Chapter 1
Are Synchronous Programs Logic Programs?

Klaus Schneider and Marc Dahlem

Abstract
Synchronous languages have been introduced as programming languages that directly reflect the nature of reactive systems: Their execution is divided into discrete reaction steps such that in each reaction step, inputs are read from the environment and outputs are instantaneously computed. Reaction steps, which are also called macro steps, consist thereby of a set of atomic micro step actions that are executed in the variable environment associated with the macro step. At the beginning of the macro step, only the values of the input variables are known in this variable environment, and the values of the remaining variables have to be computed according to the data-dependencies. Since the micro step actions depend on the variable environment that they also create, it might be the case that there are cyclic dependencies. Whether such cyclic dependencies can be constructively resolved has to be checked by a compile-time causality analysis which will ensure that there is for all inputs a suitable schedule of the micro steps. If the synchronous programs are converted to guarded actions as done in the author’s Averest system, some relationships with logic programs can be found: In particular, the concepts of reaction-to-absence of synchronous languages and negation-to-failure of logic programs seem to be the same; another analogy is found for the generation of equation-based code of synchronous programs and the completion of logic programs, and also for the fix-point analyses defined in both paradigms. This paper outlines these analogies between the two paradigms of programming languages and discusses whether further known semantics of logic programs like well-founded and stable models may find useful counterparts in the synchronous world in future.

Klaus Schneider
TU Kaiserslautern, 67653 Kaiserslautern, Germany, https://es.cs.uni-kl.de, e-mail: schneider@cs.uni-kl.de

Marc Dahlem
Insiders Technologies GmbH, Kaiserslautern, Germany, https://insiders-technologies.de, e-mail: m_dahlem@cs.uni-kl.de
1.1 Introduction

In contrast to transformational systems that read inputs only at starting time and provide their outputs only at termination time, reactive systems [32, 30] have an ongoing computation that is divided into so-called reaction steps. Within every reaction step, the reactive system reads inputs given by its environment and instantaneously computes the outputs for the environment in the same macro step. Reactive systems occur in many areas, in particular, most embedded systems are reactive in nature, and also many distributed systems have a notion of rounds that correspond to reaction steps. While the idea of reactive systems seems very simple, it turned out that the semantics of reactive programming languages turned out to be a challenge as was early realized by the family of statechart languages [44, 31, 58, 24].

Synchronous languages [28, 3] have been introduced as a new class of programming languages for reactive systems. Classic synchronous languages were Esterel [6, 8], Lustre [16, 29], and Signal [26, 38] that followed different paradigms: While Esterel is an imperative language, Lustre is data-flow oriented, and Signal is even declarative. Still, all of these languages have a notion of a reaction step where input signals are given by the environment, and where output signals have to be instantaneously computed by the synchronous program. These reaction steps are defined by a set of atomic actions that are in the programmer’s view executed in zero time, i.e., in the same reaction step. As a consequence, the atomic actions of a reaction step have an instantaneous feedback to the variable environment of their reaction step. Reaction steps and atomic actions are also often called macro and micro steps, respectively.

The distinction between macro and micro steps is an elegant abstraction that allows programmers to reason about their programs in terms of macro steps so that there is no need to worry about the schedules of the micro steps. However, the compilers have to make sure that this abstraction is meaningful: Due to the instantaneous feedback of the atomic micro step actions to their variable environment, reaction steps may have for one and the same input no consistent behavior or even several consistent behaviors. To ensure determinism, compilers for synchronous languages have to analyze the programs for consistency. Depending on the synchronous language, the main consistency analyses are clock consistency and causality analysis [5, 52, 49, 50].

Causality problems arise from the fact that synchronous programs can read – and therefore instantaneously react to – their own outputs. This way, they may assign a value to a variable because that variable is already seen to have this value, or they may instantaneously assign another value to a variable because that variable has another value in this step. Of course, these cyclic dependencies may be also transitive via other variables, and they may involve arbitrary data-dependent conditions as well. Causality problems arise also in other areas where abstractions to reaction steps were made: For example, the composition of Mealy automata suffers from the same problems, and also in the synthesis of digital hardware circuits, the false cycle problem is well-known.
A simple solution to avoid causality problems is to forbid all cyclic dependencies of micro step actions. This is, for example, done in STATEMATE and also in the Lustre language, and it can be easily achieved by either syntactic checks of the data-dependencies or by simply delaying the outputs by one reaction step. However, it was already observed in the early seventies [33, 34, 46] that digital hardware circuits can have meaningful combinational feedback loops. It was also already known that there are circuits with combinational feedback loops that are smaller than all equivalent circuits without such loops. This is also exploited today in hardware synthesis in that operands and results are shared using multiplexers, and this may lead to false loops [54, 40, 41]. False loops are pseudo-cycles in the sense that they syntactically appear as cyclic dependencies that will however never show up at runtime. One reason may be that different multiplexers will make always the same or opposite selections with is not easily seen without knowing all the reachable states.

Malik [40, 41] was the first who introduced algorithms to analyze whether feedback loops in circuits are combinational or lead to unwanted states. Malik’s analysis is based on the ternary symbolic simulation of the circuits due to Bryant [11] that was also used for the analysis of asynchronous circuits [51, 14, 12, 15, 13] by Brzozowski and Seger. Since the abstraction from asynchronous circuits to synchronous circuits is comparable to the abstraction from micro steps to macro steps in general, Shiple, Berry, and Touati [53] realized that the causality analysis of synchronous languages can be dealt with the same algorithms and introduced thereby the notion of constructive circuits [53, 5] which are circuits whose output values stabilize for all possible gate and wire delays. The third truth value \( \perp \) has the meaning that the value has not yet been determined and it may change in the progress of the macro step to one of the two boolean truth values 0 or 1. It was conjectured that a logical definition based on a constructive logic, a semantic definition based on three-valued lattices, and the electrical definition should be equivalent.

Berry defined then the semantics of Esterel based on a ternary analysis of control-flow conditions that can also be formulated using abstract interpretation by a may/must analysis. We also followed this great idea and defined the semantics of our Quartz language [48] using SOS transition and reaction rules which are based on a multivalued logic that directly reflects the information flow during macro steps. Hence, semantics of synchronous programs – like Esterel and Quartz – are defined by multivalued logics where the truth value \( \perp \) is used for not yet determined variables.

In ternary logic, where we define \( \neg \perp := 0 \perp \) and \( \perp \lor x := x \lor \perp := 1 \) if \( x = 1 \) and \( \perp \) otherwise, we cannot prove certain two-valued theorems like \( \varphi \lor \neg \varphi \). If \( \varphi \) evaluates to \( \perp \), then also \( \neg \varphi \) does so, and no information has yet arrived at this place. Causality demands however that some partial information will trigger the computation of further information which is not possible in these examples.

Similarly, \( \varphi \lor \neg \varphi \) cannot be proved in constructive logic. A proof for \( \varphi \) in constructive logic would require a way/algorithm to construct \( \varphi \), and a proof for \( \neg \varphi \) would require to prove that no such algorithm exists. Even though this suggests that multivalued logic and constructive logic have some similarities, there are good arguments why the relationship between causality analysis and constructive logic
cannot be as simple as suggested at first: Kurt Gödel already proved that constructive logic is not equivalent to any multivalued logic. Since causality analysis is defined by ternary logic, it cannot be equivalent to constructive logic. Another argument to see this difference is the double negation theorem of constructive logic due to Glivenko: Any propositional formula \( \phi \) is valid in classic logic if and only if \( \neg\neg\phi \) is valid in intuitionistic logic. However, in ternary logic, \( \neg\neg\phi \) is equivalent to \( \phi \) and thus, makes no difference. In constructive logic, also other forms of negation have been considered like ‘negation as implication of impossibility’ also known as ‘strong negation’. We therefore see that while there are similarities to constructive logic, there are also many differences, so that there is no simple equivalence to constructive logic.

The main motivation of the use of ternary logic for causality analysis was to describe with the value \( \bot \) that the information flowing from the known input values has not yet been sufficient to generate the considered variable’s value. Besides this positive flow of information, there is another very important concept in synchronous languages: Due to the notion of macro steps, there is logical notion of time, and if a value does not arrive within such a time step, we can conclude that no action is there to determine it in this step. For this reason, the reaction to absence is applied and determines the variable’s value as a default value like 0 for boolean values.

In the Averest system, the Quartz programs are first translated to synchronous guarded actions which are essentially pairs \( (\gamma, \alpha) \) with a control-flow condition \( \gamma \) and an atomic action \( \alpha \) having the simple meaning that whenever \( \gamma = 1 \) holds, then \( \alpha \) must be instantaneously executed. In previous work [48, 10], we have shown how Quartz programs can be compiled to such guarded actions, and in further work [9, 55], we also showed how other synchronous languages can be converted to that intermediate representation. This has also been observed by other researchers in the field [62, 7].

In this paper, we therefore abstract away from a particular synchronous language and consider synchronous guarded actions as a representative for any synchronous language. Causality analysis can be done on the guarded actions as well, and to this end, we have to proceed in rounds within a macro step where in each round, those actions \( \alpha \) are executed whose trigger condition \( \gamma \) is 1 (‘must’ actions). Also, all guarded actions \( (\gamma, \alpha) \) are removed whose trigger condition \( \gamma \) is 0 (‘cannot’ actions). If no action to assign a value to a variable \( x \) that is still having the value \( \bot \) is left, we perform the reaction to absence in that we assign that variable a default value. A little tool for teaching to illustrate this can be found at http://es.cs.uni-kl.de/tools/teaching.

Looking at the causal execution of guarded actions, we found many similarities to logic programs. In particular, the information flow of known conditions \( \gamma \) to compute new information by firing actions \( \alpha \) is similar to the resolution steps done by logic programs. Moreover, the reaction to absence seems to be equivalent to the concept of negation as failure that exists for logic programs. For this reason, we discuss in this paper the relationship between synchronous programs and logic programs. The first author of this paper remembers a discussion he had with Arnd Poetzsch-Heffter about these relationship that turned out to be very interesting, but
has not yet been published. We are not aware of any paper discussing these interesting relationships. We therefore want to address this point here, and will discuss in the following some details about the relationship between synchronous and logic programs.

1.2 Semantics of Logic and Synchronous Programs

In the following, different kinds of semantics of logic programs are considered to compare these semantics with the constructive semantics of synchronous programs. For the sake of simplicity, we restrict the consideration to only propositional variables. There are some obvious goals like the computation of a model of a set of clauses/guarded actions by a polynomial algorithm, and one wants to make sure that either a unique model exists or that at least a canonical model is chosen among the existing ones.

To this end, we first start with basic definitions like Horn clauses and their minimal models that can be computed by a fixpoint analysis that is exactly the same as the one used for synchronous programs. The addition of negation generalizes Horn clauses to logic programs, which also increases their expressiveness. In general, any propositional formula can be written as a logic program (with goal clauses as explained below), but in addition to simple clauses, the rules of a logic program also follow some causality, i.e., because the subgoals/guards are satisfied, the conclusion holds, and if there is no rule to derive the truth of a variable, then this variable is considered to be false. It is therefore not allowed that variables become true without such a justification, which is also a similar point of view as taken in synchronous programs (and in constructive logic).

To consider this justification, one defines the completion of a logic program, which is in essence an equation system that exists in exactly the same form for synchronous programs. The equation system formalizes that a variable is false if no rule will justify it, and true otherwise. However, one loses the causality with the equation system, at least if the equations are read in a logical sense, where equations have no causal direction (in contrast to guarded actions).

The models of the equation system are therefore a general frame to determine the semantics of a logic program, and the same holds for synchronous programs. However, it is disturbing that for given inputs, there might be more than one solution, and therefore one aims at choosing a canonical model for logic programs. For this reason, several attempts were made to define canonical models, and in the following, we review the fixpoint semantics by Fitting [25], which is the same as the constructive semantics of synchronous programs. A generalization of this is the well-founded semantics [57], where a stronger form of the ‘reaction-to-absence’ is used to define the unique fixpoint, and therefore more programs can be given a unique meaning. A recommended survey of this material is [21], and of course, the original references [25, 27, 57, 23, 39] are worth to be read.
1.2.1 Horn Clauses

A clause is a set of literals, i.e., variables or negated variables, and the meaning of this set is the disjunction of these literals. Thus, a set of clauses can be considered as a propositional logic formula in conjunctive normal form.

A **Horn clause** is a clause with at most one positive literal, i.e., \{¬x_1, ..., ¬x_n\} or \{¬x_1, ..., ¬x_n, y\}. These clauses are named after the logician Alfred Horn and play an important role in logic programming and constructive logic.

A Horn clause with exactly one positive literal like \{¬x_1, ..., ¬x_n, y\} is called a **definite clause**; a definite clause without negative literals like \{y\} is called a **fact**; and a Horn clause without a positive literal like \{¬x_1, ..., ¬x_n\} is called a **goal** clause.

Since the literals of a clause are disjunctively connected, one may also write a Horn clause as an implication \(x_1 \land ... \land x_n \rightarrow y\). This implication written in the Prolog and Quartz programming languages are shown in the following table.

| y := x_1, ..., x_n | module P1(event y,x_1,...,x_n) {
|                  |   if(x_1 and ... and x_n) emit(y); |
|                  | } |

The **resolvent** of two Horn clauses is also a Horn clause: To compute a resolvent, one of the clauses must contain a negated literal of the other one. Thus, at least one of the two clauses must have a positive literal, say \{¬x_1, ..., ¬x_n, y\}, while the other one contains that literal \(y\) negatively, say \{¬y, ¬z_1, ..., ¬z_m, u\} or \{¬y, ¬z_1, ..., ¬z_m\}.

The resolvent is in the first case the clause \{¬x_1, ..., ¬x_n, ¬z_1, ..., ¬z_m, u\} and \{¬x_1, ..., ¬x_n, ¬z_1, ..., ¬z_m\} in the second case which are both Horn clauses.

Checking the satisfiability of a set of propositional Horn clauses is P-complete [20], and it can be even solved in linear time. Moreover, Emden and Kowalski showed that every set of definite clauses has a **unique minimal model** [56], and 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.

1.2.2 Minimal Models of Horn Clauses and the Marking Algorithm

**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**
which distinguishes again between facts, definite clauses, and goal clauses that can be alternatively written as follows:

- facts: \( \{ \} \rightarrow y \)
- goal clauses: \( \{ x_1, \ldots, x_n \} \rightarrow \text{false} \)
- definite clauses: \( \{ x_1, \ldots, x_n \} \rightarrow y \)

Given a set of Horn clauses, the marking algorithm first assigns all variables \( y \) of facts the value true and propagates these values in the remaining clauses, i.e., it simplifies the remaining clauses using a partial assignment as follows:

- A definite clause of the form \( \{ x_1, \ldots, x_n \} \rightarrow y \) where \( y \) occurs on the right is thereby removed (since it is already satisfied).
- A definite clause of the form \( \{ x_1, \ldots, x_n, y \} \rightarrow z \) where \( y \) occurs on the left is replaced with \( \{ x_1, \ldots, x_n \} \rightarrow z \), i.e., \( y \) is removed from the premises.
- A goal clause \( \{ x_1, \ldots, x_n, y \} \rightarrow \text{false} \) is replaced with \( \{ x_1, \ldots, x_n \} \rightarrow \text{false} \) if \( n > 0 \) holds. The goal clause \( \{ y \} \rightarrow \text{false} \) leads to a contradiction so that the considered set of Horn clauses is then seen to be unsatisfiable.

This procedure is repeated as long as new facts are derived. If no new facts are derived, the algorithm terminates and returns the so far obtained assignment (meaning that all other variables are interpreted as false). Note that the marking algorithm only makes variables true if needed, i.e., to satisfy new facts, otherwise variables are made false. It therefore obviously generates a minimal model if a model exists at all. Using appropriate data structures, this algorithm can be implemented with a linear runtime [20]. It only returns unsatisfiable if the empty clause \( \{ \} \rightarrow \text{false} \) is detected.

### 1.2.3 Selective Linear Definite Clause (SLD) Resolution

Proving that the addition of a goal clause \( G \) to a set of definite clauses and facts \( P \) makes this set inconsistent is the same as proving that \( P \) implies \( \neg G \), i.e., \( P \) implies all literals that occur in \( G \), i.e., all variables \( y \in G \) are contained in the minimal model (note that all other models include the minimal model).

There is also a syntactic characterization of this logical implication: Based on selective linear definite clause (SLD) resolution [36], one can prove that \( G \) is implied by \( P \): The leftmost literal \( y \) of the goal clause is replaced as follows: if it is a fact, it is simply removed (since proved), otherwise, if there is a definite clause with right hand side \( y \), then \( y \) is replaced with the left hand side of that definite clause. Note that there is some freedom in choosing one of the available definite clauses having this right hand side. If the goal clause becomes finally empty, the goal has been proved. In principle, SLD resolution is a backward reasoning in correspondence to the marking algorithm.
1.2.4 Introducing Negation (Logic Programs)

Horn clauses are not as powerful as general clauses; it is not difficult to find a propositional formula whose conjunctive normal form contains clauses that are no Horn clauses. For this reason, one may wish to introduce ‘some kind’ of negation in the Horn clauses. If this kind of negation would be 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. For this reason, special forms of negations are preferred, as the ‘negation by failure’ [17] meaning 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. For example, consider the following logic program on the left hand side and its corresponding Quartz program on the right hand side:

| p   | module P2(event p,q,r,s) { emit(p); emit(q);         |
| r := p,q if(p and q) emit(r);         |
| s := p,not(q) if(p and not q) emit(s); |

Let us first consider the logic program: Clearly, p is provable since p is a fact. However, q is not provable, since q is neither a fact nor is there a rule having q as conclusion. Thus, q is viewed as being false, and therefore not(q) is declared to be provable. For this reason, s holds, and r is false since the premises of the only rule having r as conclusion are not all true. Thus, negation as failure determines the assignment {p,s}. However, considering ‘not’ as logical negation, there would be also the following further satisfying assignments {p,s}, {p,r,s}, {p,q,r}, {p,q,r,s}.

The Quartz program given on the right hand side makes similar conclusions: It generates the three guarded actions (true.emit(p)), (p & q.emit(r)), and (p&!q.emit(s)) and first starts with the assignment where all variables have the value ⊥. In the first iteration, emit(p) fires, so that the value of p is changed to true. Moreover, it is seen that there is no action that will assign q any value, so q is given the default value false. These changes will now satisfy the trigger of (p&!q.emit(s)) so firing its action will make s true. The trigger of (p & q.emit(r)) is however false, so that also r cannot be assigned any other value than the default value false. The same unique model is found by the causal semantics of Quartz which ignores the logically possible other models.

1.2.5 Completion of Horn Clauses

Keith Clark showed in [17] an explanation of the ‘negation as failure’ in that he considered the completion of a logic program. The completion is obtained by com-
puting for each variable \( y \) an equivalence \( y \leftrightarrow \phi_y \), where \( \phi_y \) is the disjunction of all premises of clauses with conclusion \( y \). Thus, for the above example, we obtain the following completion:

\[
\begin{align*}
p & \leftrightarrow \text{true} \\
q & \leftrightarrow \text{false} \\
r & \leftrightarrow p \land q \\
s & \leftrightarrow p \land \neg q
\end{align*}
\]

In the above example, the model found by negation as failure is the model of the completion of the logic program (meaning the conjunction of the equivalences).

For Quartz programs, a hardware synthesis has been defined in [47] similar to the circuit semantics given by Berry in [5]. In essence, circuits define nothing else than equation systems in that outputs of gates are determined by a Boolean expression in terms of the inputs of the gate. The synthesis of circuits from synchronous program can also be used for software synthesis and has the nice advantage that in contrast to automaton-based code generation, programs of only polynomial size are obtained. For this reason, [48] explicitly describes how to generate from the guarded actions of a Quartz program a corresponding equation system. For Boolean variables this is exactly the Clark completion shown above.

However, there are still some differences between the completion and the semantics of logic programs, and also between the equation systems and the semantics of Quartz programs: for example, consider the following logic program and Quartz programs:

\[
\begin{align*}
p & :\neg \text{not}(p) \\
\end{align*}
\]

For the above single clause \( p :\neg \neg p \), we obtain the completion \( p \leftrightarrow \neg p \) which has no two-valued models, so all formulas are implied by it. However, Prolog answers ‘no’ to the query \( q \) and ‘yes’ to \( \neg q \). The corresponding Quartz on the right hand side is declared to be not constructive by the compiler which still can derive that \( q \) is false, but it cannot assign a value different than \( \bot \) to \( p \). So, while the completion does not yet give a complete characterization of the semantics of logic programs, the Quartz programs still behave in exactly the same way.

### 1.2.6 Fitting’s Fixpoint Semantics is Causal Synchronous Semantics

To characterize the semantics of logic programs, Fitting [25] suggested to use three-valued logic to denote whether a variable is true, false or unknown (\( \bot \)). To compare
this iteration with well-founded models, we formally introduce two transformations of environments. To this end, an environment $I$ is represented as a set of literals with the meaning that $x \in I$ means that $x$ is true, $\neg x \in I$ means that $x$ is false, and if neither is the case, then $x$ is unknown.

For a given logic program $P$ and an environment $I$, we then define the following two functions:

1. $T_P(I)$ is the set of variables $y$ that have a rule $\ell_1, \ldots, \ell_n \rightarrow y$ in $P$ such that all literals $\ell_1, \ldots, \ell_n$ are true in $I$.
2. $F_P(I)$ is the set of variables $y$ where all rules $\ell_1, \ldots, \ell_n \rightarrow y$ in $P$ have at least one literal $\ell_i$ that is false in $I$.

Note that (2) also applies if there is no rule with conclusion $y$. In [25], it has been shown that the minimal model of Horn clauses is the least fixpoint of the function $f_P(I) = T_P(I) \cup \{\neg x | x \in F_P(I)\}$. Moreover, it has been proved in [25] 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 $f_P$.

Again, we find exactly the same kind of reasoning for the synchronous programs: The above function $T_P(I)$ corresponds with that part of the causality analysis that determines the ‘must’ actions of a program, i.e., those guarded actions $(\gamma, \alpha)$ where $\gamma$ is true under $I$. The function $F_P(I)$ determines the variables that ‘cannot’ be assigned a value, since for all existing guarded actions $(\gamma, \alpha)$ where $\alpha$ could assign a value to them, the guard $\gamma$ is false under $I$. The causality analysis computes these two sets and then updates the environment by applying the above function $f_P(I)$. Thus, causality analysis of Quartz programs is exactly what is defined by Fitting as the meaning of a logic program.

Let $P$ be an arbitrary logic program, $E$ be its Clark completion, and let $I$ be the least fixpoint obtained as the meaning of $P$. One may wonder whether one can say that $x \in I$ implies that $x$ holds in all two-valued models of $E$, and that $\neg x \in I$ implies that $x$ is false in all two-valued models of $E$, i.e., that the computed fixpoint is the minimal Boolean requirements. However, this is not the case, as shown by the following example:

```
module P4(event x1,x2,x3,x4) {  
if(!x1 & !x2) emit(x1);  
if(x3) emit(x2);  
if(x4) emit(x2);  
if(x3) emit(x3);  
if(x4) emit(x4);  
}
```

Considering Boolean variable assignments, there are the three models $I_1 = \{x2,x3\}$, $I_2 = \{x2,x4\}$, and $I_3 = \{x2,x3,x4\}$. Their ‘agreement’ is $I = \{\neg x1,x2\}$, i.e., all models agree on the values for $x1$ and $x2$. However, the agreement $I$ is not a three-valued model of the equation system. The least fixpoint is the variable assignment where all variables are unknown (and of course that is also computed by the
Quartz simulator for the program on the right hand side). Thus, it is not the case that if a variable has a Boolean value in all two-valued models of the equation system, then it must also have this value in the three-valued fixpoint.

Since the least fixpoint is however the infimum of all other fixpoints, and a three-valued interpretation is a model of the completion of a program if and only if it is a fixpoint of \( f_P \), it follows that \( x \) in contained in the least fixpoint of \( f_P \) holds if and only if all three-valued models \( \mathcal{J} \) contain \( x \) (same for \( \neg x \)). Note that the above equation system is satisfied by assigning all variables the value unknown.

### 1.2.7 Beyond Causal Synchronous Semantics: Well-founded Semantics

Van Gelder, Ross and Schlipf [57] introduced the well-founded semantics of logic programs. To this end, again a three-valued interpretation of variables is used as in the previous section. As above and in [57], 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 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 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. an three-valued interpretation \( \mathcal{I} \) is denoted as \( \mathcal{U}_P(\mathcal{I}) \), and can be computed as a greatest fixpoint as follows: First, we remove all rules from \( P \) 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 \( P \). We now seek the greatest set of variables \( \mathcal{U}_P(\mathcal{I}) \) 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, we start with the set of all variables $V_0$, and successively remove variables $x$ 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.

For example, for $\mathcal{I} = \{\}$ and the above program $P$, we obtain the greatest unfounded set $U_P(\mathcal{I}) = \{x_2, x_4, x_5, x_6\}$ so that these variables could now be made false. Note that we remove $x_1$ and $x_3$ since these variables have a rule without positive subgoals.

For a fixed program $P$, let $U_P(\mathcal{I})$ denote the greatest unfounded set, and let $T_P(\mathcal{I})$ 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 $g_P(\mathcal{I}) := T_P(\mathcal{I}) \cup \{\neg x \mid x \in U_P(\mathcal{I})\}$, 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, we obtain $\mathcal{I}_0 = \{\}, \mathcal{I}_1 = \{x_1, \neg x_2, \neg x_4, \neg x_5, \neg x_6\}$, and $\mathcal{I}_2 = \{x_1, \neg x_2, x_3, \neg x_4, \neg x_5, \neg x_6\}$. Fitting’s fixpoint semantics, however, computes $\mathcal{I}_0 = \{\}$ and $\mathcal{I}_1 = \{x_1, \neg x_6\}$ which is also the result of the Quartz simulator for the corresponding program on the right hand side above.

The two-valued models of the completion are $M_1 = \{x_1, \neg x_2, x_3, \neg x_4, \neg x_5, \neg x_6\}$, $M_2 = \{x_1, \neg x_2, x_3, \neg x_4, \neg x_5, x_6\}$, and $M_3 = \{x_1, x_2, \neg x_3, x_4, x_5, \neg x_6\}$ 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 $U_P(\mathcal{I})$ 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:

For a fixed program $P$, let $U_P(\mathcal{I})$ denote the greatest unfounded set, and let $T_P(\mathcal{I})$ 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 $g_P(\mathcal{I}) := T_P(\mathcal{I}) \cup \{\neg x \mid x \in U_P(\mathcal{I})\}$, 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, we obtain $\mathcal{I}_0 = \{\}, \mathcal{I}_1 = \{x_1, \neg x_2, \neg x_4, \neg x_5, \neg x_6\}$, and $\mathcal{I}_2 = \{x_1, \neg x_2, x_3, \neg x_4, \neg x_5, \neg x_6\}$. Fitting’s fixpoint semantics, however, computes $\mathcal{I}_0 = \{\}$ and $\mathcal{I}_1 = \{x_1, \neg x_6\}$ which is also the result of the Quartz simulator for the corresponding program on the right hand side above.

The two-valued models of the completion are $M_1 = \{x_1, \neg x_2, x_3, \neg x_4, \neg x_5, \neg x_6\}$, $M_2 = \{x_1, \neg x_2, x_3, \neg x_4, \neg x_5, x_6\}$, and $M_3 = \{x_1, x_2, \neg x_3, x_4, x_5, \neg x_6\}$ 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 $U_P(\mathcal{I})$ 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:
The well-founded semantics computes \( I_0 = \{ \} \), \( I_1 = \{ x_1, \neg x_2, \neg x_4, \neg x_5, \neg x_6 \} \), and \( I_2 = \{ x_1, \neg x_2, x_3, \neg x_4, \neg x_5, \neg x_6 \} \) as before, and is therefore not able to determine values for \( x_7 \) and \( x_8 \) that depend on each other via negations. If the negations of the last two rules were omitted, the new variables \( x_7 \) and \( x_8 \) are made false.

Since \( f_P(I) \) is a subset of \( g_P(I) \), 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, we see 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 [57] defined for logic programs. Similar to the existing semantics, a canonical model will be chosen this way that can be computed by fixpoints, 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.

### 1.3 Summary and Future Directions

As shown in the previous section, there is a strong relationship between the semantics of synchronous programs and logic programs. For our discussion, we restricted the programs to Boolean variables only, and furthermore, only considered one macro step where all variables are output variables. In particular, Fitting’s fixpoint semantics for logic programs can be easily seen to be equivalent to the current definition of the causal semantics of synchronous Quartz programs. We have also seen that more powerful semantics known for logic programs like the well-founded semantics of van Gelder, Ross and Schlipf [57] can be defined for synchronous programs so that more synchronous programs can be accepted by the compilers as constructive programs so that deterministic code can be generated also for them.

The list of semantics discussed for logic programs in this paper is by far not complete. In particular, there is the stable model semantics [27] which is based on the definition of a ‘reduct’ of a logic program. Another direction is the concept of stratified and locally stratified programs [37]. These and more alternatives allow to compute models for logical programs which cannot be found with the semantics discussed in this paper.
A recent interest in new ways to define causal semantics for synchronous programs like [60, 61, 59, 2, 45] might be also influenced by the already existing work done for logic programs through the past decades. However, just defining semantics for synchronous programs is not sufficient, it also has to be shown by future work that code can also be efficiently generated for these semantics. We are looking forward to further research in that direction to see the fields of logic and synchronous programs being cross-fertilized in the future.

References

1 Are Synchronous Programs Logic Programs?