On semantics of reactive rule-based systems

Man Lin, Jacek Malec, Simin Nadjm-Tehrani
Email: linma@ida.liu.se, jam@ida.liu.se, snt@ida.liu.se

Abstract

In this paper we address special issues which arise when the rule-based programming paradigm is emlployed in development of reactive real-time systems. We begin by presenting a rule-based language which has emerged while developing intelligent cruise control systems. We describe a naive operational semantics based on an an implemented inference engine, and explain why the semantics is not appropriate in safety-critical real-time contexts. We then define a number of correctness criteria ensuring termination, consistency and determinism, based on a formal declarative semantics. However, the declarative semantics is non-constructive. In order to be able to build an implementation satisfying the correctness criteria, two alternative approaches are proposed. Both approaches build upon static checks of a rule-based program. In the first approach some correctness criteria are met by accepting only *stratified* programs after the static checks. In the second approach we accept programs which are correct with respect to a *constructive* semantics. For both approaches we present the analysis methods and soundness results with respect to the desired declarative semantics.

This research has been supported by Volvo Research Foundation, Volvo Educational Foundation, the Center for Industrial Information Technology (CENIIT), and the Swedish Research Council for Engineering Sciences.

IDA Technical Report 1996
LiTH-IDA-R-96-35
ISSN-0281-4250

Department of Computer
and Information Science,
Linköping University,
S-581 83 Linköping, Sweden

On semantics of reactive rule-based systems*
Draft

Man Lin[†] Jacek Malec[†‡] Simin Nadjm-Tehrani[†]

[†]Department of Computer and Information Science
Linköping University
S–581 83 Linköping, Sweden
(linma,jam,simin)@ida.liu.se
fax: (46)(13)282606

[‡]Department of Computer Engineering
Mälardalens Högskola
Box 883
S–721 23 Västerås, Sweden
jacek.malec@mdh.se

Abstract

In this paper we address special issues which arise when the rule-based programming paradigm is emlployed in development of reactive real-time systems. We begin by presenting a rule-based language which has emerged while developing intelligent cruise control systems. We describe a naive operational semantics based on an an implemented inference engine, and explain why the semantics is not appropriate in safety-critical real-time contexts. We then define a number of correctness criteria ensuring termination, consistency and determinism, based on a formal declarative semantics. However, the declarative semantics is non-constructive. In order to be able to build an implementation satisfying the correctness criteria, two alternative approaches are proposed. Both approaches build upon static checks of a rule-based program. In the first approach some correctness criteria are met by accepting only *stratified* programs after the static checks. In the second approach we accept programs which are correct with respect to a *constructive* semantics. For both approaches we present the analysis methods and soundness results with respect to the desired declarative semantics.

*This research has been supported by Volvo Research Foundation, Volvo Educational Foundation, the Center for Industrial Information Technology (CENIIT), and the Swedish Research Council for Engineering Sciences.

1 Introduction

Humans have a natural tendency to describe dynamic systems using rules. Therefore, rule-based languages are valuable tools for design and prototyping in certain applications, e.g. when a car-industry engineer is to specify the way a driver support system dynamically changes its behaviour as a result of varying road conditions.

In this paper we address a number of issues which are of critical importance when a rule-based language is used for creating a *reactive* program in an embedded system, as opposed to its use while writing a *transformational* program [16].

Real-time rule-based systems have attracted some attention in the last decennium which can be distinguished as two tracks of activity. The first consists of adapting existing rule-based systems used in practical applications to get acceptable responses in the applications under consideration. Usually this leads to a system satisfying soft real-time limitations. The other track focuses on guaranteeing predictable behaviour of a rule-based system, i.e. it concentrates on designing hard real-time systems (see e.g. [23]). This objective forces the rule languages used to be somewhat limited in their expressive power, so that the programs are amenable to formal analysis of their properties.

This paper reports on research done in-between those two approaches. On one hand, the rule-based system under consideration has been created in the context of a particular application [14, 18], and therefore the rule language used, RL [17], has been tested in a realistic setting. On the other hand, we have been working towards providing a formal characterization of behaviour for a system specified (and implemented) using RL. Though it should be pointed out that the characterization is by no means application-dependent and is generally applicable to a large class of reactive rule based systems. This paper is a summary of our attempts made towards this end.

The paper is organized as follows. In section 2 we introduce the rule language RL and describe informally the intended properties of a system based on this language; postponing the formalization of properties until section 5. In section 4 the operational semantics in terms of an implemented inference engine is introduced. The section also presents some undesired properties of rule-based programs which are not adequately dealt with by the naive operational semantics. Section 5 contains a specification of a declarative semantics that fulfills the intended properties mentioned in section 2. It also includes a formalization of our correctness criteria in terms of termination, rule-consistency and slot-consistency. However, the semantics is not too attractive as it lacks a direct operational counterpart. Therefore in sections 6 and 7 we suggest two possible ways of (partially) remedying the situation: introduction of stratified RL programs, and static analysis based on a constructive semantics. Finally, section 8 contains some pointers to related research.

2 Reactive rule-based systems

2.1 A reactive rule-based system

A reactive rule-based system (illustrated in Figure 1) is a system that reacts to the changes of its environment by firing a set of rules [18]. The system is composed of three entities called *state*, *program*, and *inference engine*. The state consists of *slots*: state variables, with associated pairs of values indicating the *previous* and the *current* value of the slot, respectively. During a period when no changes happen (*equilibrium period*, EP), the two values of a slot are identical. At a point when there is a *change* (a stimulus comes from the environment), the current value of some slot becomes updated. We call such a moment an *asynchronous computational point* (ACP). At each ACP, the stimulus triggers one or more rules, what produces new changes in the slots, which in turn trigger other rules, and so on, until no changes are possible, i.e. a steady state is reached. Then the system starts "resting" in its new EP, awaiting new stimuli. The inference engine is in charge of the computations at the ACPs.

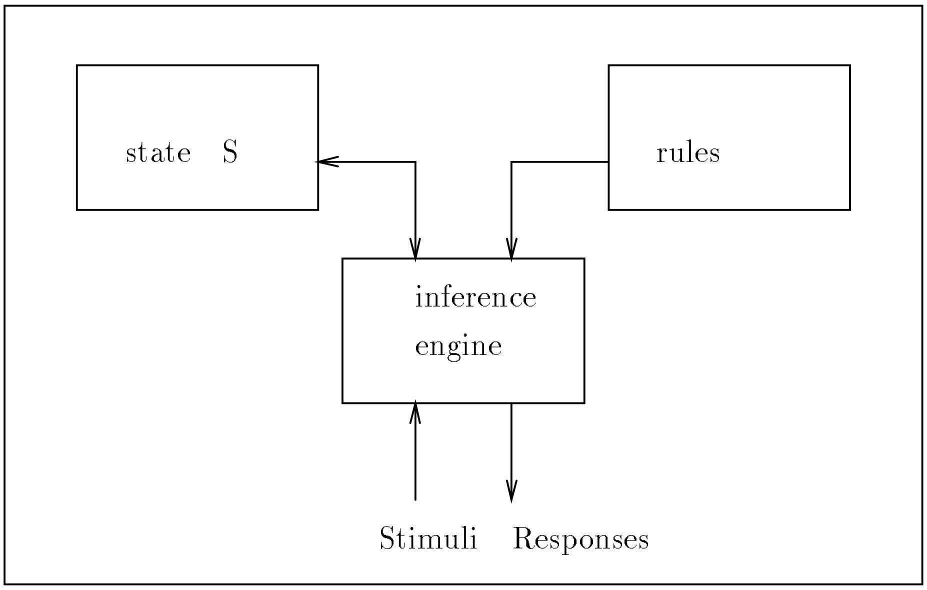


Figure 1: A reactive rule-based system.

A program consists of a sequence of rules that specify the relation between current and subsequent slot values. Speaking informally, the rules express the desired reaction of the system to all the changes that might occur on the inputs. A change (which is encoded by the difference between the previous and the current value of a slot) triggers a rule provided that some conditions are satisfied. This paradigm allows one to easily specify chains of reactions that are triggered by a particular occurance of an input value.

Suppose that a driver support system (our flagship application [14]) is notified that the road has become slippery. The support system has to immediately change its mode of behaviour, warning the driver about this unpleasant road condition, and assuming that from now on, until the road is no longer slippery, the driver's workload rises, leaving no possibility of observing a side display. Therefore, the default channel for communicating messages to the driver has to be changed to voice.

Rule-based systems are very well suited to encoding these kind of dependencies, where each rule encodes a change of state (channel becomes voice, workload becomes high) due to some triggering event (road becomes slippery) and a small number of additional conditions (the voice channel is not off, there is no passenger with whom the driver may be in conversation). A natural modularization of knowledge is possible. The triggering conditions are used to attach a cluster of rules to a particular slot, which has two benefits. First, the structure in the application domain is reflected in the rule set in an "object-oriented" manner - each slot reflecting some changeable aspect of the environment and the rules associated with the change attached to this slot. Secondly, this structure leads to an effective realization of engines for evaluating only the relevant rules at each reaction (those whose triggering slots have just changed).

However, the price to pay for the ease of specification is high: the systems containing hundreds of rules might (and usually do) include conflicts or lead to several conclusions from the same premises. This paper addresses these issues in later sections.

2.2 The syntax of the rule language RL

In this section we present the formal definition of the rule language RL [17]. This language will be the subject of all semantical considerations in the rest of this paper.

Definition 1 *A rule is a string*

$$\texttt{WHEN}\ \langle r_{trigger}\rangle\ \texttt{IF}\ \langle r_{cond}\rangle\ \texttt{THEN}\ \langle r_{assign}\rangle$$

fulfilling the requirements of the following grammar:

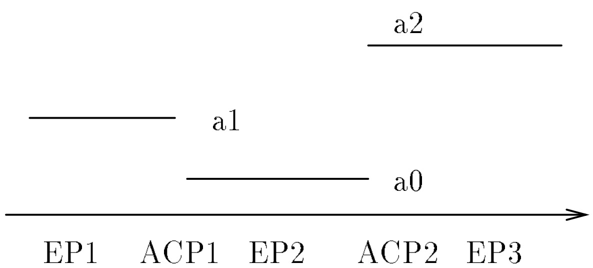
$$\begin{array}{lll}
\langle r_{trigger}\rangle & ::= & \langle \textit{slot-name}\rangle\ \texttt{*=}\ \ \langle \textit{slotvals}\rangle \\
\langle \textit{slotvals}\rangle & ::= & \langle \textit{slotval}\rangle\ \texttt{|}\ \texttt{\{}\ \ \langle \textit{slotval-list}\rangle\ \texttt{\}}\ \texttt{|}\ \texttt{\{*\}} \\
\langle \textit{slotval-list}\rangle & ::= & \langle \textit{slotval}\rangle\ \texttt{|}\ \langle \textit{slotval-list}\rangle\ \ \texttt{,}\ \langle \textit{slotval}\rangle \\
\langle \textit{slotval}\rangle & ::= & \langle \textit{ident}\rangle \\
\langle r_{cond}\rangle & ::= & \langle r_{cond}\rangle\ \texttt{OR}\ \langle r_{cond}\rangle \\
 & & \quad \texttt{|}\ \langle r_{cond}\rangle\ \texttt{AND}\ \ \langle r_{cond}\rangle \\
 & & \quad \texttt{|}\ \texttt{NOT}\ \langle r_{cond}\rangle \\
 & & \quad \texttt{|}\ (\ \langle r_{cond}\rangle\ ) \\
 & & \quad \texttt{|}\ \langle \textit{slot-name}\rangle\ \texttt{*=}\ \ \langle \textit{slotval}\rangle \\
 & & \quad \texttt{|}\ \langle \textit{slot-name}\rangle\ \texttt{|=}\ \ \langle \textit{slotval}\rangle \\
 & & \quad \texttt{|}\ \texttt{TRUE} \\
 & & \quad \texttt{|}\ \texttt{FALSE}
\end{array}$$

<r_{assign}> ::= <*assignment*> | { <*assignment-list*> }
<*assignment-list*> ::= <*assignment*> | <*assignment-list*> , <*assignment*>
<*assignment*> ::= <*slot-name*> := <*slotval*>
<*slot-name*> ::= <*ident*>

where <*ident*> denotes an identifier. □

In the rest of the paper, when referring to the strings appearing in the rules, we shall use the term [s operator v] meaning the denotation of the syntactical element of a rule, i.e. the actual slot and value substituted for the strings s and v appearing in the rule. Sometimes the braces [ ] are skipped, if it cannot lead to misunderstanding.

The two temporal comparison operators used in the rules have the following intended meaning: `*=` is used to denote changes that occurred during the current ACP, i.e. when the slot has acquired a new value due to the current change of the input, whereas `|=` is used to denote the "old" value of a slot, i.e. the one the slot had before the current ACP. Figure 2 illustrates the difference between the two operators. Note that, as the rules are evaluated only during ACPs, the truth value for the operators needs not be defined for an EP. Finally, the assignment operator `:=` is used to denote the operation



	ACP1	ACP2
A `*=` a0	T	F
A `|=` a0	F	T
A `*=` a1	F	F
A `|=` a1	T	F
A `*=` a2	F	T
A `|=` a2	F	F

Figure 2: The temporal comparison operators `*=` and `|=`.

of assigning a new value to a slot.

Definition 2 *A* rule program *P is a tuple $\langle N, D, R, \sqsubseteq_R\rangle$, where $N = \{s_1, \ldots, s_n\}$, $n \in \mathbb{N}$, is a finite set of* slots, *$D = \{D_{s_1}, \ldots, D_{s_n}\}$ is a set of finite* slot domains, *(D_{s_i} being the domain associated with slot s_i), and R is a finite set of rules. $\sqsubseteq_R$ is a linear ordering on R defining the order of the rules in the program.* □

When considering a set X together with an ordering $\sqsubseteq$ defined on it, we shall often use terms "the sequence X" or, to make the order explicit, "the sequence $X_\sqsubseteq$", "first element in X" ($X_\sqsubseteq$, respectively), "next element in X" ($X_\sqsubseteq$), etc. Vice versa, the terms "set X" will denote X *without* taking ordering $\sqsubseteq$ into account. These terms will appear only when it cannot lead to misunderstanding.

In the sequel we will consider cases when the ordering of rules $\sqsubseteq_R$ is not important, or even irrelevant. In such cases we shall speak about a program P being a triple $\langle N, D, R\rangle$.

2.3 The desired properties of the system

We are interested in the system's behaviour at the ACPs. The computation of the system is not terminating (it is reactive), however, we would like the response of the system to its stimuli at each ACP to be *terminating* and *deterministic*. These two properties are considered critical in a real-time system.

Intuitively, a given program is terminating if the response of the system to every stimulus terminates (i.e. its computation at an ACP does not enter an infinite cycle). Furthermore, a given program is deterministic if independently of the order of the rules in the program, the system computes a unique new state for every stimulus at any state.

We will define a program as correct only if it is both terminating and deterministic. The formal definitions will be given in section 5. Obviously not all the syntactically correct programs are (semantically) correct.

The problem we address in this paper is the construction of operational and declarative semantics that are equivalent and which enable the above criteria to be formally expressed and tested (leading to rejection of any semantically incorrect programs).

3 Preliminaries

The following definitions provide basic notions for discussing changes of the state of a program. Let $P = \langle N, D, R, \sqsubseteq_R \rangle$ be a program.

Definition 3 *A* change *is any pair $\langle s, v\rangle$ such that $s \in N$, $v \in D_s$.* □

Definition 4 *An* assignment *$\mathcal{A}_r$ of a rule r is the set of all pairs $\langle s, v\rangle$ occurring in the r_{assign} part of the rule r as [s:=v].* □

Definition 5 *An* ordered assignment *$\mathcal{A}_{r\sqsubseteq}$ of a rule r is the sequence of all pairs $\langle s, v\rangle$ occurring in the r_{assign} part of the rule r as [s:=v], and preserving their order of occurrence in the rule.* □

Definition 6 *Let $C_1 = \langle c^1_1, c^1_2, \ldots, c^1_k\rangle$ and $C_2 = \langle c^2_1, c^2_2, \ldots, c^2_l\rangle$ be two finite sequences of changes. C is a concatenation of C_1 and C_2 (denoted by $C = C_1 \odot C_2$) if and only if $C = \langle c^1_1, c^1_2, \ldots, c^1_k, c^2_1, c^2_2, \ldots, c^2_l\rangle$.* □

Definition 7 *The* system state *(or simply* state*) SS is a pair (S, C), where $S = \langle v_{s_1}, \ldots, v_{s_n}\rangle$ is the sequence of values of all slots, $s_i \in N, n = |N|$ (the number of slots) and $C = \{c_1, c_2, \ldots, c_m\}$ is a set of changes such that*

$$\forall c((c = \langle x, v\rangle \wedge c \in C) \rightarrow x \in N \wedge v \in D_x \wedge v \neq v_x).$$

□

One can envision S being a list of values of slots during the most recent EP, whereas C containing the changes made (so far) during the current ACP.

A rule-based system receives external stimuli from its environment and responds to them by changing values of some slots, according to the laws defined by the rules.

Definition 8 *A* stimulus *I is a finite set of changes. An ordered stimulus to a program P is the sequence $I_{\sqsubseteq_I}$ induced by some linear order $\sqsubseteq_I$ on I.* □

4 Operational semantics: the inference engine

In this section we provide a formal account of an implemented engine for executing RL programs, reported earlier in [18]. This engine was used for implementation of a prototype driver support system within the European Prometheus and the Swedish RTI programs [14]. Thus, this section provides the operational semantics of a RL program when run with this engine.

4.1 The execution mechanism

The basic execution mechanism of the engine is the *firing* of relevant rules. The notion of firing a rule r is intuitively understood as checking its trigger r_{trig} and condition r_{cond}, and executing its assignments $\mathcal{A}_{r_{\sqsubseteq}}$. These auxiliary notions are formally defined as follows.

Definition 9 *Let r be a rule. Rule r is* triggered *in a state (S, C) by a change $z \in C$, denoted by $(S, C) \vdash_t^z r$, if and only if one of the following conditions holds:*

1. $r_{trig} = [x\texttt{*=}v]$, *$v \in D_x$ and $z = \langle x, v\rangle$;*
2. $r_{trig} = [x\texttt{*=}v]$, *$v = v_1, \ldots, v_m$ (i.e. v is a list of slot values), for all i, $v_i \in D_x$ and for some j, $z = \langle x, v_j\rangle$, where $i, j \in \{1, \ldots, m\}$;*
3. $r_{trig} = [x\texttt{*=}\{*\}]$, *and $z = \langle x, v\rangle$, where $v \in D_x$.* □

Definition 10 *Let r be a rule. Rule r is* triggered *in a state (S, C), denoted by $(S, C) \vdash_t r$, if and only if $\exists z \in C((S, C) \vdash_t^z r)$.* □

Definition 11 *Let r be a rule. Rule r's condition is satisfied in a state (S, C), denoted by $(S, C) \vdash_c r$, if and only if one of the following conditions holds:*

1. $r_{cond} = \texttt{TRUE}$;
2. $r_{cond} = [x* = v]$, $v \in D_x$, and $\langle x, v\rangle \in C$;
3. $r_{cond} = [x| = v]$, $v \in D_x$, and $v_x = v$;
4. $r_{cond} = r_{cond1}\ \texttt{AND}\ r_{cond2}$, and $(S, C) \vdash_c r_{cond1}$ and $(S, C) \vdash_c r_{cond2}$;
5. $r_{cond} = r_{cond1}\ \texttt{OR}\ r_{cond2}$ and $(S, C) \vdash_c r_{cond1}$ or $(S, C) \vdash_c r_{cond2}$;
6. $r_{cond} = \texttt{NOT}\ r_{cond1}$ and it is not the case that $(S, C) \vdash_c r_{cond1}$. □

Definition 12 *A rule r is* enabled *in a state (S, C), $(S, C) \vdash r$, if and only if $(S, C) \vdash_t r$ and $(S, C) \vdash_c r$.* □

A rule r is fired as a result of a change c in a given state, resulting in a sequence of changes defined by $\mathcal{A}_{r\sqsubseteq}$. The following definition captures this notion.

Definition 13 *Let (S, C) be a state of the program $P = \langle N, D, R, \sqsubseteq_R \rangle$, $\sqsubseteq_C$ be a linear order on C, and let $z \in C$. The function* `fire` *yields for any sequence of rules $\langle r_1, \ldots, r_m \rangle$ (i.e. $R_{\sqsubseteq_R}$) and any $S, C_{\sqsubseteq_C}, z$, a sequence of changes:*

$$\mathtt{fire}(\langle r_1, \ldots, r_m \rangle, S, C_{\sqsubseteq_C}, z) =$$

$$= \begin{cases} C_{\sqsubseteq_C} & \text{if } \langle r_1, \ldots, r_m \rangle = \langle \rangle \\ \mathtt{fire}(\langle r_2, \ldots, r_m \rangle, S, C_{\sqsubseteq_C} \odot \mathcal{A}_{r_1 \sqsubseteq}, z) & \text{if } (S, C) \vdash_c r_1 \text{ and } (S, C) \vdash_t^z r_1 \\ \mathtt{fire}(\langle r_2, \ldots, r_m \rangle, S, C_{\sqsubseteq_C}, z) & \text{otherwise} \end{cases}$$

□

Informally speaking, there is a list $C_{\sqsubseteq_C}$ which stores *changes* (initially it contains just I), and there is a pointer which points to the current trigger (a *change*). The system fires the enabled rules triggered by the current trigger, in the order of their occurrence in the program, and adds the new changes to the list $C_{\sqsubseteq_C}$ whenever firing a rule. When there are no more enabled rules triggered by the current trigger, the pointer moves to the next change in the list and the system continues firing rules and adding new changes to the list, until the pointer is in the end of the list and there are no more changes to be added to the list. The next definition formally describes this procedure.

Definition 14 *The* operational response *of the program $P = \langle N, D, R, \sqsubseteq_R \rangle$ in a state $(S, \emptyset)$ to a stimulus $I_{\sqsubseteq_I}$ is a sequence:*

$$\phi_1 p_1 \phi_2 p_2 \ldots$$

such that

- $\phi_1 = (S, C_1)$, *where* $C_1 = I_{\sqsubseteq_I}$,
- $p_i \in \mathbb{N}$,
- $p_1 = 1$,
- $\phi_{i+1} = (S, \mathtt{fire}(R, S, C_i, z))$, *where* $C_i = \langle c_1^i, c_2^i, \ldots, c_m^i \rangle$ *and* $z = c_{p_i}^i$,
- $p_{i+1} = \begin{cases} p_i & \text{if and only if } p_i = |C_i| \\ p_i + 1 & \text{otherwise.} \end{cases}$ □

Definition 15 *The operational response of a program P to a stimulus $I_{\sqsubseteq_I}$* is terminating *if and only if*

$$\phi_k = \phi_{k+1} \text{ and } p_k = p_{k+1}$$

for some k. □

Thus, a system with terminating responses achieves a stable state (an EP) after reacting to any change in the environment. Having come to a rest in a state (S, C) after a response, the state before the reaction to the next stimulus can be computed as $(S', \emptyset)$ where $S' = \langle v'_{s_1}, \ldots, v'_{s_n} \rangle$ and

$$v'_{s_i} \stackrel{df}{=} \begin{cases} v & \text{if and only if } \langle s_i, v \rangle \in C \\ v_{s_i} & \text{otherwise} \end{cases}$$

4.2 Is this semantics satisfactory?

There are two major problems with the operational semantics defined above. One is the lack of a guarantee that the responses are always terminating, and the other is the possibility of multiple outcomes for a given state and a stimulus, due to the sensitivity to rule ordering.

Since the number of slots is finite, loops in the response are possible in two cases: if one or more slots are assigned a new value more than once, or when a slot is assigned the same value more than once. Though there may be a long sequence of firings forming such a loop, the following example illustrates a simple case. The response of Program 1 in the state $(\langle a2, b2, c2, d2 \rangle, \emptyset)$ to a stimulus $I = \langle A, a0 \rangle$ will run into an infinite sequence of rule firings.

Program 1

```
r1:   WHEN A *= a0 IF true THEN B := b0;
r2:   WHEN B *= b0 IF true THEN C := c1;
r3:   WHEN C *= c1 IF true THEN D := d1;
r4:   WHEN D *= d1 IF true THEN A := a0;
```

A sufficient condition for avoiding non-termination is to restrict the number of value assignments for each slot during every response. That is, a slot can be allowed to be assigned a value at most once during every response. However, a simple runtime detection of this condition is not satisfactory from the predictability point of view and we have to identify and reject such possibilities statically (as described in the section 7).

The next example illustrates the sensitivity of the response of the implemented engine to the ordering $\sqsubseteq_R$ of the rules:

Program 2

```
r1:   WHEN A *= a1 IF NOT B *= b1 then C := c1;
r2:   WHEN A *= a1 IF B |= b0 then B := b1;
```

Program 3

```
r2:   WHEN A *= a1 IF B |= b0 then B := b1;
r1:   WHEN A *= a1 IF NOT B *= b1 then C := c1;
```

Note that both programs use the same set of rules; the only difference between them is the ordering of the rules ($\sqsubseteq_2 = \{(\mathtt{r1}, \mathtt{r2})\}$, $\sqsubseteq_3 = \{(\mathtt{r2}, \mathtt{r1})\}$). Given the same initial state, and the same stimulus, the implemented engine would not produce the same operational response for the two programs. Suppose the system state is $SS = (\langle a0, b0, c0\rangle, \emptyset)$ and the external stimuli is $\langle (A, a1)\rangle$. We will get two different final states in responses of the two programs:

$$\begin{array}{rcl} (S, C_1) & = & (\langle a0, b0, c0\rangle, \{\langle A, a1\rangle, \langle B, b1\rangle, \langle C, c1\rangle\}) \\ (S, C_2) & = & (\langle a0, b0, c0\rangle, \{\langle A, a1\rangle, \langle B, b1\rangle\}) \end{array}$$

Thus, the final state depends on the rule ordering in the program. This definitely does not correspond to our intuition about the declarative interpretation of a rule-based program. Furthermore, since the fired rule `r1` has a condition which is not satisfied in the final state, we might ask whether this was the actual intention underlying this particular program.

Obviously the rule ordering can also be exploited by the programmer to obtain one of the multiple possible responses to a stimulus. Also, in analogy with logic programs (without negation), there are many cases when the implemented engine produces single responses. However, for large knowledge bases and more complex applications some support for the programmer is essential.

5 Desired declarative semantics

Since a rule-based language should ideally be used to capture the knowledge about an application declaratively, we now provide an intended declarative semantics of a program P. The defined notions are further used to identify the programs which are considered well-defined in a semantical sense.

5.1 The declarative response

Definition 16 *Suppose R is the set of rules of a program P. The* declarative response *of the program P in a state $(S, \emptyset)$ to a stimulus I is any sequence of firings*

$$\sigma_0 \sigma_1 \ldots$$

such that

- $\sigma_0 = (C_0, R_0^f) \stackrel{df}{=} (I, \emptyset)$,
- $\sigma_{i+1} = (C_{i+1}, R_{i+1}^f) \stackrel{df}{=}$

$$= \begin{cases} (C_i \cup \mathcal{A}_{r_f}, R_i^f \cup \{r_f\}) & \textit{where } r_f \in \overline{R} = \{r | r \in R \setminus R_i^f \wedge (S, C_i) \vdash r\} \\ & \textit{if } \overline{R} \neq \emptyset \\ \sigma_i & \textit{if } \overline{R} = \emptyset \end{cases}$$

□

In the definition above R_i^f is used to denote the set of rules that have been fired up to point i in the sequence of firings.

Corollary 1 *Suppose R is the set of rules of a program P. For every declarative response*

$$\sigma_0\sigma_1\dots$$

of the program P in a state $(S,\emptyset)$ to a stimulus I there exists some $j \in \mathbb{N}$ such that $\sigma_j = \sigma_{j+1}$. □

Proof: Each step from σ_i to σ_{i+1} either adds a (fired) rule to R^f or leaves it intact. As R is finite, we have a finite sequence (of length at most $|R|$) of steps. □

The immediate consequence of this result is that

$$\forall k > j(\sigma_k = \sigma_{k+1})$$

Therefore we can speak about declarative responses being just finite sequences of firings understanding it to mean infinite sequences according to the definition above, but with the trailing stuttering cases omitted.

Obviously, the declarative response of a program to a stimulus need not be unique. Different choices of r_f at each step can lead to different firing sequences.

It should be noted that the C_i component in each σ_i in the above definition of declarative response could have been computed using an operator, the minimal fixed point(s) of which would give the final C_m. We have adopted the above definition so that our notion of correctness can refer to the fired rule set R_f as well as the state after the reaction. In the following definition we provide three additional notions which capture our well-definedness criteria.

Definition 17 *Let P be a program $\langle N, D, R\rangle$. Let a declarative response of the program in a state $(S,\emptyset)$ to a stimulus I be $\sigma_0\sigma_1\dots\sigma_m$. Let $\sigma_m = (C_m, R^f)$ and $R^f = \{r_1, r_2, \dots, r_m\}$. The declarative response is* correct *if and only if*

- *the response is* slot consistent*:*

$$\forall x(\langle x, v_1\rangle \in C_m \wedge \langle x, v_2\rangle \in C_m \rightarrow v_1 = v_2)$$

  that is, no slot can have more than one change of value in this response;

- *the response is* rule consistent*:*

$$\forall r(r \in R^f \rightarrow (S, C_m) \vdash r)$$

  that is, none of the rules fired in this response will become disabled after the final firing;

- *the response is* unambiguous*: for any other declarative response $\sigma_0\sigma_1'\dots\sigma_k'$ that is both rule consistent and slot consistent, we have $C_k' = C_m$, where $\sigma_k' = (C_k', R'^f)$.* □

The program is *correct* if and only if for any state $(S,\emptyset)$ and any stimulus I, the program has a correct declarative response.

5.2 Properties of the declarative semantics

The following results exploit the iterative construction of the fired rule set.

Proposition 1 *Let P be a program $\langle N, D, R\rangle$. Let a declarative response of the program in a state $(S, \emptyset)$ to a stimulus I be $\sigma_0\sigma_1\ldots\sigma_m$, and let $\sigma_m = (C_m, R^f)$. Then*

$$\forall r(r \in R \setminus R^f \rightarrow \neg((S, C_m) \vdash r))$$

□

Proposition 2 *Given a state $(S, \emptyset)$ and a stimulus I, if all the rule consistent responses are unambiguous, then the sets of fired rules, R^f, for all such responses are equal.* □

Proof: Let $DR_1 = \sigma_1, \ldots, \sigma_{m_1}$, $DR_2 = \sigma_1, \ldots, \sigma_{m_2}$ be any two *rule consistent* responses of the program P in a state $(S, \emptyset)$ to a stimulus I. By the definition of rule consistency, we know that $\forall r \in R^f_{m_1}((S, C_{m_1}) \vdash r)$. Then $(S, C_{m_2}) \vdash r$ since $C_{m_1} = C_{m_2}$ (unambiguity). Hence, $r \in R^f_{m_2}$, otherwise the response would not have terminated at $\sigma_{m_2} = (C_{m_2}, R^f_{m_2})$. By the symmetrical argument we get $\forall r(r \in R^f_{m_2} \rightarrow r \in R^f_{m_1})$. □

Hence R^f is unique for correct responses.

5.3 The examples revisited

Having defined our notion of correctness in terms of declarative semantics, we will now return to Programs 2 and 3 and study their declarative response for the same state and stimulus. The programs have the following two declarative responses:

$$\begin{aligned} DR_2 &= (\{\langle A, 1\rangle\}, \emptyset),\ \ (\{\langle A, a1\rangle, \langle C, c1\rangle\}, \{\mathtt{r1}\}),\\ &\quad\ (\{\langle A, a1\rangle, \langle C, c1\rangle, \langle B, b1\rangle\}, \{\mathtt{r1}, \mathtt{r2}\})\\ DR_3 &= (\{\langle A, a1\rangle\}, \emptyset),\ \ (\{\langle A, a1\rangle, \langle B, b1\rangle\}, \{\mathtt{r2}\}) \end{aligned}$$

Because it is not the case that $(S, \{\langle A, a1\rangle, \langle C, c1\rangle, \langle B, b1\rangle\}) \vdash \mathtt{r1}$, DR_2 is not a rule consistent response, and therefore it is not correct. DR_3 is a rule consistent and slot consistent response and hence the unique correct response.

6 Stratified rule-based programs

Below we give two alternative approaches for (partially) implementing the declarative semantics. First, we present a syntactic approach by checking every rule-based program for being *stratified*. We propose an engine to implement the declarative semantics and show that this engine gives a better operational semantics for this class of rule-based programs. That is, the programs run under this engine are at least guaranteed to be rule consistent. However, the slot consistency is not guaranteed. This corresponds to the approach we adopted in practice when developing prototypes for our support systems. Later in section 7 we adopt an operational semantics based on a three-valued semantics and discuss its correspondence to the declarative semantics.

In this section we use the idea of stratification in logic programs with negation [7, 1] to achieve rule consistency in reactive rule-based systems. In the case of stratified logic programs the predicate letters are divided into disjoint sets or *strata* with a linear ordering among the strata. This will induce certain orderings among the clauses in a logic program. In rule-based programs we apply a similar method to divide the rule set into disjoint sets. The ordering between the strata is then used to define the ordering $\sqsubseteq_R$ in the rule set such that lower strata appear before higher strata, but the order of the rules in each stratum is arbitrary.

The crucial point is the way in which strata are determined. The procedure described below will determine whether stratification of a rule set is possible. If the rule set cannot be stratified it will be rejected at compile time. Otherwise, the responses derived by the implemented engine while firing the rules in the stratified order will always lead to rule consistent final states. That is, working with stratified rule sets has the following effect: every time a rule which has a condition part including negation over $[s\texttt{*=}v]$ is tested for being enabled, we can be sure that a rule with an assignment to s has been fired earlier in the response (if it is included in the final fired rule set of this response at all)[1].

A stratified rule-based program consists of a disjoint set of rules $R = R^1 \cup \ldots \cup R^i \cup \ldots \cup R^k$ called *strata*. We can decide whether a rule-based program P is stratifiable by analyzing the *Extended Dependency Graph* (EDG) of P. First, those rules that have more than one assignment in the r_{assign} parts are split into several rules so that all rules have only one assignment in the r_{assign} part. The EDG of P is then constructed as follows: The nodes of EDG(P) consist of all slots in N. A directed edge $\langle p, q\rangle$ from node p to node q indicates that a value is assigned to the slot p in the r_{assign} part of a rule whose r_{trig} or r_{cond} part test the change of value for the slot q, that is, $q\texttt{*=}v$ or `NOT` $q\texttt{*=}v$, where $v \in D_q$. If the test involves `NOT` $q\texttt{*=}v$, then the edge is marked with "$\neg$".

A rule-based program P is stratifiable if and only if EDG of P contains no cycles involving edges marked with "$\neg$". If a program is stratifiable, its stratification is constructed (in analogy with [22]) as follows:

- If a slot p occurs in the r_{assign} part of a rule whose r_{trig} or r_{cond} part include a non-negated occurrence of $q\texttt{*=}v$ (for any v), then the stratum of p is at least at high as the stratum of q.

- If a slot p occurs in the r_{assign} part of a rule whose r_{cond} part includes an occurrence of `NOT` $q\texttt{*=}v$ (for any v), then q is in a lower stratum than p.

[1] Note that negation over $s\texttt{|=}v$ is not critical since this information is always known during an ACP.

- The stratification of the rules follows from the stratification of slots since (after splitting) every rule assigns to one slot only.

In what follows we illustrate the application of the procedure to two programs. Figures 3 and 4 show the EDGs of Program 2 presented earlier and Program 4 below, respectively. Program 2 is stratified, but Program 4 is not.

Program 4

```
rule1:   WHEN B *= b1  IF  A |= a1 then  A := a2;
rule2:   WHEN B *= b1  IF NOT A *= a1 then  C := c2;
rule3:   WHEN C *= c1  IF  A |= a1  then  B := b2;
```

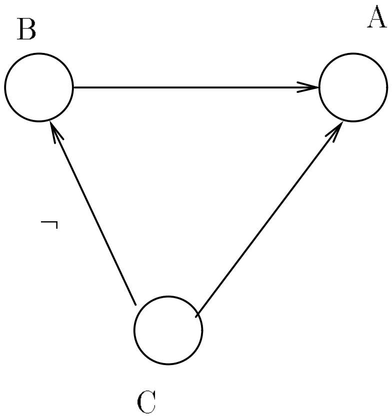


Figure 3: The EDG for Program 2

Now we can construct an s-engine (for "stratification-engine") which fires rules in the strata order.

Let P be a program $\langle N, D, R\rangle$, and $R^1, R^2 \ldots R^k$ be a stratification of P. A response of the program in a state $(S, \emptyset)$ to a stimulus I is a sequence of firings: $\sigma^1, \sigma^2, \ldots \sigma^k$ where $\sigma^j = \sigma^j_0\sigma^j_1 \ldots \sigma^j_{m_j}$ such that $\sigma^j_{m_j-1} = \sigma^j_{m_j}$

- $\sigma^1_0 = (C^1_0, R^{f,1}_0) \stackrel{df}{=} (I, \emptyset)$,
- $\sigma^{j+1}_0 = (C^{j+1}_0, R^{f,j+1}_0) \stackrel{df}{=} \sigma^j_{m_j}$,
- $\sigma^j_{i+1} = (C^j_{i+1}, R^{f,j}_{i+1}) \stackrel{df}{=}$

$$\begin{cases} (C^j_i \cup \mathcal{A}_{r_f}, R^{f,j}_i \cup \{r_f\}) & \text{where } r_f \in \overline{R}^j = \{r | r \in R^j \setminus R^{f,j}_i \wedge (S, C^j_i) \vdash r\} \\ & \quad \text{if } \overline{R}^j \neq \emptyset, \\ \sigma^j_i & \text{if } \overline{R}^j = \emptyset \end{cases}$$

$\square$

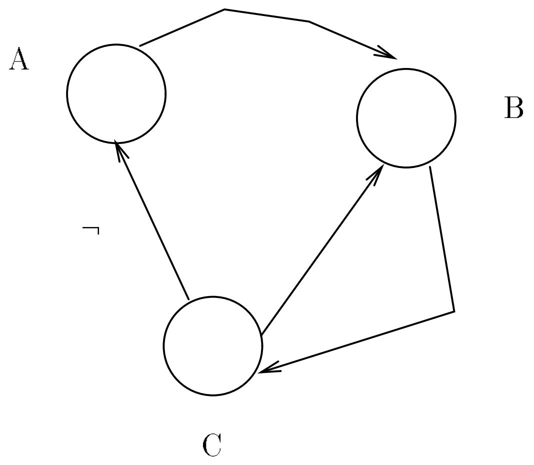


Figure 4: The EDG for Program 4

Lemma 1 *Given an stratified program P, the responses of the program produced by the s-engine in any state $(S, \emptyset)$ to any stimulus I is rule-consistent.* □

Proof: By the defintion of stratification and construction of the s-engine. □

Proposition 3 *Given any correct program P, if P can be stratified, then irrespective of the stratification of P, any slot-consistent response generated by the s-engine is a correct response.* □

Proof: By reduction to lemma 1 and by the definitions of a correct response and of a correct program. □

From the above proposition, we are sure that a response is correct if it is a slot consistent response produced by the implemented engine running on a stratified correct program. Unfortunately, for a stratified program the operational semantics does not guarantee slot consistency. Given a program, it is not possible to say that it is a correct program just by testing whether it is stratified.

Furthermore, stratified programs are too restrictive a class as far as applications are concerned. For example, Program 4 is actually acceptable according to our desired semantics, but it is rejected since it is not stratified. In the following section we define a semantical approach which rejects fewer programs (and accepts Program 4 among others).

7 Constructive semantics

In what follows we present an alternative approach to the analysis of RL programs. The idea consists of checking whether a program P fulfills a set of logical-correctness criteria during compile time and then executing correct programs using an efficient, simplified inference engine implementing the constructive semantics.

7.1 The semantics

Our constructive semantics is an application of the three-valued logic approach to non-monotonic reasoning in the setting of reactive systems. It also resembles the recent proposal of constructive semantics for pure Esterel [3]. The main differences are in the structure of programs (rule-based in our case, imperative in the case of Esterel), and the means of communication (change in slot values in our case, pure signals/events in Esterel).

The main difference from earlier semantics presented in this paper is that instead of dealing with a set (or sequence) of changes we deal with annotated sets of slot changes. The potential slot changes are annotated with a *status*. The status is an element from the set $\{+, -, \perp\}$, where $\langle x, v\rangle^{\perp}$ means that the status of the slot change $\langle x, v\rangle$ is unknown; $\langle x, v\rangle^{+}$ means that the value of slot x does change to v at this ACP; $\langle x, v\rangle^{-}$ means that the slot x definitely cannot change to v at this ACP.

Definition 18 *An* extended system state *is a tuple (S, Z) where S is a tuple $\langle v_1, \ldots, v_n\rangle$ of all slot values, and $Z = \{\langle x, v\rangle^{a} | x \in N \wedge v \in D_x\}$ such that $a \in \{+, -, \perp\}$ and $\forall\langle x, v\rangle((\langle x, v\rangle^{a} \in Z \wedge \langle x, v\rangle^{a'} \in Z) \rightarrow a = a')$.* □

We define the following partial order on the status markings:

$$\preceq = \{(\perp, -), (\perp, +), (\perp, \perp), (-, -), (+, +)\}.$$

Definition 19 *Let X be a set of changes.*

$$\mathtt{ext}(X) = \{\langle x, v\rangle^{+} \mid \langle x, v\rangle \in X\} \cup \{\langle x, v'\rangle^{-} \mid \langle x, v\rangle \in X \wedge v' \in D_x \wedge v' \neq v\}$$

is the set of annotated changes *derived from X.* □

Definition 20 *Let Z, Z' be sets of annotated changes. Z' is* more informative *than Z, denoted $Z \preceq Z'$ if and only if*

$$(\forall\langle x, v\rangle^{a} \in Z)(\exists a')(\langle x, v\rangle^{a'} \in Z' \wedge a \preceq a').$$

□

By Z^{-} we denote the set of changes derived from a set of annotated changes Z:

$$Z^{-} \stackrel{df}{=} \{\langle x, v\rangle \mid \langle x, v\rangle^{+} \in Z\}.$$

Corollary 2 *If $Z \preceq Z'$ and $Z' \preceq Z$, then $Z = Z'$.* □

Proof: Let $\langle x, v\rangle^{a} \in Z$. By $Z \preceq Z'$, $(\exists a')(\langle x, v\rangle^{a'} \in Z' \wedge a \preceq a')$. By $Z' \preceq Z$, $(\exists a'')(\langle x, v\rangle^{a''} \in Z \wedge a' \preceq a'')$. Therefore, $\forall\langle x, v\rangle\forall a\exists a'' \wedge \langle x, v\rangle^{a} \in Z \wedge \langle x, v\rangle^{a''} \in Z \wedge a \preceq a''$. By the Definition 18, $a = a' = a''$. Thus, $Z = Z'$. □

In the next step we introduce the three-valued evaluation of rules.

Definition 21 *A rule r is* triggered *in an extended system state (S, Z) by a change $z \in Z$, denoted by $(S, Z) \vdash_{3t}^{z} r$, if and only if one of the following conditions holds:*

1. $r_{trig} = [x\texttt{*=}v]$, $v \in D_x$ *and* $z = \langle x, v \rangle^+$;
2. $r_{trig} = [x\texttt{*=}v]$, $v = v_1, \ldots, v_m$, *where for all* i, $v_i \in D_x$ *and for some* j $z = \langle x, v_j \rangle^+$, *and* $i, j \in \{1, \ldots, m\}$;
3. $r_{trig} = [x\texttt{*=}\{*\}]$, *and* $\exists v \in D_x(z = \langle x, v \rangle^+)$. □

Definition 22 *Let r be a rule, (S, Z) be an extended state. Rule r is* triggered *in the state (S, Z), denoted by $(S, Z) \vdash_{3t} r$ if and only if $\exists z \in Z((S, Z) \vdash_{3t}^{z} r)$.* □

Definition 23 *A rule r is* blocked *in an extended system state (S, Z), denoted by $(S, Z) \not\vdash_{3t} r$, if and only if one of the following conditions holds:*

1. $r_{trig} = [x\texttt{*=}v]$, $v \in D_x$ *and* $\langle x, v \rangle^- \in Z$,
2. $r_{trig} = [x\texttt{*=}v]$, $v = v_1, \ldots, v_m$, *where for all* i, $v_i \in D_x$ *and* $\langle x, v_i \rangle^- \in Z$, *and* $i \in \{1, \ldots, m\}$;
3. *If* $r_{trig} = [x\texttt{*=}\{*\}]$, *and* $\forall v \in D_x(\langle x, v \rangle^- \in Z)$; □

Definition 24 *A rule r's condition is satisfied in a state (S, Z), denoted by $(S, Z) \vdash_{3c} r$ if and only if one of the following conditions holds:*

1. $r_{cond} = \texttt{TRUE}$;
2. $r_{cond} = [x* = v]$, $v \in D_x$, and $\langle x, v \rangle^+ \in Z$;
3. $r_{cond} = [\texttt{NOT } x* = v]$, $v \in D_x$, and $\langle x, v \rangle^- \in Z$;
4. $r_{cond} = [x| = v]$, $v \in D_x$, and $V_x = v$.
5. $r_{cond} = r_{cond1}$ `AND` r_{cond2} and $(S, Z) \vdash_{3c} r_{cond1}$ and $(S, Z) \vdash_{3c} r_{cond2}$;
6. $r_{cond} = r_{cond1}$ `OR` r_{cond2} and $(S, Z) \vdash_{3c} r_{cond1}$ or $(S, Z) \vdash_{3c} r_{cond2}$;
7. $r_{cond} = \texttt{NOT } r_{cond1}$ and $(S, Z) \not\vdash_{3c} r_{cond1}$ □

Definition 25 *A rule r's condition is non-satisfied in a state (S, Z), denoted by $(S, Z) \not\vdash_{3c} r$ if and only if one of the following conditions holds:*

1. $r_{cond} = \texttt{FALSE}$;
2. $r_{cond} = [x* = v]$, $v \in D_x$, and $\langle x, v \rangle^- \in Z$;
3. $r_{cond} = [\texttt{NOT } x* = v]$, $v \in D_x$, and $\langle x, v \rangle^+ \in Z$;
4. $r_{cond} = [x| = v]$, $v \in D_x$, and $V_x \neq v$.
5. $r_{cond} = r_{cond1}$ `AND` r_{cond2} and $(S, Z) \not\vdash_{3c} r_{cond1}$ or $(S, Z) \not\vdash_{3c} r_{cond2}$;

6. $r_{cond} = r_{cond1}$ `OR` r_{cond2} and $(S, Z) \not\vdash_{3c} r_{cond1}$ and $(S, Z) \not\vdash_{3c} r_{cond2}$*;*

7. $r_{cond} =$ `NOT` r_{cond1} and $(S, Z) \vdash_{3c} r_{cond1}$ □

We use $(S, Z) \vdash_3 r$ for $((S, Z) \vdash_{3c} r) \wedge ((S, Z) \vdash_{3t} r)$. We say $(S, Z) \not\vdash_3 r$ instead of $((S, Z) \not\vdash_{3c} r) \vee ((S, Z) \not\vdash_{3t} r)$. It should be observed that there are intermediate cases when neither $(S, Z) \vdash_3 r$ nor $(S, Z) \not\vdash_3 r$ is true.

Corollary 3 *Let (S, Z) be an extended state. Let (S, C) be the system state, with $C = Z^-$. For any $C' \supseteq C$ such that $\forall x(\langle x, v\rangle \in C' \wedge \langle x, v'\rangle \in C' \to v = v')$ and $\forall\langle x, v\rangle(\langle x, v\rangle^- \in Z \to \langle x, v\rangle \notin C')$, we have*

$$\forall r((S, Z) \vdash_3 r \to (S, C') \vdash r), \tag{1}$$

$$\forall r((S, Z) \not\vdash_3 r \to (S, C') \not\vdash r). \tag{2}$$

□

Corollary 4 *Let (S, Z) be an extended state. Let (S, C) be the system state, with $C = Z^-$. If Z contains the complete information about each possible change, that is, $(\forall\langle x, v\rangle^a \in Z)(a \neq \bot)$, then*

$$\forall r((S, Z) \vdash_3 r \leftrightarrow (S, C) \vdash r),$$

$$\forall r((S, Z) \not\vdash_3 r \leftrightarrow (S, C) \not\vdash r).$$

□

Corollary 5 *Given two extended states (S, Z) and (S, Z') such that $Z \preceq Z'$, the following holds*

$$\forall r((S, Z) \vdash_3 r \to (S, Z') \vdash_3 r),$$

$$\forall r((S, Z) \not\vdash_3 r \to (S, Z') \not\vdash_3 r).$$

□

In the sequel we will speak about those properties as monotonicity of $\vdash_3$ and $\not\vdash_3$.

Next we define relative consistency of two sets of annotated changes, and the way of merging two consistent annotated changes together.

Definition 26 *Two sets of annotated changes Z and U are* slot consistent, *denoted by slot-cons(Z, U), if and only if the following two conditions hold:*

$$\forall\langle x, v\rangle\ (\langle x, v\rangle^+ \in Z \to (\langle x, v\rangle^- \notin U \wedge (\forall v' \in D_x)(v' \neq v \to \langle x, v'\rangle^+ \notin U)))$$

$$\forall\langle x, v\rangle(\langle x, v\rangle^- \in Z \to \langle x, v\rangle^+ \notin U)$$

□

The next three definitions are the core of the constructive semantics.

Definition 27 *The function* `update` *takes two sets of annotated changes Z and U, and updates Z with respect to U as follows:*

$\mathtt{update}(Z, U) \stackrel{df}{=}$

$$= \begin{cases} \{\langle x, v\rangle^a \mid \langle x, v\rangle^a \in Z \wedge (\langle x, v\rangle^{a'} \in U \rightarrow a = a')\}\cup \\ \cup\{\langle x, v\rangle^a \mid \langle x, v\rangle^a \in U \wedge (\langle x, v\rangle^{a'} \in Z \rightarrow a' = \perp)\} & \textit{if slot-cons}(Z, U), \\ \emptyset & \textit{otherwise.} \end{cases}$$

□

Note that an empty set is returned as a result when the set U which includes the updating information is not slot consistent with Z.

The function `never`, defined below, increases the negative information in an extended state by adding pairs annotated with "$-$", if the current extended state dictates that certain rules in the program cannot be enabled in a given response.

Definition 28 *The function* `never` *yields for any extended state $\langle S, Z\rangle$ and a set of rules R, a new extended state:*

$\mathtt{never}(S, Z, R) \stackrel{df}{=}$

$$= \begin{cases} \emptyset & \textit{if } Z = \emptyset \\ Z & \textit{if } Neg = \emptyset, \\ \mathtt{never}(S, \mathtt{update}(Z, Neg), R) & \textit{otherwise,} \end{cases}$$

where $Neg = \{\langle x, v\rangle^- \mid \langle x, v\rangle^\perp \in Z \wedge \forall r \in R(\langle x, v\rangle \in \mathcal{A}_r \rightarrow \langle S, Z\rangle \not\vdash_3 r)\}$. □

Obviously, `never` is well defined since after some finite number of recursion steps, either Z becomes $\emptyset$ due to slot-inconsistency, or Z becomes stable since the corresponding Neg is $\emptyset$.

Finally, we define the notion of a constructive response. Note that we follow a pattern similar to the one used for declarative semantics except for updating the extended state. In each step we add the positive information coming from firing a rule, and then we add the negative information resulting from the (positive) update.

Definition 29 *Given a rule-based program $P = \langle N, D, R\rangle$, an initial system state $(S, \emptyset)$, and a stimulus I, the* constructive response *of the program is a sequence*

$$\gamma_0\gamma_1\cdots$$

such that

- $\gamma_0 = (Z_0, \emptyset)$, *where*

$$Z_0 = \mathtt{update}(\mathtt{ext}(I), \{\langle x, v\rangle^- \mid \forall r \in R(\langle x, v\rangle \notin \mathcal{A}_r)\}),$$

- $\gamma_{i+1} = (Z_{i+1}, R^f_{i+1}) \stackrel{df}{=}$

$$= \begin{cases} (\emptyset, R^f_i) & \text{if } Z_i = \emptyset, \\ (\mathtt{never}(S, \mathtt{update}(Z_i, \mathtt{ext}(\mathcal{A}_{r_f})), R), R^f_i \cup \{r_f\}) & \text{if } Z_i \neq \emptyset \text{ and} \\ & r_f \in \overline{R_i} \neq \emptyset, \\ (Z_i, R^f_i) & \text{if } \overline{R_i} = \emptyset, \end{cases}$$

where $\overline{R_i} = \{r | r \in R \setminus R^f_i \wedge (S, Z_i) \vdash_3 r\}$. □

We say that the constructive response *terminates* at Z_m if and only if $Z_m = \emptyset$ or $R_m = \emptyset$. A terminating constructive response is *accepted* if and only if $Z_m \neq \emptyset$ and $(\forall \langle x, v\rangle^a \in Z_m)(a \neq \perp)$.

In the rest of the paper, we will often speak of a constructive response being just a sequence of extended states (i.e. neglecting its R^f part).

7.2 Some properties of the semantics

First we introduce a lemma which will be used later.

Lemma 2 *Let $P' = Z'_0, \ldots, Z', \ldots Z'_m$ be a constructive response with $Z'_m \neq \emptyset$. Let $Z \preceq Z'$. Then for any constructive response $P = Z_0, \ldots, Z, \ldots Z_p$, where $Z_p \neq \emptyset$, for any Z_k appearing in P after Z, we have $Z_k \preceq Z'_m$.* □

Theorem 1 *Given a program $P = \langle N, D, R\rangle$, an initial state $(S, \emptyset)$ and a stimulus I, all the constructive responses terminating in a non-empty state must reach the same final state.* □

This means that independently of the order of the rules, if there exists a constructive response terminating in a non-empty extended state, then all constructive responses terminating in non-empty extended states must reach the same final state.

Theorem 2 *Given a program $P = \langle N, D, R\rangle$, an initial state $(S, \emptyset)$ and a stimulus I, if there is one constructive response terminating in an empty set, then all the other constructive responses will also terminate in an empty set.* □

Finally, we prove that any accepted constructive response is a correct response with respect to the declarative semantics. In order to prove this, we first define a mapping from an accepted constructive response to a sequence of firings and then prove that this sequence is a declarative response. Secondly, we prove that this declarative response is a correct one with respect to the declarative semantics.

Definition 30 *Let $CR = \gamma_0\gamma_1 \ldots \gamma_m$ be an accepted constructive response, where $\gamma_i = \langle Z_i, R^f_i\rangle, 0 \leq i \leq m$. Then*

$$\mathtt{map}(CR) \stackrel{df}{=} \sigma_0\sigma_1 \ldots \sigma_m,$$

where $\sigma_i = (Z^-_i, R^f_i)$. □

Corollary 6 *Let $P = \langle N, D, R\rangle$ be a program. Given an initial state $(S, \emptyset)$ and a stimulus I, if $CR = \gamma_0\gamma_1 \ldots \gamma_m$ is an accepted constructive response, then $DR = \mathtt{map}(CR)$ is a declarative response.* □

We now proceed with the major steps in the soundness proof. We assume the following notation in the coming lemmas. Let $P = \langle N, D, R\rangle$ be a program, $(S, \emptyset)$ be an initial state and I be a stimulus. Let $CR = \gamma_0\gamma_1 \ldots \gamma_m$ be an accepted constructive response. Let the corresponding declarative response be $DR = \mathtt{map}(CR) = \sigma_0, \sigma_1, \ldots, \sigma_m$, where $\gamma_m = (Z_m, R^f_m), \sigma_m = (C_m, R^f_m)$.

Lemma 3 *If $DR = \mathtt{map}(CR)$ is a declarative response obtained from an accepted constructive response CR, then DR is slot-consistent.* □

Lemma 4 *If $DR = \mathtt{map}(CR)$ is a declarative response obtained from an accepted constructive response CR, then DR is rule-consistent.* □

Lemma 5 *If $DR = \mathtt{map}(CR)$ is a declarative response obtained from an accepted constructive response CR, then DR is unambiguous.* □

Theorem 3 *(Soundness) If $DR = \mathtt{map}(CR)$ is a declarative response obtained from an accepted constructive response CR, then DR is correct.* □

Since the constructive semantics is not sensitive to the order of the rules in a program, we can derive the response using any arbitrary ordering of rules.

The static analysis based on the constructive semantics accepts a program if and only if for every $(S, \emptyset)$ and any I, the constructive response is acceptable. The benefit of this approach is that some nonstratifiable programs correct with respect to the declarative semantics can be accepted by the procedure. Programs 4 and 5 are such examples.

Program 5

```
rule1:   WHEN A *= a1  IF NOT B *= b1 then C := c0;
rule2:   WHEN A *= a1  IF NOT C *= c1 then B := b1;
```

8 Related works

The programs written in RL can be seen as logic programs (in an exotic temporal logic [13]). The semantics of logic programs has been investigated for a long time [20, 6, 8] in the logic programming and deductive database communities. Since our reactive rule-based programs contain negation, we are able to apply some results from the above areas in a new context.

Other reactive and real-time programming languages, such as Statecharts [10, 11, 12, 19] and Esterel [4, 2, 3] have to tackle similar semantical problems. The causality paradox in synchronous languages [9] is a well-known problem leading to ill-defined programs. This corresponds to our programs being rule-inconsistent or ambiguous. Esterel has a similar approach as ours, and attempts to reject ill-defined programs at compile time [2, 3]. The next version of Esterel compiler is expected to adopt a constructive semantics [3]. That constructive semantics is also used in cyclic circuit analysis [15, 21], somewhat resembling analysis of RL programs. In case of Statecharts [19] global consistency is considered, which corresponds to the rule-consistency in our declarative semantics.

Our rule-based programs operate on state variables (slots), which are different from events or signals adopted in pure synchronous languages. Therefore, the slot-consistency condition is specific for our approach.

9 Conclusion

The paper discusses some semantical problems related with reactive rule-based languages. We have introduced correctness criteria for a program and its responses, corresponding to our intuition regarding the behaviour of unambiguous, declarative programs and the demands of real-time safety-critical applications.

We have described the operational semantics of an implemented inference engine and discussed problems it had raised. Then we have formalized the intended behaviour of the system using a declarative approach. Finally, in order to be able to recognize correct programs, we have discussed two methods. The simpler one consists of a syntactical check whether the program is stratifiable or not. The other one introduces a semantics-based procedure using constructive semantics, which performs an exhaustive check of acceptability of the responses for all possible states and stimuli. Considering the complexity issue, the latest techniques for state space search (e.g. BDD based methods [5]) have to be employed in an implementation of the static checker. Finally, we have proven that all the programs passing the constructive check procedure are correct ones with respect to the desired declarative semantics.

This work is going to be continued in several directions. As RL and its inference engine are an element in a larger software architecture [18], we intend to incorporate the results into semantical analysis of layered (hybrid) software. Another issue we plan to investigate is timing analysis of correct programs, possibly building on the results obtained for MRL [23].

References

[1] C. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In *IBM Res. Rep*, RC 11681. April 1986.

[2] G. Berry. The semantics of pure Esterel. In M. Broy, editor, *Program Design Calculi*, volume 118 of *F: Computer and System Sciences*, pages 361–409. NATO ASI Series, 1993.

[3] G. Berry. The constructive semantics of pure Esterel. Technical report, Centre de Mathematiques Appliquees, 1995.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, Semantics, Implementation. *Science of computer programming*, (19):87–152, 1992.

[5] R .E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. In *ACM Computing Surveys*, volume 24, pages 293–318. september 1992.

[6] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1, March 1989.

[7] A. Chandra and D. Harel. Horn clause queries and generalizations. *J. Logic Programming*, 1:1–15, 1985.

[8] M. Fitting. A Kripke-Kleene semantics for logic programs. *J. Logic Programming*, 2:295–312, 1985.

[9] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[10] D. Harel. A visual formalism for complex system. *Sci. Comput. Programming*, 8(3):231–274, 1987.

[11] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *The 2nd IEEE Symposium on Logic in Computer Science*, 1987.

[12] C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. *LNCS*, 600:291–314, 1992.

[13] Jacek Malec. Applied knowledge representation. *CC-AI: The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, 9(1):9–41, 1992.

[14] Jacek Malec, Magnus Morin, and Ulf Palmqvist. Driver support in intelligent autonomous cruise control. In *Proceedings of the IEEE Intelligent Vehicles Symposium'94*, pages 160–164, Paris, France, October 1994.

[15] Sharad Malik. Analysis of cyclic combinatinal circuits. *IEEE Transactions on Computer-aided Design of Integraged Circuits and Systems*, 13(7):955–956, July 1994.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.

[17] Magnus Morin. RL: An embedded rule-based system. Technical Report LAIC-IDA-94-TR2, Linköping University, 1994.

[18] Magnus Morin, Simin Nadjm-Tehrani, Per Österling, and Erik Sandewall. Real-time hierarchical control. *IEEE Software*, 9(5):51–57, September 1992.

[19] A. Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. *Theoretical Aspects of Computer Software, LNCS*, 526:510–584, 1991.

[20] J.C. Shepherdson. Negation as failure. *J. Logic Programming*, 2(3):185–202, 1985.

[21] Thomas R. Shiple, Gerard Berry, and Herve Touati. Constructive analysis of cyclic circuits. In *EDTC*, Paris, March 1996.

[22] J. D. Ullman. *Principles of Databases and Knowledges-base Systems*, volume 1. Computer Science Press, Potomac, MD, 1988.

[23] R. H. Wang and A. K. Mok. Timing analysis of MRL: a real time rule-based system. *J. Real Time Systems*, March 1993.

A The proofs of the theorems

Lemma 2 *Let $P' = Z'_0, \ldots, Z', \ldots Z'_m$ be a constructive response with $Z'_m \neq \emptyset$. Let $Z \preceq Z'$. Then for any constructive response $P = Z_0, \ldots, Z, \ldots Z_p$, where $Z_p \neq \emptyset$, for any Z_k appearing in P after Z, we have $Z_k \preceq Z'_m$.* □

Proof: By induction over the sequence $Z, \ldots, Z_p$:

- Base step: Since $Z \preceq Z'$, $Z' \preceq Z'_m$ and $Z'_m \neq \emptyset$, we have $Z \preceq Z'_m$.
- Induction step: Let Z_i denote any element in the sequence $Z, \ldots, Z_p$ We prove that if $Z_i \preceq Z'_m$, then $Z_{i+1} \preceq Z'_m$. The constructive response derives this new extended state Z_{i+1} by firing a rule r, with $(S, Z_i) \vdash_3 r$. Since $Z_i \preceq Z'_m$, we have $(S, Z'_m) \vdash_3 r$, then r must be fired before Z'_m in response P', otherwise P' would not terminate. Hence, all the changes from Z_i to Z_{i+1} (the status of some tuples changes from $\perp$ to - or +) must also be included in Z'_m. Thus, $Z_{i+1} \preceq Z'_m$.
- Therefore, for any Z_k in the sequence $Z, \ldots, Z_p$, we have $Z_k \preceq Z'_m$. □

Theorem 1 *Given a program $P = \langle N, D, R \rangle$, an initial state $(S, \emptyset)$ and a stimulus I, all the constructive responses terminating in a non-empty state must reach the same final state.* □

Proof: Let P_1 and P_2 be two responses for $\langle S, \emptyset \rangle$ and I,

$$P_1 = Z_1, \ldots, Z_{m_1},$$

$$P_2 = Z'_1, \ldots, Z'_{m_2},$$

such that Z_{m_1} and Z'_{m_2} are the final states for P_1 and P_2, respectively, and $Z_{m_1} \neq \emptyset$, $Z'_{m_2} \neq \emptyset$. We shall prove $Z_{m_1} = Z'_{m_2}$ by contradiction. Assume that $Z_{m_1} \neq Z'_{m_2}$. Since the initial state is the same, there exists such i that $Z_i \neq Z'_i$ but $Z_{i-1} = Z'_{i-1}$. Since Z_{m_1} is a non-empty state in P_1 which contains Z_{i-1} as an element, and $Z_{i-1} \preceq Z'_{m_2}$, then by lemma 2 we know that $Z_{m_1} \preceq Z'_{m_2}$. From the symmetrical argument, we have $Z'_{m_2} \preceq Z_{m_1}$. Thus $Z'_{m_2} = Z_{m_1}$. □

Theorem 2 *Given a program $P = \langle N, D, R\rangle$, an initial state $(S, \emptyset)$ and a stimulus I, if there is one constructive response terminating in an empty set, then all the other constructive responses will also terminate in an empty set.* □

Proof: By contradiction. Let $P_1 = Z_0, \ldots, Z_k, \emptyset$ be a constructive response, with $Z_k \neq \emptyset$, and let r_1 be a rule such that $(S, Z_k) \vdash_3 r_1 \wedge \neg\text{slot-cons}(Z_k, \mathcal{A}_{r_1})$. Assume that there is a response $P_2 = Z'_1, \ldots, Z'_m$ terminating in $Z'_m \neq \emptyset$. First, we prove that $Z_k \preceq Z'_m$. Since the initial state is the same, there exists such i that$Z_i \neq Z'_i$ and $Z_{i-1} = Z'_{i-1}$, $(i < k)$. Because of monotonicity of the response, $Z_{i-1} \preceq Z'_m$. Since the non-empty state Z_k is derived from Z_{i-1}, and Z'_m is the final state of the constructive response, by lemma 2 we have $Z_k \preceq Z'_m$. Thus $(S, Z'_m) \vdash_3 r_1$, and r_1 should have been fired before (S, Z'_m), otherwise, P_2 would have not terminated. Since $Z'_m \neq \emptyset$, we have slot-cons$(Z_m, \mathcal{A}_{r_1})$. And since $Z_k \preceq Z'_m$, it must be the case that slot-cons$(Z_k, \mathcal{A}_{r_1})$ what contradicts the fact that $\neg$slot-cons$(Z_k, \mathcal{A}_{r_1})$. □

Corollary 6 *Let $P = \langle N, D, R\rangle$ be a program. Given an initial state $(S, \emptyset)$ and a stimulus I, if $CR = \gamma_0\gamma_1 \ldots \gamma_m$ is an accepted constructive response, then $DR =$ `map`(CR) is a declarative response.* □

Proof: Let `map`(CR) be $\sigma_0, \sigma_1, \ldots, \sigma_m$. By the definition of `map`, $\sigma_0 = (I, \emptyset)$. Now we prove that for all i such that $(0 \leq i < m)$, $\sigma_{i+1} = (Z^-_{i+1}, R^f_{i+1})$ succeeds $\sigma_i = (Z^-_i, R^f_i)$ in acrordance with the declarative semantics. Let r_f be the element of the set $R^f_{i+1} \setminus R^f_i$ (note that this set is a singleton). Since $(S, Z_i) \vdash_3 r_f$, by corollary 3 we have $(S, C_i) \vdash r_f$. Z_{i+1} is derived from Z_i by firing the rule r_f, therefore, all the new positive changes in Z_{i+1} are $\mathcal{A}_{r_f}$, thus $Z^-_{i+1} = Z^-_i \cup \mathcal{A}_{r_f}$. Firing r_f at (S, C_i) yields (S, C'), where $C' = C_i \cup \mathcal{A}_{r_f}$. Since $C_i = Z^-_i$, we have $C' = Z^-_{i+1} = C_{i+1}$. Thus, Firing r_f at (C_i, R^f_i) using the declarative semantics yields exactly (C_{i+1}, R^f_{i+1}), where $C_{i+1} = Z^-_{i+1}$. Thus, DR is a declarative response. Finally, we prove that using the declarative semantics, DR will terminate in (C_m, R^f_m). Since Z_m is non-empty, the constructive response terminates at (Z_m, R^f_m) if and only if $\neg(\exists r \in (R \setminus R^f_m)((S, Z_m) \vdash_3 r))$. Since Z_m has no changes annotated with a $\perp$ annotation, by corollary 4 we have that $\neg(\exists r \in (R \setminus R^f_m)((S, C_m) \vdash r))$. By the terminating condition of declarative semantics, DR terminates exactly at (C_m, R^f_m). □

Let $P = \langle N, D, R\rangle$ be a program, $(S, \emptyset)$ be an initial state and I be a stimulus. Let $CR = \gamma_0\gamma_1 \ldots \gamma_m$ be an accepted constructive response. Let the corresponding declarative response be $DR = \texttt{map}(CR) = \sigma_0, \sigma_1, \ldots, \sigma_m$, where $\gamma_m = (Z_m, R^f_m), \sigma_m = (C_m, R^f_m)$.

Lemma 3 *If $DR = \texttt{map}(CR)$ is a declarative response obtained from an accepted constructive response CR, then DR is slot-consistent.* □

Proof: Since $Z_m \neq \emptyset$, by the construction of the constructive response we have

$$\forall \langle x, v\rangle (\langle x, v\rangle^+ \in Z_m \rightarrow (\forall v' \in D_x (v' \neq v \rightarrow \langle x, v'\rangle^+ \notin Z_m))).$$

By the definition of C_m (the `map` function), we have

$$\forall \langle x, v\rangle (\langle x, v\rangle \in C_m \rightarrow (\forall v' \in D_x (v' \neq v \rightarrow \langle x, v'\rangle \notin C_m))).$$

Hence, $\neg \exists x (\langle x, v\rangle \in C_m, \langle x, v'\rangle \in C_m \wedge v \neq v')$. Thus, DR is slot consistent. □

Lemma 4 *If $DR = \texttt{map}(CR)$ is a declarative response obtained from an accepted constructive response CR, then DR is rule-consistent.* □

Proof: Since $Z_m \neq \emptyset$, and $\forall i (0 \leq i \leq m) \rightarrow Z_i \preceq Z_m$, by corollary 5, we have

$$\forall r \in R^f_m ((S, Z_m) \vdash_3 r).$$

And since Z_m has no changes with $\perp$ annotations, by corollary 3, we have

$$\forall r \in R^f_m ((S, C_m) \vdash r).$$

Thus, DR is rule consistent. □

Lemma 5 *If $DR = \texttt{map}(CR)$ is a declarative response obtained from an accepted constructive response CR, then DR is unambiguous.* □

Proof:

We shall prove that DR is unambiguous. Let $DR' \neq DR$ be any slot-consistent, rule-consistent declarative response of the program P, where

$$DR' = \sigma_0 \sigma'_1 \ldots \sigma'_{m'} \wedge \sigma'_i = (C'_i, R^{f'}_i).$$

Then we shall prove that $C'_{m'} = C_m$. We only need to prove that

$$C_m \subseteq C'_{m'} \text{ and } C'_{m'} \subseteq C_m.$$

To prove $C_m \subseteq C'_{m'}$, that is $Z^-_m \subseteq C'_{m'}$, we show by induction over k that for any $1 \leq k \leq m$, $P(k)$ holds, where

$$P(k) : Z^-_k \subseteq C'_{m'} \wedge \forall \langle x, v\rangle (\langle x, v\rangle^- \in Z_k \rightarrow \langle x, v\rangle \notin C'_{m'}).$$

Since the initial states of DR and DR' are the same, and $DR' \neq DR$, there must exist an i, where the firings of DR and DR' become different, that is,

$$\sigma_i \neq \sigma'_i \wedge \sigma_{i-1} = \sigma'_{i-1}.$$

`Base step:` $1 \leq k \leq (i-1)$ We have

$$Z_k^- \subseteq Z_{i-1}^- = C_{i-1} = C'_{i-1} \subseteq C'_{m'}.$$

Thus we only need to show that

$$\forall \langle x, v\rangle(\langle x, v\rangle^- \in Z_k \rightarrow \langle x, v\rangle \notin C'_{m'}).$$

It suffices to show it for $k = (i-1)$. To do this, we show by induction over j that $Q(j)$ holds for $(i-1) \leq j \leq m'$, where

$$Q(j) : \forall \langle x, v\rangle(((\langle x, v\rangle^- \in Z_{i-1}) \wedge (\langle x, v\rangle \notin C'_{j-1})) \rightarrow \langle x, v\rangle \notin C'_j).$$

- Base case: $j = i-1$. Since CR terminates at a non-empty state and

  $$C'_{i-1} = C_{i-1} = Z_{i-1}^-,$$

  we have

  $$\forall \langle x, v\rangle(\langle x, v\rangle^- \in Z_{i-1} \rightarrow \langle x, v\rangle^+ \notin Z_{i-1} \rightarrow \langle x, v\rangle^+ \notin C'_{i-1}),$$

  that is, $Q(j)$ holds for $j = i-1$.

- Induction step: $i \leq j \leq m'$. Suppose

  $$\forall \langle x, v\rangle(((\langle x, v\rangle^- \in Z_{i-1}) \rightarrow ((\langle x, v\rangle \notin C'_{j-1})).$$

  Assume r_j is fired at σ'_{j-1} in DR' and results in σ'_j, we only need to prove that

  $$\forall \langle x, v\rangle(((\langle x, v\rangle^- \in Z_{i-1}) \rightarrow ((\langle x, v\rangle \notin \mathcal{A}_{r_j})).$$

  We prove this by contradiction. Assume that there exists an $\langle x^*, v^*\rangle$, where

  $$\langle x^*, v^*\rangle^- \in Z_{i-1} \wedge \langle x^*, v^*\rangle \in \mathcal{A}_{r_j}.$$

  Since $\langle x^*, v^*\rangle^- \in Z_{i-1}$, we have

  $$\forall r(\langle x^*, v^*\rangle \in \mathcal{A}_r \rightarrow (S, Z_{i-1}) \not\vdash_3 r),$$

  and since $\langle x^*, v^*\rangle \in \mathcal{A}_{r_j}$, therefore

  $$S, Z_{i-1}) \not\vdash_3 r_j.$$

  And since

  $$Z_{i-1}^- = C_{i-1} = C'_{i-1} \subseteq C'_{j-1}$$

  and

  $$\forall \langle x, v\rangle(((\langle x, v\rangle^- \in Z_{i-1}) \rightarrow ((\langle x, v\rangle \notin C'_{j-1}))$$

  and DR' is slot-consistent, that is

  $$\forall x((x, v_1) \in C'_{j-1} \wedge (x, v_2) \in C'_{j-1} \rightarrow v_1 = v_2),$$

  by corollary 3. 2 we have $(S, C'_{j-1}) \not\vdash r_j$, which means that r_j can not be fired at σ'_{j-1} in DR', what leads to contradiction.

Therefore, the base case for $P(k), 1 \leq k \leq (i-1)$ follows. Now we proceed with the induction step for $P(k)$.
`Induction step` for $P(k)$: Assume that

$$Z_{k-1}^{-} \subseteq C'_{m'} \wedge \forall \langle x, v\rangle((\langle x, v\rangle^{-} \in Z_{k-1} \rightarrow \langle x, v\rangle \notin C'_{m'}).$$

Thus we need to show that $Z_k^{-} \subseteq C'_{m'}$ and

$$\forall \langle x, v\rangle((\langle x, v\rangle^{-} \in Z_k \rightarrow \langle x, v\rangle \notin C'_{m'}).$$

We show $Z_k^{-} \subseteq C'_{m'}$ first. By corollary 3. 1, we have

$$\forall r((S, Z_{k-1} \vdash_3 r \rightarrow (S, C'_{m'}) \vdash r),$$

thus any rule fired at step γ_{k-1} in CR should also have been fired up to $\sigma'_{m'}$ in DR', hence, all the positive changes added to Z_k should also be included $C'_{m'}$, that is,

$$Z_k^{-} \subseteq C'_{m'}.$$

Next, we show that

$$\forall \langle x, v\rangle((\langle x, v\rangle^{-} \in Z_k \rightarrow \langle x, v\rangle \notin C'_{m'}).$$

Since

$$\forall \langle x, v\rangle((\langle x, v\rangle^{-} \in Z_{k-1} \rightarrow \langle x, v\rangle \notin C'_{m'}),$$

we only need to show that

$$\forall \langle x, v\rangle((((\langle x, v\rangle^{-} \in Z_k) \wedge ((\langle x, v\rangle^{-} \notin Z_{k-1})) \rightarrow \langle x, v\rangle \notin C'_{m'}).$$

By the definition of the recursive function `never`, we only need to show the following argument: for an extended state Z, if $Z^{-} \subseteq C'_{m'}$ and

$$\forall \langle x, v\rangle((\langle x, v\rangle^{-} \in Z) \rightarrow \langle x, v\rangle \notin C'_{m'})$$

and

$$Neg = \{\langle x, v\rangle^{-} \mid \forall r(\langle x, v\rangle \in \mathcal{A}_r \rightarrow (S, Z) \not\vdash_3 r)\},$$

then

$$\forall \langle x, v\rangle((\langle x, v\rangle^{-} \in Neg) \rightarrow \langle x, v\rangle \notin C'_{m'}).$$

Assume $\langle x^*, v^*\rangle \in Neg$, thus

$$\forall r(\langle x^*, v^*\rangle \in \mathcal{A}_r \rightarrow (S, Z) \not\vdash_3 r),$$

and since $Z^{-} \subseteq C'_{m'}$ and

$$\forall \langle x, v\rangle((\langle x, v\rangle^{-} \in Z) \rightarrow \langle x, v\rangle \notin C'_{m'}),$$

and $C'_{m'}$ is slot-consistent, by corollary 3. 2

$$\forall r((S, Z) \not\vdash_3 r \rightarrow (S, C'_{m'} \not\vdash r),$$

thus

$$\forall r(\langle x^*, v^* \rangle \in \mathcal{A}_r \rightarrow (S, C'_{m'}) \not\vdash r),$$

and by the rule consistency of DR',

$$\forall r(\langle x^*, v^* \rangle \in \mathcal{A}_r \rightarrow r \notin R^{f'}_{m'}).$$

Hence $\langle x^*, v^* \rangle \notin C'_{m'}$. Therefore the above argument holds. Applying the above argument to Z_{k-1} and the intermediate extended states derived by function `never`, we have

$$\forall \langle x, v \rangle (\langle x, v \rangle^- \in Z_k \rightarrow \langle x, v \rangle \notin C'_{m'}).$$

Thus, the induction steps hold.

Therefore,

$$Z^-_m \subseteq C'_{m'},$$

hence

$$C_m \subseteq C'_{m'}.$$

Finally, we shall prove that

$$C'_{m'} \subseteq C_m.$$

Since

$$\forall \langle x, v \rangle (\langle x, v \rangle^- \in Z_m \rightarrow \langle x, v \rangle \notin C'_{m'}),$$

that is,

$$\forall \langle x, v \rangle (\langle x, v \rangle \in C'_{m'} \rightarrow \langle x, v \rangle^- \notin Z_m).$$

And since Z_m contains no changes annotated with $\perp$, then

$$\forall \langle x, v \rangle (\langle x, v \rangle^- \notin Z_m \rightarrow \langle x, v \rangle^+ \in Z_m)$$

holds. Hence,

$$\forall \langle x, v \rangle (\langle x, v \rangle \in C'_{m'} \rightarrow \langle x, v \rangle^+ \in Z_m).$$

Therefore,

$$C'_{m'} \subseteq Z^-_m,$$

since $C_m = Z^-_m$,

$$C'_{m'} \subseteq C_m$$

follows.

Hence

$$C_m = C'_{m'}.$$

This concludes our proof. □

Theorem 3 *(Soundness) If $DR =$ `map`(CR) is a declarative response obtained from an accepted constructive response CR, then DR is correct.* □

Proof: Directly from lemmas 3, 4 and 5. □