On semantics and correctness of reactive rule-based programs

Man Lin[1]*, Jacek Malec[2]**, Simin Nadjm-Tehrani[1]

[1] Department of Computer and Information Science
Linköping University
S–581 83 Linköping, Sweden
linma,jam,snt@ida.liu.se
[2] Department of Computer Engineering
Mälardalens Högskola
Box 883, S–721 23 Västerås, Sweden

Abstract. The rule-based paradigm for knowledge representation appears in many disguises within computer science. In this paper we address special issues which arise when the rule-based programming paradigm is employed in the development of reactive systems. We begin by presenting a rule-based language RL which has emerged while developing intelligent cruise control systems. We define a desired declarative semantics and correctness criteria for rule-based programs which respect causality, synchrony assumption and desired determinism. Two alternative approaches are proposed to analyze RL programs. Both approaches build upon static checks of a rule-based program. In the first approach we accept programs which are correct with respect to a constructive semantics while in the second approach, a stratification check is imposed. The combination of rules and reactive behaviour, together with a formal analysis of this behaviour is the main contribution of our work.

1 Overview

The rule-based paradigm for knowledge representation appears in many disguises within computer science. Language issues related to this paradigm appear in production systems [3], parallel program design (e.g. Unity [2]), default reasoning within AI [9], logic programming [1], rewriting [7], active and deductive databases [4], and logics for action and change [15].

Our work combines results from the three areas of rule-based knowledge representation, reactive systems [11, 6], and programming language semantics. The combination of rules and reactive behaviour, together with a formal analysis of this behaviour is thus the main contribution of our work. Different approaches

* Man Lin has been supported by TFR (Swedish Research Council for Engineering Sciences) and WITAS (the Wallenberg laboratory for research on Information Technology and Autonomous Systems).
** Jacek Malec has been partially supported by Mälardalens Real-Time Research Center (MRTC).

for specification of real-time and reactive systems range over automata-based, temporal logics, Petri nets, action systems, and process algebras. In our view a rule-based language with a formal semantics shares the benefits of these specification languages. In addition, it has a special appeal: it mimics the natural mode of reasoning by humans in many applications. Therefore, it can be considered as a powerful tool for capturing expert knowledge and formally analyzing it. Moreover, rules can be executed and can therefore be seen as both a specification and a programming language.

The synchronous family of high-level programming languages [5] for real-time systems (Lustre, Esterel, Signal) share the above characteristic. They too can be used both for capturing high level design and as executable code. Though very different in syntax and style of programming, adding reactiveness to our rules leads to formal semantics which is reminiscent of a couple of the proposed semantics for Statecharts [14], and Esterel [13].

2 Rules and Reactiveness

A reactive rule-based system (illustrated in Figure 1) is a system that reacts to the changes of its environment continuously [12]. Such a system is composed of three entities called *state*, *rules*, 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, producing new changes in the slots, which in turn trigger other rules, and so on. This is continued 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.

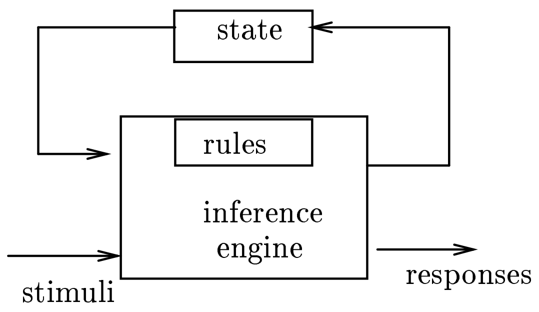

Fig. 1. A reactive rule-based system.

The rule language RL (syntax can be found in the appendix) is developed to express responses of the system at each ACP. The language has been successfully used for developing a reactive application: a driver-support system [10].

The rules in an RL program have an event-condition-action form, e.g.:

```
WHEN A *= a IF (B *= b AND NOT E |= e) THEN D := d;
```

read as "When `A` changes to `a` then if `B` changes simultaneously to `b` and `E` has not been `e` then `D` obtains value `d`". The `WHEN` part: `A *= a` is called the *trigger* part of the rule, the `IF` part: `(B *= b AND NOT E |= e)` is called the *condition* part of the rule, and the `THEN` part `D := d` is called the *assignment* part of the rule. The trigger part and the condition part together are called the *precondition* of the rule. The characteristics of this language are:

- The meaning of a reactive program is independent of the ordering of the rules (in case of larger systems rule ordering is a cumbersome and error-prone process; the semantics of such programs is unclear and easy to distort). In our approach a program can be enhanced by simply adding new rules to the existing rule base;
- The language assumes finite domains for variables (c.f. datalog) allowing a finite model;
- The language allows the logical operations, negation and conjunction;
- The language allows for taking account of concurrent events (in the example rule events `A *= a` and `B *= b` occur simultaneously);
- The language models time flow without introducing metric time (`E |= e` checks if "`E` has had value `e` before", while `E *= e` checks if "`E` has changed to value `e`");

A rule responds to external stimuli at a given state by checking whether the rule is enabled at the current state, and firing the rule (performing the assignments) if so is the case.

A stimulus to a system, denoted as I, is a set of *changes* which are (slot, value) pairs. A state of a rule-based system is a pair (S, C) where S contains the values of all the variables (slots), and C contains the set of changes. We use S_x to denote the value of x in the latest EP. During an EP, S is the same and $C = \emptyset$. At an ACP, S is the same as S in the previous EP and C contains the changes occurring at this ACP including the external stimuli and the changes derived as the result of the assignments of the enabled rules.

A rule r being enabled at a state (S, C) is denoted by $(S, C) \vdash r$. A rule r being not enabled at a state (S, C) is denoted by $(S, C) \not\vdash r$. To check whether $(S, C) \vdash r$, we only need to check if all the primitive preconditions of rule r are satisfied at (S, C). By primitive precondition, we mean positive condition including X `|=` v (was), X `*=` v (changes to), or negative condition including NOT X `|=` v (was not), NOT X `*=` v (does not change to). The trigger part of a rule contains only one primitive condition X `*=` v , while the condition part of a rule can be a conjunction of primitive conditions. We define $\vdash$ for rules by first defining $\vdash$ for primitive conditions of rules, here delimited by [ ].

- $(S, C) \vdash [x\,|\!=\!v]$ iff $S_x = v$;
- $(S, C) \vdash [x{*}{=}v]$ iff $S_x \neq v$ and $\langle x, v\rangle \in C$;
- negation (NOT) and conjunction (AND) are interpreted as standard logical connectives. That is:
  - $(S, C) \vdash [\text{ NOT } p]$ where p is any positive primitive iff not $(S, C) \vdash p$.
  - $(S, C) \vdash [p_1 \text{ AND } p_2]$ where p_1 and p_2 are primitive preconditions iff $(S, C) \vdash p_1$ and $(S, C) \vdash p_2$.
- $(S, C) \not\vdash r$ iff not $(S, C) \vdash r$.

Let's look at a simple example. Suppose x, y, and z are the three slots of the system. Let $S_x = 0$, $S_y = 0$, $S_z = 0$ and $C = I = \{\langle x, 1\rangle\}$. Program `P1` contains only one rule `r1`:

```
r1:     WHEN x *= 1 IF y |= 0 THEN z := 1;
```

Rule `r1` is enabled at (S, C) since $\langle x, 1\rangle \in C$ and $S_y = 0$. The effect of firing this rule is to assign 1 to z. Therefore, the set of changes becomes $C1 = \{\langle x, 1\rangle, \langle z, 1\rangle\}$.

Let's consider another program `P2` containing only `r2` with the same (S, C):

```
r2:     WHEN z *= 1 IF y |= 0 THEN y := 1;
```

Rule `r2` is not enabled at (S, C) since $\langle z, 1\rangle \notin C$. Therefore, the set of changes is still $\{\langle x, 1\rangle\}$.

If an RL program contains several rules, then the response of the system at each ACP may no longer be only one (or zero) firing of rule. There could be several rule firings some of which are caused by others.

3 Synchrony Assumption and Causality

One might ask why the responses only occur at ACPs. The fundamental assumption taken here is the *synchrony* assumption: each response is assumed to be synchronous with the effects it causes. This assumption is realistic if the responses of the system are fast enough so that the environment does not change during the responses (which should be checked in practice). The effects of the execution of one component are instantly broadcast to all the other components of the system. Therefore, all the components of the system have the same view of the system state.

The smallest component of an RL program is one single rule. If several rules get fired at the same ACP, then all the rule firings are considered to occur at the same time. We don't care how the rule firings are done step by step if only synchrony requirement is considered. What is interesting is only the result of the response. The result of a response at (S, I) is a stable state (S, C') and a set of fired rules R^f where:

- C' is the result of firing all the rules in R^f at the given initial state (S, I). Let $\mathcal{A}_r$ denote the assignments of rule r. Then

$$C' = \bigcup_{r \in R^f} \mathcal{A}_r \cup I.$$

(S, C') is seen as the state after the response.

- R^f is the maximal set of rules that are enabled at state (S, C'). First, all the rules in R^f are enabled at (S, C'). Second, no other rules not belonging to R^f are enabled at (S, C').

However, we would like to retain *causality* which is a very important property for a reasoning system. The principle of causality requires that any change issued should have a sequence of (enabled) rule firings leading to it. The following example shows a causal reasoning. By composing earlier programs `P1` and `P2`, we get a new program `P3` which contains two rules: `r1` and `r2`. One can infer that both `r1` and `r2` are fired and the new set of changes becomes $C3 = \{\langle x, 1\rangle, \langle y, 1\rangle, \langle z, 1\rangle\}$. The reasoning is simple. Since `r1` is enabled at (S, I), `r1` is fired and the effect: the change $\langle z, 1\rangle$ is instantaneously broadcast. The system state becomes $(S, C1)$ where $C1 = \{\langle x, 1\rangle, \langle z, 1\rangle\}$. Since `r2` is enabled at $(S, C1)$, `r2` is also fired and results in the final set of changes $C3$.

For the above example, $C' = C3$ and $R^f = \{r1, r2\}$. The synchrony requirement is also satisfied since $C3 = I \cup \mathcal{A}_{r1} \cup \mathcal{A}_{r2}$, and `r1` and `r2` are the only rules enabled at $(S, C3)$.

However, not all the responses respect both synchrony hypothesis and the principle of causality. Let's look at two examples.

Given S where $S_x = 0, S_y = 0, S_z = 0$, $I = \{\langle y, 1\rangle\}$ and a program with two rules `r3` and `r4`, what are the final state and the fired rule set?

```
r3: WHEN x *= 1 IF y |= 0 THEN z := 1;
r4: WHEN z *= 1 IF y |= 0 THEN x := 1;
```

There are two solutions which satisfy the synchrony requirement. One is $C' = I$ and $R^f = \emptyset$. The other is $C' = \{\langle y, 1\rangle, \langle x, 1\rangle, \langle z, 1\rangle\}$ and $R^f = \{r3, r4\}$. The problem with the second solution is that without the firing of `r4`, `r3` can not get fired. The same is for `r4`: without the firing of `r3`, `r4` can not get fired. The result is self-triggered. Or, in other words, it is not causal since we can not generate this final result via a causal sequence of rule firings.

The above example shows that not all the responses satisfying synchrony requirement are causal. Next, we show that not all the causal responses satisfy the synchrony requirement either.

Suppose (S, I) be $S_x = 0, S_y = 0, S_z = 0$, $I = \{\langle y, 1\rangle\}$, and a program be as follows.

```
r5: WHEN y *= 1 IF NOT x *= 1 THEN z := 1;
r6: WHEN y *= 1 IF x |= 0 THEN x := 1;
```

A causal rule firing sequence is `r5` followed by `r6` which results in
$C' = \{\langle y, 1\rangle, \langle x, 1\rangle, \langle z, 1\rangle\}$. The problem is that $r5$ is not enabled at (S, C') which violates the synchrony requirement.

4 Other Requirements

As we deal with variables, one important requirement is not to assign different values to the same variable at the same ACP. Another requirement is that there should be only one final result at each ACP. This requirement is understood as observable determinism.

Next, we provide a desired semantics definition for a response which respects the synchrony hypothesis, the principle of causality and the above requirements.

5 Declarative Semantics

Definition 1. Suppose R is the set of rules of a program P. The *declarative response* of the program P in a state (S, I) is any sequence of firings

$$\sigma_0\sigma_1\ldots$$

such that

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

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

□

In the definition, each firing (σ_i) contains a set of changes (C_i) and a set of fired rules (R_i^f).

It can be proved that a declarative response has always a finite length [8].

Definition 2. Let R be the rule set in a program P. Let a declarative response of the program in a state $(S, \emptyset)$ to a stimulus I be $\sigma_0\sigma_1\ldots\sigma_m$. Let $\sigma_m = (C_m, R^f)$ and $R^f = \{r_1, r_2, \ldots, r_m\}$. The declarative response is *correct* if and only if

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

A correct response is the desired response. This semantics is referred to as declarative semantics. An RL program is **correct** if and only if it has a correct response for any possible combination of state and stimuli. Two natural questions arise:

- Can we construct an operational semantics to implement the desired declarative semantics?
- Can we identify the ill-behaved programs during compile time without having to generate all the responses for each state-and-stimulus combination?

We will devote the next two sections to answering the above questions.

6 Constructive Semantics

6.1 The semantics

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 recently proposed semantics for pure Esterel [13]. 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). In what follows we present the constructive semantics.

The constructive semantics needs not only positive information about the changes of the system, but also negative information about the lack of changes. In constructive semantics, we deal with extended system state (S, Z) where S records the values of the slots before the ACP and Z contains a set of annotated changes where each (slot, value) pair has a annotation indicating the *status* of this change. The status is an element from the set $\{+, -, \perp\}$. $+$ is read as *positive*, and $\langle x, v\rangle^+$ means that $\langle x, v\rangle$ does occur in this ACP; $-$ is read as *negative*, and $\langle x, v\rangle^-$ means that the change $\langle x, v\rangle$ can not possibly occur in this ACP; $\perp$ is read as *Unknown*, and $\langle x, v\rangle^\perp$ means that the change of x to v is not present yet at this point of the computation, but it is not sure whether it will take place later.

The result of evaluation of a rule is one of the following: *True*, *False* or *Unknown* instead of only *True* or *False* as in 2-valued logic. The evaluation evaluates a rule to be *Unknown* if it is not known whether the rule will evaluate to true or false after this response. More specifically, a primitive condition (NOT x *= v) is evaluated to be *True* at a state (S, C) if $\langle x, v\rangle$ does not belong to C when reasoning under 2-valued logic, but *Unknown* in the case of constructive semantics if $\langle x, v\rangle$ is not explicitly marked with unchangeable status (positive or negative).

The ordering between the status annotations is

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

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

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

Given C, C^+ is defined as the extension of C where:

$$C^+ = \{\langle x, v\rangle^+ \mid \langle x, v\rangle \in C\} \cup \{\langle x, v'\rangle^- \mid \langle x, v\rangle \in C \wedge v' \neq v\} \cup \\ \cup \{\langle x, v\rangle^\perp \mid \forall v' \langle x, v'\rangle \notin C\}$$

Symmetrically, given Z, Z^- is defined as the reduction of Z where:

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

A rule being 3-enabled at an extended state (S,Z) is denoted by $(S,Z) \vdash_3 r$. A rule being 3-non-enabled at an extended state (S,Z) is denoted by $(S,Z) \nvdash_3 r$. $(S,Z) \vdash_3 r$ if and only if all the primitive preconditions are evaluated to be True at (S,Z). $(S,Z) \nvdash_3 r$ if and only if one of the primitive precondition is evaluated to be False at (S,Z). The evaluation of a primitive condition p at a given extended state (S,Z) is shown as follows:

- $[x\,|{=}v]$ is True if $S_x = v$;
  $[x\,|{=}v]$ is False if $S_x \neq v$;
- $[x{*}{=}v]$ is True if $\langle x, v\rangle^+ \in Z$ and $S_x \neq v$;
  $[x{*}{=}v]$ is False if $\langle x, v\rangle^- \in Z$ or $S_x = v$;
- [NOT p] is True if p is False; [NOT p] is False if p is True;
- With the exception of [True], all other primitive conditions are evaluated to Unknown.

It should be observed that there are intermediate cases when neither $(S,Z) \vdash_3 r$ nor $(S,Z) \nvdash_3 r$ is true.

The negative changes are derived by function `never`. Function `never` works iteratively. At each iteration, a negative change is added into the set of annotated changes. The change added has one of the following characteristics:

- No rule in the program can issue such change.
- All the rules that can issue such change are 3-non-enabled at the current extended state.

When we say adding negative changes or positive changes, we mean updating the annotation of the (slot, value) pair in the set of annotated changes. This is done by `update` function. The annotation can only be changed from $\perp$ to $+$ or $-$. An attempt to change the status from $+$ to $-$ or vice versa indicates a symptom of slot-inconsistency. When such situation occurs, the set of annotated changes returned is an empty set to indicate failure. The formal definition for `never` and `update` can be found in [8].

We are now in a position to define an operational semantics.

Definition 3. Given a program P with a rule set R, 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 $Z_0 = \texttt{never}(S, I^+, R)$,

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

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

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

As we can see, if there exists an unfired rule 3-enabled in the current state, and no slot-inconsistency occurred in the `update` of the previous step ($Z_i \neq \emptyset$), then the current set of annotated changes Z_i is updated with positive changes, and negative changes. The positive changes come either from the external stimulus (step 0) or from the assignments of the selected rule that is 3-enabled at the current extended state (subsequent steps). The negative changes derived by `never` function are those potential negative changes that could be deduced from the current state. If there is no unfired rule 3-enabled by the current state, then the procedure returns the same tuple as in the previous step. Finally, if the state indicates the occurrence of slot-inconsistency ($Z_i = \emptyset$), the procedure returns the empty set as the new set of annotated changes.

We say that the constructive response *terminates* at Z_m if and only if ($Z_m = \emptyset$ and $Z_{m-1} \neq Z_m$)or ($\overline{R_m} = \emptyset$ and $\overline{R_{m-1}} \neq \emptyset$), that is, a slot-inconsistency occurs or there is no rule to be selected.

A terminating constructive response is *accepted* if and only if it terminates at Z_m and $Z_m \neq \emptyset$ and $(\forall \langle x, v\rangle^a \in Z_m)(a \neq \bot)$. That is, an accepted constructive response terminates *normally*, meaning that no slot-inconsistency occurs ($Z_m \neq \emptyset$), and the set of annotated changes of its final state is *complete*, meaning that no change in Z_m is marked with $\bot$.

6.2 Properties

It can be proved that given a program P and (S, I), all the constructive responses reach the same final set of annotated changes (the thereom can be found in [8]).

It can also be proved that any accepted constructive response yields a correct declarative response. 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 construction of a declarative response (see [8]). Then, we prove that this declarative response is a correct one (see theorem Soundness).

Definition 4. 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

$$\texttt{map}(CR) {=} \sigma_0, \sigma_1, \ldots, \sigma_m,$$

where $\sigma_i = (Z^-_i, R^f_i)$. □

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

The proof can be found in [8].

The static checker performs an exhaustive check of acceptability of the responses for all possible states and stimuli. It can be easily proved that that all the programs passing the constructive check procedure are correct ones with respect to the desired declarative semantics.

7 Stratified Program

Stratified program is a well-known notion in logic programming and deductive databases. It was an early attempt to deal with dependencies between relations in presence of negation. The fixpoint computation along strata gives this class of programs a natural semantics. We introduce the idea of stratification into reactive rule-based systems to achieve rule-consistency. An arbitrary declarative response is not necessarily rule-consistent since a condition (NOT x *= v) of a rule r can be disabled by firing other rules after r, which may generate $\langle x, v\rangle$. By firing rules in a *stratified* order, this kind of situation can be avoided. Working with stratified rule sets has the following effect: every time a rule which has a condition part including negation over [s*=v] is tested for being enabled, we can be sure that a rule with an assignment v to s has been fired earlier in the response (if it is included in the final fired rule set of this response at all).

Note, however, that the user needs not explicitly consider these dependencies when introducing rules. The support at compile time is supposed to check whether such a stratification exists. Given a program P and a pair $\langle x, v\rangle$, the *definition* of $\langle x, v\rangle$ is the set of rules in whose assignment part $\langle x, v\rangle$ appears.

A stratified rule-based program consists of a disjoint set of rules $P = P^1 \cup \ldots \cup P^i \cup \ldots \cup P^k$ called *strata*. If a program is stratifiable, its stratification is constructed as follows:

- If a positive pair [x*= v] appears in the trigger part or condition part of a rule from P_i, then its definition is contained within $\bigcup_{j \leq i} P_j$;
- If a negative pair [NOT x*=v] appears in the condition part of a rule from P_i, then its definition is contained within $\bigcup_{j<i} P_j$.

For a given a stratified correct program, the responses generated by such operational semantics are correct if they are slot-consistent. Unfortunately, for a stratified program this operational semantics does not guarantee slot-consistency. Stratification simply provides a sufficient condition for rule-consistency.

8 Summary

The technical results obtained in our research can be summarized as follows:

- We have defined a rule-based language RL that combines asynchronous interaction with an environment with synchronous treatment of a response. Time and concurrency are thus dealt with in a simple manner;
- For this language we have defined a declarative semantics which enables a natural treatment of causality, atomicity, and desired determinism;
- We have defined a correctness criterion for reactive RL programs. A correct program ensures termination of rule firings at each reaction, consistency of the fired rules and a unique reaction for each new set of stimuli to the system;
- We have defined and implemented constructive semantics, based on three-valued evaluation of rules, that guarantees the correct results of computations for correct programs;
- We have developed and implemented a static procedure for checking the correctness of programs;
- We have proven soundness of the obtained results;
- For stratified programs we have developed the computational support which guarantees correctness w.r.t. one particular consistency requirement.

References

1. K. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.
2. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* MA: Addison-Wesley, 1988.
3. T. A. Cooper and N. Wogrin. *Rule-based Programming with OPS5.* Morgan Kaufmann Publishers, Inc, 1988.
4. K.R. Dittrich, S. Gatziu, and A. Geppert. The active database management systems manifesto: A rulebase of ADBMS features. In Timos Sellis, editor, *Rules in Database System.* RIDS'95, Springer Verlag, 1995.
5. N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.
6. D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series.* Springer Verlag, 1985.
7. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of ACM*, 27(4):797–821, 1980.
8. M. Lin. *Formal Analysis of Reactive Rule-based Programs.* Licentiate thesis, Linköping University, 1997. Linköping Studies in Science and Technology, Thesis No 643, ISBN 91-7219-030-2, ISSN 0280-7971.
9. W. Łukaszewicz. *Non-Monotonic Reasoning.* Ellis Horwood, 1990.
10. J. Malec, M. Morin, and U. Palmqvist. Driver support in intelligent autonomous cruise control. In *Proceedings of the IEEE Intelligent Vehicles Symposium'94*, pages 160–164, Paris, France, October 1994.
11. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.
12. M. Morin, S. Nadjm-Tehrani, P. Österling, and E. Sandewall. Real-time hierarchical control. *IEEE Software*, 9(5):51–57, September 1992.

13. G. Plotkin, C. Stirling, and M. Tofte, editors. *Language and Interaction: Essays in Honour of Robin Milner*, chapter The Foundations of Esterel. MIT Press, 1998. To Appear.
14. 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.
15. E. Sandewall. *Features and Fluents*, volume 1. Clarendon Press. Oxford, 1994.

A Appendix: Syntax

The syntax for RL is defined as follows.

Definition 5. A rule is a string

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

fulfilling the requirements of the following grammar:

$\langle r_{trig} \rangle$	$::= \langle \textit{slot-name} \rangle \ \texttt{*=}\ \langle \textit{slotval} \rangle$
$\langle \textit{slotval} \rangle$	$::= \langle \textit{ident} \rangle$
$\langle r_{cond} \rangle$	$::= \langle r_{cond} \rangle \ \texttt{AND}\ \langle r_{literal} \rangle$
	$\quad \mid \langle r_{literal} \rangle$
	$\quad \mid \texttt{TRUE}$
$\langle r_{literal} \rangle$	$::= \texttt{NOT}\ \langle r_{literal} \rangle$
	$\quad \mid \langle \textit{slot-name} \rangle \ \texttt{*=}\ \langle \textit{slotval} \rangle$
	$\quad \mid \langle \textit{slot-name} \rangle \ \texttt{\|=}\ \langle \textit{slotval} \rangle$
$\langle r_{assign} \rangle$	$::= \langle \textit{assignment} \rangle \mid \texttt{\{}\ \langle \textit{assignment-list} \rangle\ \texttt{\}}$
$\langle \textit{assignment-list} \rangle$	$::= \langle \textit{assignment} \rangle \mid \langle \textit{assignment-list} \rangle\ \texttt{,}\ \langle \textit{assignment} \rangle$
$\langle \textit{assignment} \rangle$	$::= \langle \textit{slot-name} \rangle\ \texttt{:=}\ \langle \textit{slotval} \rangle$
$\langle \textit{slot-name} \rangle$	$::= \langle \textit{ident} \rangle$

where $\langle \textit{ident} \rangle$ denotes an identifier. □

This article was processed using the LaTeX macro package with LLNCS style