) input buffers while retaining the variable order of the target PU. This enables the SCAD machine to produce the values in the needed order.

For the shown example in Figure 4.5, some valid schedules of the basic block have been collected in Figure 4.6. Here it is to mention that during actual execution more than one copy of a variable can be produced in the output buffer if it is needed by the given basic block, although only one copy of each variable is shown in the figure. For example, the following is a valid sequence of

move instructions for Solution 1. It allows the execution of the example basic block on the SCAD machine while it respects the order of variables (every PU_n has two input buffers, the left one named $PU_{n,L}$ and the right one $PU_{n,R}$): $x_0 \rightarrow PU_{0,L}$, $x_0 \rightarrow PU_{0,L}$, $x_0 \rightarrow PU_{1,L}$, $x_1 \rightarrow PU_{0,L}$, $x_1 \rightarrow PU_{1,R}$, $x_2 \rightarrow PU_{1,R}$, $x_2 \rightarrow PU_{1,R}$, $x_2 \rightarrow PU_{0,R}$, $x_3 \rightarrow PU_{0,R}$, $x_3 \rightarrow PU_{0,R}$, $x_4 \rightarrow PU_{1,L}$, $x_4 \rightarrow PU_{1,L}$.

In the same matter it is possible to generate valid move code for all other solutions. The input variables of the basic block, e.g. the ones without incoming edges in the original DAG, are considered to be loaded from the main memory of the SCAD machine (in the example the variables x_0, x_1 and x_2) and are considered not to be computed by any PU.

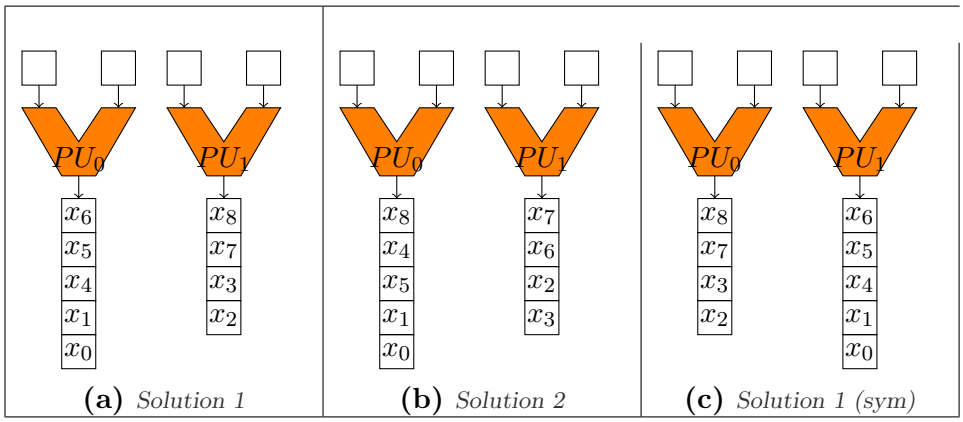


Figure 4.6.: *Three valid variable assignments and orderings that allow computation of the basic block in Figure 4.5 without any overhead*

The next steps will show how such solutions can be found with the help of an according ASP encoding.

First of all, all variables out of the given input DAG (as *operands*) are extracted, so that they can be used throughout the complete ASP problem definition:

$$\begin{aligned} var(X) &:- operand(X,Y,Z). \\ var(Y) &:- operand(X,Y,Z). \\ var(Z) &:- operand(X,Y,Z). \end{aligned}$$

The so-called *generate part* of the ASP problem definition (the part which defines the general solution domain) is shown next. For the optimal move code generation, it is defined that every variable must be mapped to exactly one PU which produces its value. Further, if any two variables are assigned to the same PU, they must be ordered in their production order (either $V_1 < V_2$ or $V_2 < V_1$). To find the corresponding order and assignments is the task of the ASP solver.

$$\begin{aligned} 1\ \{asgn(VAR,PU) : pu(PU)\}\ 1 &:- var(VAR). \\ order(V_1,\ V_2),\ order(V_2,V_1) &:- asgn(V_1,\ PU),\ asgn(V_2,\ PU), \\ & \quad V_1 \neq V_2. \end{aligned}$$

Next, the so-called *test part* takes place, which has the subject to restrict the solutions of the defined domain to valid solutions regarding the problem

definition. In case for the shown optimal move code search problem, the test part adds restrictions to the production order of variables, such that the computations of the basic block can be performed and there is no contradiction in the data dependency order. [BhSc16] and [BhSc17] showed that it is sufficient to test that the order of two variables V_1, V_2 produced by a processing unit PU_1 is also given and preserved in their corresponding operands $V_{1,L/R}$ and $V_{2,L/R}$ if both those operands are produced by a same processing unit PU_2.

$$
\begin{aligned}
order(V_{1,L},\ V_{2,L}) \ :- \ & asgn(V_1,\ PU),\ asgn(V_2,\ PU),\ V_1 \neq V_2,\\
& order(V_1,\ V_2),\ operand(V_1,\ V_{1,L},\ V_{1,R}),\\
& operand(V_2,\ V_{2,L},\ V_{2,R}),\ asgn(V_{1,L},\ PU_2),\\
& asgn(V_{2,L},\ PU_2),\ V_{1,L} \neq V_{2,L}.\\
order(V_{1,R},\ V_{2,R}) \ :- \ & asgn(V_1,\ PU),\ asgn(V_2,\ PU),\ V_1 \neq V_2,\\
& order(V_1,\ V_2),\ operand(V_1,\ V_{1,L},\ V_{1,R}),\\
& operand(V_2,\ V_{2,L},\ V_{2,R}),\ asgn(V_{1,R},\ PU_2),\\
& asgn(V_{2,R},\ PU_2),\ V_{1,R} \neq V_{2,R}.
\end{aligned}
$$

Further to that, the production order to be found must additionally preserve the data dependencies defined in the basic block definition. If, for example, an operand $V_{1.L}$ is produced by the same PU as its consumer V_1, it must be given that then also this order from the initial DAG must be preserved in the production order. This fact must not only be valid to the direct operands, but to all predecessors of a variable V_1. To this end, first a construction of the DAG and based on that definition, the definition of an according predicate *predecessor(X,Y)* is added to the ASP encoding. This predicate is `true` if there exists a path in the DAG between the two given nodes X and Y:

$$
\begin{aligned}
node(X) \ :- \ & var(X).\\
edge\_initial(X,Y) \ :- \ & operand(Y,X,\_).\\
edge\_initial(X,Y) \ :- \ & operand(Y,\_,X).\\
rootNode(X) \ :- \ & not\ edge\_initial(\_,X),\ node(X).\\
\\
predecessor(X,Y) \ :- \ & edge\_initial(X,Y).\\
predecessor(X,Z) \ :- \ & predecessor(X,Y),\ predecessor(Y,Z).
\end{aligned}
$$

This predicate can then be used to restrict the production order of every PU to only allow orders which do correspond with the predecessors order.

$$:- \ predecessor(X,Y),\ asgn(X,\ PU),\ asgn(Y,\ PU),\ order(Y,X).$$

Finally, the transitive closure of the ordering relation is defined, so that it is possible to compare every position of every variable on the same PU.

$$order(V_1,\ V_3) \ :- \ order(V_1,V_2),\ order(V_2,\ V_3).$$

In the end, the first overall problem definition, which is to minimize the amount of PUs needed to allow a definition of such a production order, can be formulated as follows:

$$
\begin{aligned}
&pu(0..PUS-1) \ :- \ amountPUs(PUS).\\
&possiblePUAmount(0..max\_pus).\\
&1 \ \{amountPUs(N) : \ possiblePUAmount(N)\} \ 1.\\
&\#minimize\{N : amountPUs(N)\}.
\end{aligned}
$$

Executing this problem definition with the above example leads to 8800 different solutions, whereas three of them have been picked out and have been shown in Figure 4.6. When taking a closer look, it can be seen that the example solution (c) is a symmetric solution of (a). This computation overhead can be removed by adding symmetry breaking constraints, which in this case sort the PUs by the minimally assigned variable.

$$
\begin{aligned}
minimum(PU, S) \ :- \ & S = \#min\{VAR : asgn(VAR, PU)\},\\
& pu(PU), asgn(\_, PU).\\
:- \ & minimum(PU, S), minimum(PU_2, S_2),\\
& PU_2 > PU, S > S_2.
\end{aligned}
$$

This symmetry breaking constraint deletes all rotated results and the according assignments from the resulting answer sets. Overall, $\frac{N}{amountPU!}$ many solutions are left when this symmetry breaking constraint is added and when there would normally have been N schedules for $amountPU$ many PUs. As an example, if there are $N = 2016$ schedules for scheduling a basic block on 4 PUs, $\frac{2016}{24} = 84$ unique solutions are left when adding this symmetry break. For the example Figure 4.5, the amount of results reduces to $\frac{8800}{2!}$ schedules: in total from 8800 to 4400.

Optimal Scheduling Regarding Execution Time So far, a basic scheduling optimization problem has been defined, which allows to find the minimum required PUs to run a given program or basic block on a SCAD machine. It was observed, that this basic optimization problem can be enhanced to more sophisticated minimization modes with small changes, such as finding a schedule with a minimal execution time on SCAD. In contrast to the already available time optimization encoding in SMT [BhSc17], where an assignment of variables to timeslots was used, in ASP a very direct way to define the time minimization problem was found. This idea scales very well, especially in contrast to the SMT encoding, which added huge amounts of constraints the more variables a program had.

This idea is based on the observation that the found scheduling in the previous section basically introduces additional edges to the initial DAG describing the basic block. All variables which are now produced by a same PU in a fixed order can add additional edges to the initial DAG for the new order constraints between some of those variables. This means that all answer sets of the previous sections describe all possibilities to add such new edges to the DAG with the only constraint to just allow minimal possible paths through

the graph, which stands for the minimal amount of PUs searched. Further, the constraints prohibit the introduction of cycles by these new edges.

When talking about the execution time from an abstract perspective, the longest path through that combined graph describes how many steps are needed to compute the result for the basic blocks, as it contains all data dependencies and therefore all computations needed to find the solution for the given basic block. By adding different costs for every OP-code executed, for every PU or even adding costs for moving values, this abstract model can easily be enhanced to a particular real hardware implementation prediction.

In the following, this idea is illustrated with the help of our example solutions from Figure 4.6. Here, solution a) introduces the additional edges $x_0 \to x_1$, $x_1 \to x_4$, $x_4 \to x_5$, and $x_5 \to x_6$ by the concrete found order of PU_0 and $x_2 \to x_3$, $x_3 \to x_7$, and $x_7 \to x_8$ by the order of PU_1. These edges can be added to the initial DAG and when the longest path is searched in this combined DAG, the execution time/amount of execution steps is found; e.g. solution 1 needs 5 steps to execute the basic block on a SCAD machine.

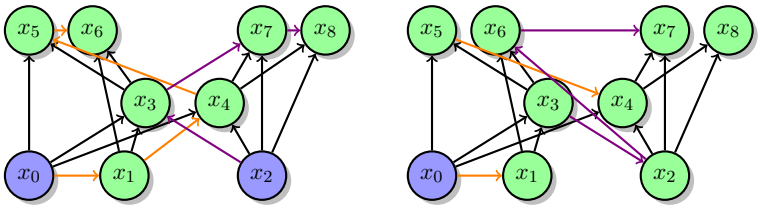


Figure 4.7.: *Combined DAG for Solutions 1 and 2 from Figure 4.6(a) and (b)*

However, if the combined DAG for solution 2 is constructed and examined, it can be seen that there exists at least one path with cost 6. Although solution 1 and 2 can both be scheduled on 2 PUs, solution 1 would be a better choice for an optimal compiler as it requires less concrete execution steps.

To model this additional minimization problem with ASP, one can first construct the combined DAG with the help of the newly introduced *order* relation of the concrete answer set:

$$\begin{aligned} edge(X,Y) &:- edge\_initial(X,Y). \\ edge(X,Y) &:- order(X,Y). \\ initialNode(X) &:- not\ edge(\_,X),\ node(X). \end{aligned}$$

Having available the combined DAG with all variable construction dependencies in ASP, the longest path and therefore the needed execution time can be found.

$$\begin{aligned} pathCosts(X,1) &:- initialNode(X). \\ pathCosts(Y,N+1) &:- edge(X,Y),\ pathCosts(X,N),\ N < (M+1), \\ &\quad\ amountVars(M). \end{aligned}$$

$$maximalCost(N) :- N = \#max\{C : pathCosts(\_,C)\}.$$

With the help of the preducate *maximalCost*, which contains the longest path through the combined graph, the execution time minimization problem can be directly expressed with the help of ASP:

$$\#minimize\{N@1 : maximalCost(N), N > \#inf\}.$$
$$:\!- \ maximalCost(\#inf).$$

It allows to restrict all solutions to those which only have the most minimal longest path in the combined DAG throughout all answer sets and only contain the concrete production orders, which lead to the best execution time regarding the found minimal amount of PUs when executed on a SCAD machine. (Hint: the infimum expression $\#inf$ in the ASP encoding ensures that a maximal cost must be found/defined).

For our example Figure 4.5, it was figured out that only 520 of all 8800 possible schedules on 2 PUs are optimal (260 with symmetry breaking). Especially the solutions (a) and (c) of Figure 4.6 are minimal with a cost of 5, whereas solution (b) costs 6 execution steps and is therefore excluded by the newly introduced optimality constraints.

So far, the minimal amount of PUs has not been touched, which raises the question, if schedules exist when having available more processing units for the computation, but which are faster and require less execution steps on a SCAD machine. And what is the overall fastest execution possible if no limitation on the available PUs are given? The latter question can be answered quickly, as the depth of the initial DAG represents the overall lower bound to the execution time. But with which minimal amount of PUs can this execution time be reached? This question can be answered in the shown ASP encoding by just switching the priorities of the two minimization statements accordingly: First compute all minimal schedules regarding execution time and then minimize the amount of PUs needed to reach this execution time.

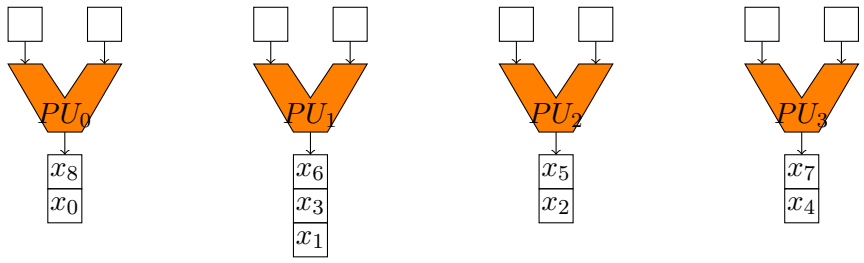


Figure 4.8.: *One example to minimize the amount of execution time and then find the minimal number of PUs needed to schedule the example basic block shown in Figure 4.5*

Figure 4.8 shows such one out of all 6912 possible minimal schedules (288 with symmetry breaking) for our example DAG Figure 4.5. The initial basic block can be scheduled on 4 PUs if it is required to minimize the overall execution time, which is, in this case, 3 steps (as this was the maximal depth in the initial DAG).

Enhanced Optimal Scheduling So far, the encoding finds schedules with an overall minimal execution time or it finds a minimal schedule on the overall least possible amount of PUs. It is absolutely more realistic to consider both resource and time constraints together instead of optimizing towards the direction of one of those two. Therefore, it may be more realistic to determine the minimum number of PUs to schedule a basic block within N steps or to find the fastest execution if a limitation for the PUs is given.

Also these two questions can be answered with small adaptions to the ASP encoding, which again shows the flexibility when programming in ASP: As a predicate holding the maximal cost of a graph *maximalCost* is already available in the ASP prorgram, the first question can be answered by introducing one additional constraint which states that the cost does not exceed a given constant for maximal executions.

$$:- \quad maximalCost(N),\ N > max\_execution.$$

For our example basic block Figure 4.5, Figure 4.9 contains one minimal solution of 3 PUs needed at minimum when an upper bound for the execution time of 4 is defined.

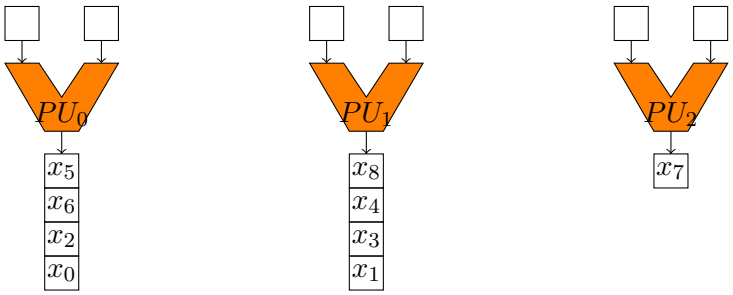


Figure 4.9.: *One example out of 8088 (without symmetries: 1348) to minimize the amount of PUs with an upper bound to 4 execution steps from the example basic block shown in Figure 4.5*

Similar to this upper bound on the execution time, the resource limitation regarding the amount of available PUs can be defined:

$$:- \quad amountPUs(N),\ N > pus\_available.$$

As a last enhancement for the ASP encoding, one concrete detail can be introduced for a realistic move code generation: As already mentioned earlier, the input variables of a complete basic block do not just appear in the according output buffers in a schedule, but they must be read from the Load-Store-Unit (LSU). This additional constraint has been included into the encoding by taking all root nodes of the initial DAG (the input variables) and by forcing them to be always assigned to the first processing unit PU_0. All other variables than input variables are not allowed to be produced by PU_0.

$$\begin{aligned} asgn(X,0) \quad &:- \quad rootNode(X). \\ &:- \quad not\ rootNode(X),\ assignment(X,0). \end{aligned}$$

This forced assignment of input variables to PU_0 (representing the LSU) leads to an exception in the symmetry breaking constraint. Only the processing units PU_L with $L > 0$ must be sorted regarding the variable names and must be included in the cyclic dependency check.

$$: -minimum(PU, S), minimum(PU_2, S_2), PU_2 > PU, S > S_2, PU > 0.$$

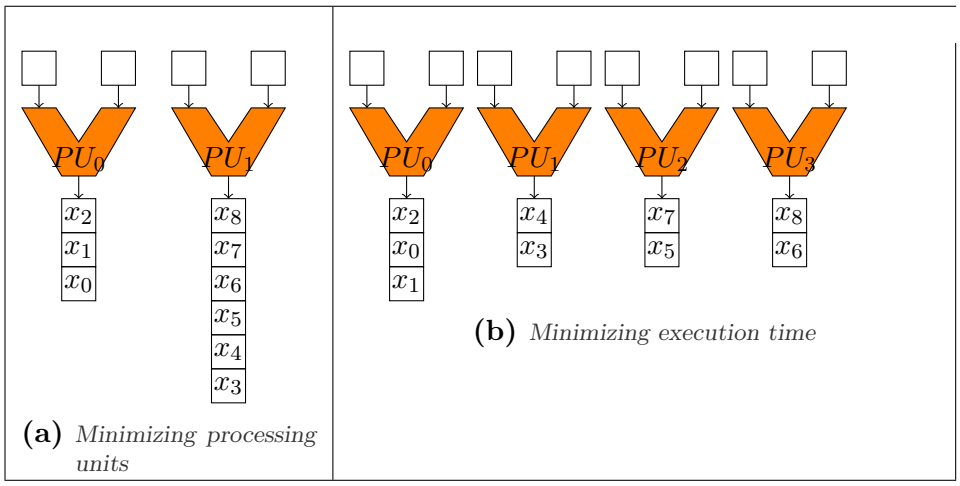


Figure 4.10.: *Example schedules with LSU constraints for Figure 4.5*

Forcing all input variables to come from one PU results for the example in 22 different unique solutions (same if removing symmetries) when minimizing the amount of PUs (see Figure 4.10). The maximal execution time is, in this case, 8. By minimizing the overall execution time with LSU, ASP produced results with execution time 5 and needed 4 PUs to schedule it (696 optimal schedulings, 116 without symmetries).

4.2.3. Experimental Results

In the last sections, the steps needed to setup a full optimal compiler pipeline from synchronous programs/ Quartz programs to SCAD move code have been shown. As the heart of this compiler, the underlying complex minimization problem has been defined with the help of stable models/ASP. All those tools have been developed as prototypes. This allowed to set up benchmarks and evaluate the feasibility of the shown approach.

The first subsection will focus on the heart of the compiler pipeline *lp2mc*, with which the ASP problem encoding could be fairly compared with corresponding SAT and SMT encodings. The second subsection will show some benchmarks for the complete compiler chain, by taking different Quartz programs and completely compiling them to runnable move code for a SCAD machine.

lp2mc: Comparison with SAT and SMT/Benchmarks The shown encoding *lp2mc* to find the optimal SCAD move code for a basic block has been

benchmarked on an Intel Core-i5 (4 x 2.67 GHz) desktop computer with 8 GB RAM running Ubuntu 14.04. To this end, a random basic block generator had been used instead of using real synchronous programs, as this allows the comparison with the already available encodings with SAT [BhSc16] and SMT [BhSc17]. The random basic block generator was able to produce DAG representations of a given size n and a given level l allowing a good control of the comparison parameters [1]. The benchmark setup generated 1000 basic blocks for every pair (node,level). For a fair comparison (1) with the SAT encoding [BhSc16], the corresponding ASP program with the only optimization criteria to find the minimal amount of PUs required to execute the basic block without overhead on a SCAD machine has been used. For the comparison (2) with the SMT encoding [BhSc17], the program setup to minimize the number of PUs and after that to minimize the time needed to execute the basic block on a SCAD machine without overhead was defined. As tools in the experiments. the ASP framework clingo 5.2.2, the Microsoft's SMT solver Z3, and an own SAT solver implemented in the Embedded Systems Group in Kaiserslautern were installed.

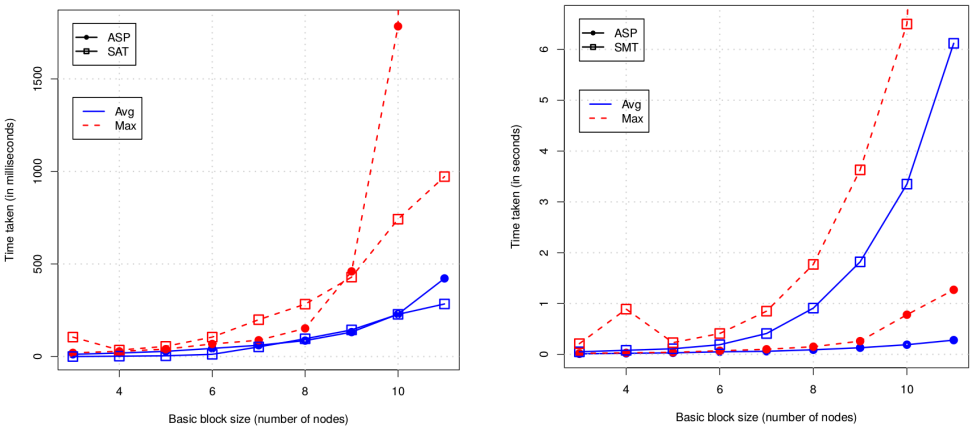


Figure 4.11.: *(a) Average and maximum time required by SAT and ASP solvers to derive resource constrained schedules for programs of different sizes. (b) Average and maximum time taken by SMT and ASP solvers to derive time constrained schedules for programs of different sizes.*

Figure 4.11 shows the benchmark results by giving the execution times of Clingo in comparison to the SAT (a) and SMT (b) encoding.

In comparison to SAT (a), no significant differences on the execution times to find the overall minimal amount of PUs to run the basic block could be observed, although the ASP encoding produced *all* possible answer sets with Clingo, whereas the SAT encoding only produced one satisfying model. However, there is a small trend that the execution times of Clingo will increase faster than in the corresponding SAT encoding.

On the other hand, the ASP encoding outperformed the SMT encoding by far. It is quite interesting to mention that the CPU times of Clingo to find the

[1] The tools are available at https://es.cs.uni-kl.de/tools/teaching/

minimal execution times on a minimal amount of PUs (b) is even faster than the version in (a), where only the amount of PUs are minimized. With the ASP encoding to find minimal execution times, there could easily be computed schedules up to 20 nodes without running into memory, time or other resource problems, which was not possible with the SMT encoding.

Experimental Results for the Full Compiler As already mentioned, a complete prototype of all compiler steps has been implemented. According to the proposed compiler chain, it first converts a synchronous program into a representation as a basic block/DAG, generates an ASP input for that DAG and uses the shown ASP encoding to find the minimal/optimal SCAD move code. Finally, it converts the resulting answer set into the syntax of the SCAD machine, such that the resulting code can be directly pasted to the available SCAD simulator available on https://es.cs.uni-kl.de/tools/teaching/.

In order to benchmark this compiler chain, various examples out of the Averest[2] benchmark suite have been used for the experiments. All results have been checked with the SCAD simulator. The experiment was performed on a Windows10 machine with an Intel i5-6600k processor (4 x 3,5GHz) and 16GB of installed RAM.

The benchmark results are shown in Table 4.3. It can be observed that the compiler chain is quite fast for most of the shown small examples and compiles the optimal move code mostly under 1 second. Only a few more complex examples containing reincarnations and circles, as the program Edwards02 and DiningPhilosophers, the ASP solver requires more runtime to find the optimal move code.

All in all, the results show that the stable model semantics and its implementation framework ASP are quite powerful and can even solve complex tasks like finding optimal code compilation results in a feasible manner. Of course this prototypical compiler is neither complete nor optimized with all available knowledge from modern compiler constructions, but it is a good example to show the power of stable model semantics and answer set solving.

4.3. Enhancing Synchronous Programs by Stable Models

So far, the focus was set on the translation of the synchronous language syntax to the ASP syntax in Chapter 3 and two examples for the use of ASP in Section 4.1 and Section 4.2, which showed the power of the stable model semantics. With the semantics extensions shown in Chapter 3, available synchronous programs can be translated to corresponding ASP programs. Nevertheless, the ASP-Core-2 language has some enhanced language constructs, which have not been considered so far by the translation scheme, but that are used quite often when defining problems in ASP. Most of those extensions appeared in the examples in Section 4.1 and Section 4.2. When taking a look

[2] `http://www.averest.org`

Experimental Compiler Results		
Program Name	Compilation Time	#PUs
ABRO	1.406s	3
Berry P01	0.032s	3
Berry P02	0.062s	4
Berry P03	0.016s	4
Berry P04	0s	3
Berry P05	0.048s	4
Berry P06	0.016s	5
Berry P07	0.048s	4
Berry P09	0.048s	5
Berry P10	0s	2
Berry P11	0.157s	7
Berry P12	0.016s	5
Berry P13	0.36s	7
Berry P14	0.094s	6
Berry P15	0.048s	4
Berry P16	0s	2
Berry P17	0.08s	7
Berry P18	0.096s	7
Berry P19	0.064s	6
Berry P20	0.204s	9
Berry P21	0.11s	7
Edwards02	416.172s	4
Fork	0.078s	3
Dining Philosophers (Simple)	5.172s	3
Redo Abort	0.032s	3
Trap vs Abort	0.048s	4
Trap vs Immediate Abort	0.063s	5
Trap vs Weak Abort	0.016s	2
Trap vs Weak Immediate Abort	0.031s	3
Schizophrenia 10	0.094s	3
Schizophrenia 11	0.109s	3

Table 4.3.: *Experimental results*

from the opposite direction, meaning to see synchronous languages like Quartz as a full front-end language for ASP, those constructs have to be contained in the translation scheme as well in order to allow users to access the full capability of modern ASP solvers from a higher language like the synchronous language Quartz. This section sketches an approach to introduce those enhanced language constructs into the synchronous language core of Quartz.

4.3.1. Disjunctive Programming

First of all, ASP allows disjunctive rule definitions meaning that the rule head can be connected disjunctively:

$$x_1|\ldots|x_n \quad :- \quad expr_1, \ldots, expr_m.$$

Such disjunctive definitions especially allow to shift the perspective away from concrete algorithm definitions to descriptive constraint programs, which describe how the solution looks like instead of defining an algorithm on how to compute the solution. Here, not the programmer or grounder defines the solution, but the solver assigns the values of the variables according to all other constraints and rules of the program.

An extension of synchronous languages could add such disjunctive descriptions syntactically by allowing the left-hand-side of the expressions to contain multiple variable names:

`x`$_1|\ldots|$`x`$_n$ `= <typed>Expr;`

In order to stay type-safe, all variables `x`$_i$ must be of the same type, and the concrete expression to be assigned must be of that type as well.

The semantics of such disjunctive expression heads are given by the according ASP rule. This means that the ASP solver decides due to all other constraints, which variable `x`$_1|\ldots|$`x`$_n$ gets assigned the according translation of the expression `<typed>Expr`.

As an example, take a quartz program which assigns an input natural number value either to a variable `odd` or `even`.

```
module Disjunction(event nat ?in) {
   event nat odd;
   event nat even;

   odd | even = in;
   ...
}
```

4.3.2. Constraints

Without further constraints, the module `Disjunction` from above would always lead to two answer sets `odd=in` or `even=in`. When describing results in a descriptive way, it is often needed to shrink down the search space for the resulting solutions by defining concrete constraints deciding which states and

variable assignments should never be reached. In ASP, such expressions do not have a rule head, e.g. `:- exp`$_1$`;...;exp`$_n$. and state that the given expressions can never appear together.

In order to include such constraint expressions into synchronous languages, one idea would be to introduce a special variable with a special name like `disallowed` and with the type **`bool`**: `disallow = boolExp`, which can be assigned in every place in the program to express that this statement at that program position is not allowed.

To that end, n different assignments to `disallowed = boolExp`$_i$ are included in the AIF program, which leads, together with the defined translation scheme from synchronous languages to the ASP-Core-2 language in Chapter 3, to several ASP rules with the head `v_disallowed :- v_boolExp`$_i$.

Finally, the ASP statement `:- disallowed.` can then be added to state that the computed solution is only valid if no disallowed boolean expression is contained in the resulting system answer.

For our example module `Disjunction` to separate odd and even numbers, the corresponding extended module `Contraints1` could introduce according constraints on the defined numbers in Quartz, such that the answer set solver only finds one answer set which depends on the input variable `in`: if `in` is odd, it is assigned to the variable `odd` and if `in` is even, it is assigned to the variable `even`.

```
module Constraints1(event nat ?in) {
   event nat odd;
   event nat even;
   event bool disallowed;

   odd | even = in;
   disallowed = odd % 2 == 0;
   disallowed = even % 2 == 1;
}
```

Of course, with the available translation scheme for other types in Quartz programs, this extension can also be used for other types as arrays, like shown in the following module `Constraints2`:

```
module Constraints2() {
   event [10]nat y;
   event [10]nat z;
   event bool disallowed;

   for(i=0..9) {
      z[i] | y[i] = i;
      disallowed = y[i] % 2 == 0;
      disallowed = z[i] % 2 == 1;
   }
}
```

The program describes with the help of constraints the wanted solution for a disjunctive rule, stating that either `z[i]` or `y[i]` is assigned the value `i` from a loop between 0 and 9. It describes via constraints that the array `y` should only contain odd numbers on every index `i` and the array `z` only even numbers.

When translated to ASP, this Quartz program will result in an according ASP program assigning the variable `disallowed` to all conditions described in this way. In the last line of code, it can finally be stated that `disallowed` can not appear in an answer of the computed answer set:

$$
\begin{aligned}
\mathit{disallowed} &:- \quad 1 == z[0]\%2. \\
&\ldots \\
\mathit{disallowed} &:- \quad 1 == z[9]\%2. \\
\mathit{disallowed} &:- \quad 0 == y[0]\%2. \\
&\ldots \\
\mathit{disallowed} &:- \quad 1 == z[9]\%2. \\
\\
&:- \quad \mathit{disallowed}.
\end{aligned}
$$

When now computing the corresponding answer set, the result only contains the correct assignments for the array containing only the odd numbers in the array `y` and the even indexes in the array `z`.

4.3.3. Choices

Another way to define programs more descriptively are the so-called choices. A choice picks a subset of solutions regarding a given constraint. For example, if it is known that there exist exactly `N` solutions for a given problem, one can choose all of them with a choice rule `N {constraint} N`, or if it is known that up to `N` solutions exist, one can state to pick 1 to `N` solutions regarding a given constraint `1 {constraint} N`. This can be especially useful if those rules are mixed with the optimization statements (see below), or if the amount of solutions regarding a constraint are not known beforehand.

According to the ASP-Core-2 language definition, a choice can only appear in rule heads. In our previous ASP examples from the last section, the choice rules were used quite often, e.g. while generating optimal concentrator circuits, the amount of needed switches was defined with a choice rule:

$$
\begin{aligned}
&\mathit{possible\_switch\_amount}(0..\mathit{max_switches}). \\
&1\ \{\mathit{num\_switches}(N) : \mathit{possible\_switch\_amount}(N)\}\ 1. \\
&\mathit{switch}(1..N)\ :- \ \mathit{num\_switches}(N),\ N > 0.
\end{aligned}
$$

A corresponding expression in synchronous languages could be syntactically defined as `s = Choose(nExpMin, nExpMax, boolExp)`, where the first and second natural number expressions stand for the minimum and maximum amount of expressions to be chosen and the `boolExp` for the expression which must be fulfilled by the chosen values `s`. Therefore, the Boolean expression should be defined with `s` as free variable. The example to define the

num_switches to be in a range between 0 and `max` can then be defined with a Quartz module `Choices` as follows:

```
module Choices(event nat ?max) {
   event {}nat num;
   num = Choose(1, 1, num ≤ max ∧ num ≥ 0);
}
```

Hereby, the `Choose` statement chooses exactly one number (both boundaries `nExpMin` and `nExpMax` are 1) between 0 and `max` and assigns this value to the variable `num`. The Boolean expression itself uses this variable `num` as free variable to define which rules must be fulfilled by the chosen variable.

As in general the variable `s` defined with a `Choose` statement stands for a representation of possibly multiple values, it is suggested to introduce a new type `{}<type>` (a set with elements of type `<type>`) into Quartz, so that variables like the chosen `s`, which are not ordered but represent multiple values, can be defined. The variables of type *set* can be used and accessed in the program as all other variables in all expressions.

For example, the definition `s = NatChoose(10,10, s<10);` assigns `s` to stand for all natural numbers up to 10. Furthermore, the expression `s2 = NatChoose(2,3, (s_2 == s)& (s_2 % 2 == 0));` chooses two up to three even values out of these ten numbers and assigns them to s_2. So the variable of s_2 can stand in one solution/answer set for s_2`={2,4}`, in another solution/answer set for s_2`={0,6,8}` and so on. If now the expression `z = s2<s` has to be evaluated, it stands for *is at least one value out of* s_2 *smaller than one value of* `s` and therefore always evaluates to `true`, as `s` can always be chosen as 9 and at least one value picked for s_2 is smaller than 9.

Those two statements for `s` and s_2 evaluate to the following ASP-core-2 rules:

$$asp\_lvar\_0(X) \quad :- \quad X < 10.$$
$$10\ \{s(X) : asp\_lvar\_0(X)\}\ 10.$$

$$asp\_lvar\_1(X) \quad :- \quad s(X); X\backslash 2 = 0.$$
$$2\ \{s\_2(X) : asp\_lvar\_1(X)\}\ 3.$$

But how to handle the operations on variables s_i with the type *`<typed>`set*? All typed operations `s = `$s_i \oplus$ `x` with all operations $\oplus$ and the variables `x` of type `<type>` can be interpreted as `s =`$\forall$`v` $\in$ s_i`.v` $\oplus$ `x` and `s` is then also a variable of type `<typed>set`. The same holds if both operands of the operation are of type `<typed>set`, e.g. `s = `$s_1 \oplus s_2$;. Here, the result is again a set containing the results of applying the operation to all elements in all sets: `s =`$\forall$`v` $\in$ s_1.$\forall$$v_2$ $\in$ s_1`.v` $\oplus$ v_2.

4.3.4. Aggregates

The ASP-core-2 language contains the ability to define so-called *aggregates*, which can be used to reason about the sets of ASP terms and term tuples. The concrete available aggregate functions in the ASP-core-2 language are

$\#count(T)$, $\#sum(T)$, $\#max(T)$, and $\#min(T)$. From the perspective of a higher programming language like Quartz, those functions can be seen as operations on the type *set*, which have been introduced due to the choice statements in the last section. They allow to reason about all available/chosen values the concrete set variable s stands for.

In order to introduce and use those aggregate functions in Quartz, the following expressions are proposed to be introduced:

- `x = Count(setExp)`
- `x = Sum(numberSetExp)`
- `x = Max(numberSetExp)`
- `x = Min(numberSetExp)`

The `Count` expression should return the amount of elements in the set and therefore works on all types of elements in a *set*. The aggregate function `Sum` needs a *number* type in order to allow building the sum of all elements in the set. As the two functions `Max` and `Min` need an according order relation, they should also only be defined on sets of numbers in Quartz.

In the two shown examples for the power of ASP/stable models, those aggregate functions have been used quite often.

For example, the $\#max$ aggregate was used to calculate the longest path through a graph:

$$\begin{aligned} pathCosts(X,1) &:- initialNode(X). \\ pathCosts(Y,N+1) &:- edge(X,Y),\ pathCosts(X,N),\ N < (M+1), \\ &\quad\ amountVars(M). \end{aligned}$$

$$maximalCost(N) :- N = \#max\{C : pathCosts(\_,C)\}.$$

Let's consider all `pathCosts` being available in an according set variable in Quartz. Then, the maximal cost can be defined as follows:

```
module Aggregate() {
   event nat maximalCost;
   event {}nat pathCosts;
   ...
   pathCosts = ...;
   maximalCost = Max(pathCosts);
}
```

4.3.5. Optimizations

As a final construct available in ASP and should be made available in Quartz, optimization statements have to be mentioned. The optimization statements $\#maximize(C)$ and $\#minimize(C)$ allow to optimize an answer set or stable

model according to a given count function C. In the shown ASP examples, all optimization statements were used as the final goal definition for the according problem.

E.g. to minimize the amount of switches in the concentrator circuit, the following ASP rule has been used:

$$\#minimize\{N : num\_switches(N)\}.$$

The same holds for the second example, which finally wanted to minimize the amount of PUs.

$$\#minimize\{N : amountPUs(N)\}.$$

The optimization statements can be introduced into Quartz by adding the standalone statements `Minimize(numberVar)` and `Maximize(numberVar)`. The mentioned ASP statements could therefore lead to the following Quartz module definition:

```
module Optimizations1() {
   event nat num_switches;
   ...
   num_switches = ...;
   Minimize(num_switches);
}

module Optimizations2() {
   event nat amountPUs;
   ...
   amountPUs = ...;
   Minimize(amountPUs);
}
```

All in all, this chapter sketches, how synchronous languages like Quartz can be extended to a full front-end for ASP languages with only small adaptions. Hereby, the most important part is the introduction of a new type *set*, which allows to reason about multiple values in synchronous languages, while it therefore stands for all instantiations of an according ASP term. By allowing the definition of additional *constraints* and with the ability to have operations on sets with *aggregates* like to calculate the sum or to count the elements in the set, also *optimization* statements could be introduced with an according meaning. Finally, this sketch shifts the perspective from Quartz as a language to define algorithms to a language showing its benefits especially by the definition of search and optimization problems in a descriptive way, like ASP languages do.

Chapter 5

Conclusions

The thesis in hand started with an introduction of current known semantics of logic programming, especially the well-founded semantics and the stable model semantics used in state-of-the-art ASP solvers. Further, a connection was drawn to the semantics of current state-of-the-art synchronous languages, which are based on an instantiation of the Fitting's fixpoint semantics. Hereby, the synchronous language Quartz and the current semantics have been introduced in detail.

The next section focused on the semantics extension of synchronous languages in order to level up the possibilities of the interpretation of synchronous programs, such that new programs could be defined a meaning, and which do not lead to a meaning and calculation result with the current used Fitting's fixpoint semantics.

To update the semantics of synchronous programs to the well-founded semantics of logic programming, two different levels of abstractions have been shown. First of all, the well-founded semantics have been introduced as an extension to the abstract SOS rules and to the abstract synchronous program interpreter procedure. This introduced abstract extension allowed to reason about the semantics from a theoretical point of view. As another option, a more practical approach has been shown, which takes an existing program and extends this program itself in order that it is automatically evaluated according to the well-founded semantics. This practical extension does not need to adapt any existing tools for synchronous languages and allows to use the fixpoint iterations of Fitting's semantics to interpret the program in the same matter, as the well-founded semantics would have done. That this procedure is a conservative extension of Fitting's fixpoint semantics has been proved.

In order to interpret synchronous programs also with the stable model semantics of ASP, another step had to be taken. First of all it was shown, that a similar extension of the program and semantics in the same matter as it was performed for the well-founded semantics could not be used here. This basically comes from the non-monotonicity of the stable model semantics. Hence, another approach was shown to interpret synchronous programs with the stable model semantics: by a compilation from every synchronous program to

the ASP-core-2-language format. This compiler defined not only how the different expressions and types of Boolean, array or numbers can be transformed to an according ASP program, but it was also implemented by a prototype. This allowed to show the feasibility of the approach, where different example benchmark programs had been completely compiled to ASP and could then be interpreted with current state-of-the-art ASP solvers.

But how can a programmer of synchronous programs directly benefit from this compilation and the power of stable model semantics? This was the focus of the next section, where two complex examples for the practical use of ASP have been introduced. The first example defined an according search and optimization program to find minimal concentrator circuits, which can move multiple values without blocking from inputs of the interconnection network to their outputs with according properties in parallel. The second example was then the definition of a complete compiler chain from synchronous programs to optimal move code for the synchronous control asynchronous data (SCAD) architecture, which was developed in the embedded systems group of the University of Kaiserslautern. Hereby, the heart of this compiler to find optimal scheduling for that architecture was developed with the help of ASP and the stable model semantics.

Having defined and analyzed those two complex examples, one could realize that the current ASP-core-2 language contains very helpful additional constructs, which especially allow to define and handle search and optimization problems in a very descriptive way. As an compilation from synchronous languages to ASP programs was already available, it was quite obvious to raise the question if synchronous languages can be seen in general as a more high-level access language to stable model semantics and ASP solvers than the ASP-core-2 language. To answer this question it was sketched how those advanced constructs, which could not directly be reached from current synchronous language expressions, can be introduced into the language design of synchronous languages. With the help of the introduced syntax and translation scheme of *Constraints*, *Disjunctive Assignments*, *Choices*, *Aggregates* and *Optimization Statements*, the hypothesis is raised that synchronous languages are a very valid high-level access language for the stable model semantics.

All in all, the current semantics and the use of synchronous languages have been analyzed and extended in this thesis to newer available semantics as the well-founded semantics or stable model semantics of logic programs. This allows not only to interpret more programs in synchronous languages than before, but it especially shifts the focus away from concrete algorithm definitions to descriptive problems and solution descriptions. During the work on this thesis, the feeling was raised that this shift and descriptive nature is very constructive and fits to the core idea of synchronous languages and synchronous semantics much better than the fixed algorithm definition synchronous languages have mostly been used for so far.

Bibliography

[ACPV11] F. Aguado, P. Cabalar, G. Pérez, and C. Vidal. "Loop Formulas for Splitable Temporal Logic Programs". In: *Logic Programming and Nonmonotonic Reasoning (LPNMR)*. Ed. by J.P. Delgrande and W. Faber. Vol. 6645. LNCS. Vancouver, Canada: Springer, 2011, pp. 80–92.

[AMHF15] J. Aguado, M. Mendler, R. von Hanxleden, and I. Fuhrmann. "Denotational Fixed-point Semantics for Constructive Scheduling of Synchronous Concurrency". In: *Acta Informatica* 52.4 (2015), pp. 393–442.

[AgMe11] J. Aguado and M. Mendler. "Constructive Semantics for Instantaneous Reactions". In: *Theoretical Computer Science (TCS)* 412.11 (2011), pp. 931–961.

[Andr95] C. André. *SyncCharts: A Visual Representation of Reactive Behaviors*. Research Report tr95-52. University of Nice, Sophia Antipolis, France, 1995.

[Andr96] C. André. *Representation and Analysis of Reactive Behaviors: A Synchronous Approach*. Research Report tr96-28. University of Nice, Sophia Antipolis, France, 1996.

[Apt03] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[BaAB04] F. Banti, J.J. Alferes, and A. Brogi. "Well Founded Semantics for Logic Program Updates". In: *Ibero-American Conference on Artificial Intelligence*. Ed. by C. Lemaître, C.A. Reyes García, and J.A. González. Vol. 3315. LNCS. Puebla, Mexico: Springer, 2004, pp. 397–407.

[Batc68] K.E. Batcher. "Sorting Networks and their Applications". In: *AFIPS Spring Joint Computer Conference*. Vol. 32. 1968, pp. 307–314.

[BoSi91] F. Boussinot and R. de Simone. "The Esterel language". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1293–1304.

[BCEH03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. "The Synchronous Languages Twelve Years Later". In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.

[Berr99] G. Berry. *The Constructive Semantics of Pure Esterel*. 1999.

[BeGo92] G. Berry and G. Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152.

[BhJS16] A. Bhagyanath, T. Jain, and K. Schneider. "Towards Code Generation for the Synchronous Control Asynchronous Dataflow (SCAD) Architectures". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Ed. by R. Wimmer. Freiburg, Germany: University of Freiburg, 2016, pp. 77–88.

[BrNT03] G. Brewka, I. Niemela, and M. Truszczynski. "Answer Set Optimization". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by G. Gottlob and T. Walsh. Acapulco, Mexico: Morgan Kaufmann, 2003, pp. 867–872.

[BGSS11] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, and J.-P. Talpin. "Integrating System Descriptions by Clocked Guarded Actions". In: *Forum on Specification and Design Languages (FDL)*. Ed. by A. Morawiec, J. Hinderscheit, and O. Ghenassia. Oldenburg, Germany: IEEE Computer Society, 2011, pp. 1–8.

[BGSS12] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. "Representation of Synchronous, Asynchronous, and Polychronous Components by Clocked Guarded Actions". In: *Design Automation for Embedded Systems (DAEM)* (2012). DOI 10.1007/s10617-012-9087-9.

[BrSc08a] J. Brandt and K. Schneider. "Formal Reasoning About Causality Analysis". In: *Theorem Proving in Higher Order Logics (TPHOL)*. Ed. by O. Ait Mohamed, C. Muñoz, and S. Tahar. Vol. 5170. LNCS. Montréal, Québec, Canada: Springer, 2008, pp. 118–133.

[BrSc11a] J. Brandt and K. Schneider. *Separate Translation of Synchronous Programs to Guarded Actions*. Internal Report 382/11. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, 2011.

[BhSc16] A. Bhagyanath and K. Schneider. "Optimal Compilation for Exposed Datapath Architectures with Buffered Processing Units by SAT Solvers". In: *Formal Methods and Models for Codesign (MEMOCODE)*. Ed. by E. Leonard and K. Schneider. Kanpur, India: IEEE Computer Society, 2016, pp. 143–152.

[BhSc17] A. Bhagyanath and K. Schneider. "Exploring the Potential of Instruction-Level Parallelism of Exposed Datapath Architectures with Buffered Processing Units". In: *Application of Concurrency to System Design (ACSD)*. Ed. by A. Legay and K. Schneider. Zaragoza, Spain: IEEE Computer Society, 2017, pp. 106–115.

[CKSS18] P. Cabalar, R. Kaminski, T. Schaub, and A. Schuhmann. "Temporal Answer Set Programming on Finite Traces". In: *Theory and Practice of Logic Programming (TPLP)* 18.3-4 (2018), pp. 406–420.

[CFGI13] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. *ASP-Core-2 Input Language Format*. ASP Standardization Working Group. 2013.

[CFGI20] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub. "ASP-Core-2 Input Language Format". In: *Theory and Practice of Logic Programming (TPLP)* 20.2 (2020), pp. 294–309.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice. "LUSTRE: A declarative language for programming synchronous systems". In: *Principles of Programming Languages (POPL)*. Munich, Germany: ACM, 1987, pp. 178–188.

[CaDi11] P. Cabalar and M. Diéguez. "STeLP - A Tool for Temporal Answer Set Programming". In: *Logic Programming and Nonmonotonic Reasoning (LPNMR)*. Ed. by J.P. Delgrande and W. Faber. Vol. 6645. LNCS. Vancouver, Canada: Springer, 2011, pp. 370–375.

[Clar77] K.L. Clark. "Negation as Failure". In: *Logic and Data Bases*. Ed. by H. Gallaire and J. Minker. Toulouse, France: Plemum Press, New York, 1977, pp. 293–322.

[CoRo96] A. Colmerauer and P. Roussel. "The Birth of Prolog". In: *History of Programming Languages II*. Ed. by T.J. Bergin and R.G. Gibson. ACM, 1996. Chap. 7, pp. 331–367.

[CaSc19] P. Cabalar and T. Schaub. "Dynamic and Temporal Answer Set Programming on Linear Finite Traces". In: *Datalog*. Vol. 2368. CEUR, 2019, pp. 3–6.

[DJSG17] M. Dahlem, T. Jain, K. Schneider, and M. Gillmann. "Automatic Synthesis of Optimal-Size Concentrators by Answer Set Programming". In: *Logic Programming and Nonmonotonic Reasoning (LPNMR)*. Ed. by M. Balduccini and T. Janhunen. Vol. 10377. LNAI. Espoo, Finland: Springer, 2017, pp. 279–285.

[DaBS18b] M. Dahlem, A. Bhagyanath, and K. Schneider. "Optimal Scheduling for Exposed Datapath Architectures with Buffered Processing Units by ASP". In: *Theory and Practice of Logic Programming (TPLP)* 18.1 (2018), pp. 438–451.

[DoGa84] W.F. Dowling and J.H. Gallier. "Linear-time algorithms for testing the satisfiability of propositional Horn formulae". In: *The Journal of Logic Programming* 1.3 (1984), pp. 267–284.

[RoLM10] W.-P. de Roever, G. Lüttgen, and M. Mendler. "What Is in a Step: New Perspectives on a Classical Question". In: *Time for Verification – Essays in Memory of Amir Pnueli*. Ed. by Z. Manna and D. Peled. Vol. 6200. LNCS. Springer, 2010, pp. 370–399.

[DaSc20] M. Dahlem and K. Schneider. "Compiling Synchronous Languages to Optimal Move Code for Exposed Datapath Architectures". In: *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. Ed. by S. Stuijk. Sankt Goar, Germany: ACM, 2020.

[Dung92] P.M. Dung. "On the relations between stable and well-founded semantics of logic programs". In: *Theoretical Computer Science* 105.1 (1992), pp. 7–25.

[ErGL16] E. Erdem, M. Gelfond, and N. Leone. "Applications of Answer Set Programming". In: *AI Magazine* 37.3 (2016), pp. 53–68.

[EiIK09] T. Eiter, G. Ianni, and T. Krennwallner. "Answer Set Programming: A Primer". In: *Reasoning Web. Semantic Technologies for Information Systems*. Ed. by S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.-C. Rousset, and R.A. Schmidt. Vol. 5689. LNCS. Brixen-Bressanone, Italy: Springer, 2009, pp. 40–110.

[Fage94] F. Fages. "Consistency of Clark's completion and existence of stable models". In: *Methods of Logic in Computer Science* 1.1 (1994), pp. 51–60.

[FFST18] A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, and E.C. Teppan. "Industrial Applications of Answer Set Programming". In: *KI – Künstliche Intelligenz* 32.2 (2018), pp. 165–176.

[FeEr81] M. Feller and M.D. Ercegovac. "Queue machines: An organization for parallel computation". In: *Conpar 81*. Ed. by W. Brauer, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, N. Wirth, and Wolfgang Händler. Vol. 111. LNCS. Nürnberg, Germany: Springer, 1981, pp. 37–47.

[Fitt85] M. Fitting. "A Kripke-Kleene Semantics for Logic Programs". In: *Journal of Logic Programming* 2.4 (1985), pp. 295–312.

[Fitt93] M. Fitting. "The family of stable models". In: *The Journal of Logic Programming* 17.2-4 (1993), pp. 197–225.

[FeLL07] P. Ferraris, J. Lee, and V. Lifschitz. "A New Perspective on Stable Models". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by M.M. Veloso. Vol. 7. Hyderabad, India, 2007, pp. 372–379.

[FeLL11] P. Ferraris, J. Lee, and V. Lifschitz. "Stable Models and Circumscription". In: *Artificial Intelligence* 175.1 (2011), pp. 236–263.

[GKNS07b] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. "Conflict-Driven Answer Set Solving". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by M.M. Veloso. Vol. 7. Hyderabad, India, 2007, pp. 386–392.

[GKKS19] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. "Multi-Shot ASP Solving with Clingo". In: *Theory and Practice of Logic Programming (TPLP)* 19.1 (2019), pp. 27–82.

[GeKS11] M. Gebser, R. Kaminski, and T. Schaub. "Complex Optimization in Answer Set Programming". In: *Theory and Practice of Logic Programming (TPLP)* 11.4-5 (2011), pp. 821–839.

[GeKS12] M. Gebser, B. Kaufmann, and T. Schaub. "Conflict-Driven Answer Set Solving: From Theory to Practice". In: *Artificial Intelligence* 187 (2012), pp. 52–89.

[GeLi88] M. Gelfond and V. Lifschitz. "The Stable Model Semantics for Logic Programming". In: *Logic Programming*. Ed. by R.A. Kowalski and K.A. Bowen. Seattle, Washington, USA: MIT Press, 1988, pp. 1070–1080.

[GeLi91] M. Gelfond and V. Lifschitz. "Classical Negation in Logic Programs and Disjunctive Databases". In: *New Generation Computing* 9.3-4 (1991), pp. 365–385.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The Synchronous Dataflow Programming Language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[Halb05] N. Halbwachs. "A synchronous language at work: the story of Lustre". In: *Formal Methods and Models for Codesign (MEMOCODE)*. Verona, Italy: IEEE Computer Society, 2005, pp. 3–11.

[Halb93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[HPSS87] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. "On the formal semantics of Statecharts". In: *Logic in Computer Science (LICS)*. Ithaca, New York, USA: IEEE Computer Society, 1987, pp. 54–64.

[Hare87] D. Harel. "Statecharts: A visual formulation for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274.

[Helj99] K. Heljanko. "Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by R. Cleaveland. Vol. 1579. LNCS. Amsterdam, The Netherlands: Springer, 1999, pp. 240–254.

[Horn51] A. Horn. "On sentences which are true of direct unions of algebras". In: *Journal of Symbolic Logic* 16.1 (1951), pp. 14–21.

[JaBS13] E.K. Jackson, N. Bjorner, and W. Schulte. *Open-world Logic Programs: A New Foundation for Formal Specifications*. Technical Report MSR-TR-2013-55. Microsoft Research Lab, 2013.

[JaSc16] T. Jain and K. Schneider. "Verifying the Concentration Property of Permutation Networks by BDDs". In: *Formal Methods and Models for Codesign (MEMOCODE)*. Ed. by E. Leonard and K. Schneider. Kanpur, India: IEEE Computer Society, 2016, pp. 43–53.

[KLPS16b] B. Kaufmann, N. Leone, S. Perri, and T. Schaub. "Grounding and Solving in Answer Set Programming". In: *AI Magazine* 37.3 (2016), pp. 25–32.

[Kowa74] R. Kowalski. "Predicate Logic as Programming Language". In: *IFIP Congress*. Stockholm, Sweden, 1974, pp. 569–574.

[KaSW17] R. Kaminski, T. Schaub, and P. Wanko. "A Tutorial on Hybrid Answer Set Solving with clingo". In: *Reasoning Web. Semantic Interoperability on the Web*. Ed. by G. Ianni, D. Lembo, L.E. Bertossi, W. Faber, B. Glimm, G. Gottlob, and S. Staab. Vol. 10370. LNCS. London, UK: Springer, 2017, pp. 167–203.

[GGBM91] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. "Programming real-time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1321–1336.

[Lier17] Y. Lierler. "What is Answer Set Programming to Propositional Satisfiability?" In: *Constraints* 22.3 (2017), pp. 307–337.

[Lifs10] V. Lifschitz. "Thirteen Definitions of a Stable Model". In: *Fields of Logic and Computation*. Ed. by A. Blass, N. Dershowitz, and W. Reisig. Vol. 6300. LNCS. Springer, 2010, pp. 488–503.

[LeRS97] N. Leone, P. Rullo, and F. Scarcello. "Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation". In: *Information and Computation* 135.2 (1997), pp. 69–112.

[LiZh04b] F. Lin and Y. Zhao. "ASSAT: Computing answer sets of a logic program by SAT solvers". In: *Artificial Intelligence* 157 (2004), pp. 115–137.

[Mara91] F. Maraninchi. "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems". In: *Visual Languages*. IEEE Computer Society, 1991.

[Mara92] F. Maraninchi. "Operational and Compositional Semantics of Synchronous Automaton Compositions". In: *Concurrency Theory (CONCUR)*. Ed. by R. Cleaveland. Vol. 630. LNCS. Stony Brook, New York, USA: Springer, 1992, pp. 550–564.

[McDe82a] D. McDermott. "Nonmonotonic Logic II: Nonmonotonic Modal Theories". In: *Journal of the ACM (JACM)* 29.1 (1982), pp. 33–57.

[MeLu00] M. Mendler and G. Luettgen. *What is in a Step: A Fully-abstract Semantics for Statecharts Macro Steps via Intuitionistic Kripke Models*. Research Memorandum CS-00-04. University of Sheffield, Sheffield, UK, 2000.

[MeMa20] C. Mencía and J. Marques-Silva. "Reasoning About Strong Inconsistency in ASP". In: *Theory and Applications of Satisfiability Testing (SAT)*. Ed. by L. Pulina and M. Seidl. Vol. 12178. LNCS. Alghero, Italy: Springer, 2020, pp. 332–342.

[MaRe01] F. Maraninchi and Y. Rémond. "Argos: An Automaton-Based Synchronous Language". In: *Computer Languages* 27.1 (2001), pp. 61–92.

[MoSc11] A. Morgenstern and K. Schneider. "Synthesis of Parallel Sorting Networks using SAT Solvers". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Ed. by F. Oppenheimer. Oldenburg, Germany: OFFIS-Institut für Informatik, 2011, pp. 71–80.

[NWSH17] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt. "Enhancing symbolic system synthesis through ASPmT with partial assignment evaluation". In: *Design, Automation and Test in Europe (DATE)*. Ed. by D. Atienza and G. Di Natale. Lausanne, Switzerland: IEEE Computer Society, 2017, pp. 306–309.

[NWSH18] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt. "Exact multi-objective design space exploration using ASPmT". In: *Design, Automation and Test in Europe (DATE)*. Ed. by A.K. Coskun. Dresden, Germany: IEEE Computer Society, 2018, pp. 257–260.

[ObRS19] P. Obermeier, J. Romero, and T. Schaub. "Multi-Shot Stream Reasoning in Answer Set Programming: A Preliminary Report". In: *Open J. Databases (OJDB)* 6.1 (2019), pp. 33–38.

[Parb92] I. Parberry. "The Pairwise Sorting Network". In: *Parallel Processing Letters (PPL)* 2.2-3 (1992), pp. 205–211.

[Pear96] D. Pearce. "A New Logical Characterisation of Stable Models and Answer Sets". In: *International Workshop on Non-monotonic Extensions of Logic Programming*. Ed. by J. Dix, L.M. Pereira, and T.C. Przymusinski. Vol. 1216. LNCS. Bad Honnef, Germany: Springer, 1996, pp. 57–70.

[Pins73] M.S. Pinsker. "On the Complexity of a Concentrator". In: *International Teletraffic Conference (ITC)*. Stockholm, Sweden, 1973, 318:1–318:4.

[Plot81] G.D. Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. FN-19. Århus, Denmark: DAIMI, 1981.

[Przy88] T.C. Przymusinski. “On the Declarative Semantics of Deductive Databases and Logic Programs”. In: *Foundations of Deductive Databases and Logic Programming*. San Francisco, CA, USA: Morgan Kaufmann, 1988, pp. 193–216.

[PnSh91] A. Pnueli and M. Shalev. “What is in a step: On the semantics of statecharts”. In: *Theoretical Aspects of Computer Software (TACS)*. Ed. by T. Ito and A.R. Meyer. Vol. 526. LNCS. Sendai, Japan: Springer, 1991, pp. 244–264.

[QuWi05] B.R. Quinton and S.J.E. Wilton. “Concentrator Access Networks for Programmable Logic Cores on SoCs”. In: *International Symposium on Circuits and Systems (ISCAS)*. Vol. 1. Kobe, Japan: IEEE Computer Society, 2005, pp. 45–48.

[Reit01] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT press, 2001.

[RFMR18] T. Ribeiro, M. Folschette, M. Magnin, O. Roux, and K. Inoue. “Learning Dynamics with Synchronous, Asynchronous and General Semantics”. In: *International Conference on Inductive Logic Programming (ILP)*. Ed. by F. Riguzzi, E. Bellodi, and R. Zese. Vol. 11105. LNCS. Ferrara, Italy: Springer, 2018, pp. 118–140.

[ScBS04b] K. Schneider, J. Brandt, and T. Schüle. “Causality Analysis of Synchronous Programs with Delayed Actions”. In: *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. Washington, District of Columbia, USA: ACM, 2004, pp. 179–189.

[SNLG18] B. Schäpers, T. Niemueller, G. Lakemeyer, M. Gebser, and T. Schaub. “ASP-Based Time-Bounded Planning for Logistics Robots”. In: *International Conference on Automated Planning and Scheduling (ICAPS)*. Ed. by S. de Weerdt, S. Koenig, G. Röger, and M.T.J. Spaan. Vol. 28. 1. Delft, The Netherlands: AAAI Press, 2018, pp. 509–517.

[Schn09] K. Schneider. *The Synchronous Programming Language Quartz*. Internal Report 375. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, 2009.

[ScDa18] K. Schneider and M. Dahlem. “Are Synchronous Programs Logic Programs?” In: *Principled Software Development*. Ed. by P. Müller and I. Schäfer. Springer Nature Switzerland, 2018, pp. 251–266.

[ScLY02] H. Schmit, B. Levine, and B. Ylvisaker. “Queue machines: hardware compilation in hardware”. In: *Field-Programmable Custom Computing Machines (FCCM)*. Ed. by J. Arnold and K.L. Pocek. Napa, California, USA: IEEE Computer Society, 2002, pp. 152–160.

[ScWe01] K. Schneider and M. Wenz. “A new method for compiling schizophrenic synchronous programs”. In: *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. Atlanta, Georgia, USA: ACM, 2001, pp. 49–58.

[SaZa90] D. Saccà and C. Zaniolo. “Stable Models and Non-Determinism in Logic Programs with Negation”. In: *Principles of Database Systems (PODS)*. Ed. by D.J. Rosenkrantz and Y. Sagiv. Nashville, Tennessee, USA: ACM, 1990, pp. 205–217.

[Tars55] A. Tarski. “A Lattice-Theoretical Fixpoint Theorem and its Applications”. In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309.

[EmKo76] M. van Emden and R. Kowalski. “The semantics of predicate logic as a programming language”. In: *Journal of the ACM (JACM)* 23.4 (1976), pp. 733–742.

[GeRS91] A. van Gelder, K.A. Ross, and J.S. Schlipf. “The Well-Founded Semantics for General Logic Programs”. In: *Journal of the ACM (JACM)* 38.3 (1991), pp. 620–650.

[YBFH16] Z. Yang, J.-P. Bodeveix, M. Filali, K. Hu, Y. Zhao, and D. Ma. “Towards a verified compiler prototype for the synchronous language SIGNAL”. In: *Frontiers of Computer Science* 10.1 (2016), pp. 37–53.

Appendix A

Curriculum Vitae

Berufserfahrung

seit 2014	**Software Engineer** Insiders Technologies GmbH Kaiserslautern Abteilung MOBILE
2013–2014	**Studentische Hilfskraft** DFKI GmbH Standort Kaiserslautern Forschungsgruppe Eingebettete Intelligenz
2011–2012	**Studentische Hilfskraft** TU Kaiserslautern Zentrale Einrichtung Allgemeiner Hochschulsport
2010–2011	**Studentische Hilfskraft** TU Kaiserslautern Fachbereich Informatik, Arbeitsgruppe Bildverstehen und Mustererkennung

Akademische Ausbildung

2012–2014	**Master of Science in Informatik** TU Kaiserslautern Masterarbeit: *Interactive Verification of Synchronous Systems in HOL*
2012–2013	**Auslandssemester in Schweden** Luleå University of Technology
2008–2012	**Bachelor of Science Informatik** TU Kaiserslautern Bachelorarbeit: *Systematische Beschreibung und Werkzeugunterstützung textueller Produktlinien*

Schulausbildung

2004-2007	**Abitur** Albert-Schweitzer-Gymnasium, Kaiserslautern