ГОДИШНИК НА СОФИЙСКИЯ УНИВЕРСИТЕТ „СВ. КЛИМЕНТ ОХРИДСКИ“

ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
Книга 2 — Приложна математика и информатика
Том 92, 1998

ANNUAIRE DE L'UNIVERSITE DE SOFIA „ST. KLIMENT OHRIDSKI“

FACULTE DE MATHEMATIQUES ET INFORMATIQUE
Livre 2 — Mathématiques Appliquée et Informatique
Tome 92, 1998

THE *SLRS* SYNCHRONOUS IMPERATIVE PROGRAMMING LANGUAGE

MAGDALINA V. TODOROVA

This paper describes the synchronous imperative programming language *SLRS*. After a brief overview of the language we define its behavioural semantics.

Keywords: synchronous language, reactive system, real time process

1991/95 Math. Subject Classification: main 68N15, secondary 68Q15

1. INTRODUCTION

Reactive systems are programs whose main role is to maintain an ongoing interaction with their environment, rather than to produce some final result on termination. Such systems should be specified and analysed in terms of their behaviour, i.e. the sequences of states or events they generate during their operation. A reactive program may be treated as a generator of computations which, for simplicity, we may assume to be infinite sequences of states or events [1]. Typical examples of reactive systems are real time process controllers, signal processing units, digital watches and video games. Operating system drivers and mouse interface drivers are examples of reactive programs too. Lustre [4], Esterel [2, 3], Signal [5] are programming languages devoted to program reactive systems.

Determinism is an important characteristic of reactive programs. A deterministic reactive program produces identical output sequences when fed with identical input.

In this paper a synchronous imperative programming language named *SLRS* (Synchronous Language to Reactive Systems) is considered. It is based on the synchrony hypothesis: each reaction is assumed to be instantaneous and therefore atomic in any possible sense. Control transmission, signal broadcasting, and elementary computations are supposed to take no time, making the outputs of a system perfectly synchronous with its inputs [2]. After a brief overview of the *Pure SLRS* we define its behavioural semantics.

2. THE *PURE SLRS* LANGUAGE

In this section we describe the *Pure SLRS* language intuitively and by examples.

A *SLRS* program:

program P;
    declaration part
    interface part
    body
end P.

has a *declaration part* that declares the external objects used by the program, an *interface part* that defines its input and output, and a *body* that is an executable statement.

Declaration part. Data declarations declare the constants, types, functions, and procedures that manipulate data. They are written in the host language (Pascal or C).

Interface part. The interface part

input I1 {, In};
output O1 {, On};
input relations;

defines program's input $I_1, \ldots, I_n$ and output $O_1, \ldots, O_n$ signals.

The basic object of the language is a *signal.* Signals are used for communication with the environment as well as for internal broadcast communication. There is a special signal called *tic.* It is assumed to be always present. In *Pure SLRS* there are only two kinds of interface signals: input and output signals.

Input signals come from the environment. *They cannot be produced internally.* They are declared in the form

input I1 {, In};

Output signals are directed towards the environment of the program by the *produce* statement. An output signal declaration has the form

output O1 {, On};

Input relations are assertions that can be used to restrict input events. That is very important for program specification and verification.

A *SLRS* program specifies a relation between input and output signals. It is activated by repeatedly giving it *input events*. These events consist of a possibly empty set of input signals assumed to be present. For each input event, the program reacts by executing its body and by outputting the produced output signals that form the *output event*. We assume that the reaction is perfectly synchronous and deterministic. A reaction is also called an *instant*.

The **kernel statements** in the language are:

- *Statement skip*:

  skip

It performs no action and terminates immediately.

- *Statement stop*:

  stop

It performs no action and never terminates.

- *Statement produce*:

  produce S

where S is a signal. It emits S and terminates immediately.

- *Statement sequence*:

  sequence stat_1, stat_2 end

where $stat_1$ and $stat_2$ are any statements. The statement $stat_2$ starts instantly when the statement $stat_1$ terminates. The sequencing operator takes no time by itself.

- *Statement parallel*:

  parallel stat_1, stat_2 end

where $stat_1$ and $stat_2$ are any statements. The statements $stat_1$ and $stat_2$ are started simultaneously when the parallel statement is started. The *parallel* statement terminates when its both branches are terminated.

- *Statement ifp-then-else-end*:

  ifp S then stat1 else stat2 end

where S is a signal, $stat_1$ and $stat_2$ are any statements. The *then* and *else* parts are optional. If some of them is omitted, it is supposed to be *skip* statement. The presence of S is tested and the *then* or *else* branch is immediately started accordingly.

- *Statement cycled-end*:

  cycled stat end

where *stat* is any statement. The body *stat* of a *cycled-end* starts immediately when the *cycled-end* statement starts and whenever *stat* terminates, it is instantly restarted. A *cycled-end* never terminates.

- *Statement watching-do*:

  watching S do stat end

where *stat* is any statement and S is a signal. S is called a *guard*. The statement *stat* is executed normally until *stat* terminates or until future occurrence of the signal S. If *stat* terminates just before S occurs or at the same time as S, so does the whole *watching-do* statement and the guard has no action. Otherwise, the occurrence of S provokes immediate preemption of the body *stat* and immediate termination of the whole *watching-do* statement.

Example. Let define

  await S $=_{\text{def}}$ watching S do stop end.

When *await S* starts executing, it retains the control until the first future reaction where S is present. If such a reaction exists, the *await* statement terminates immediately. Otherwise it never terminates.

Example. Let us consider the statement

```
watching I1 do
   sequence
      watching I2 do
         sequence
            await I3,
            produce O1
         end
      end,
      produce O2
   end
end
```

If *I1* occurs before *I2* and *I3* or at the same time as them, then the external *watching-do* preempts its body and terminates instantly. In this case no signal is produced. If *I2* occurs before *I3* or at the same time as it, but before *I1*, then the internal *watching* preempts its body, *O1* is not produced even if *I3* is present, *O2* is produced and the external *watching* instantly terminates. If *I3* occurs just before *I1* and *I2*, then the *await* statement terminates, *O1* is produced, the internal *watching-do* terminates since its body terminates, *O2* is produced and the external *watching* also terminates.

- *Statement run-until*:

  run stat until X

where *stat* is any statement and X is a parameter. The body *stat* starts instantly and determines the behaviour of the *run-until* statement until it terminates or executes *exit X*. Then the execution of *stat* is preempted and the whole *run-until* constructor terminates. If body of a *run-until* statement contains parallel components, the *run-until* is exited when one of the components executes an *exit X*, the other component is preempted.

Example. Let consider the statement

```
run
    parallel
        sequence
            await I1,
            produce O
        end,
        sequence
            await I2,
            exit X
        end
    end
until X
```

If *I1* occurs before *I2*, then *O* is produced and *run* waits for *I2* to terminate. If *I2* occurs before *I1*, then the whole statement terminates instantly, the first branch is preempted and *O* will never be produced. If *I1* and *I2* occur simultaneously, then both branches do execute and *O* is produced.

Run-until statement provides a way for breaking loops:

```
run
    cycled ... exit X ... end
until X
```

Notice that the statement

```
run
    sequence
        run
            parallel
                exit X,
                exit Y
            end
        until Y,
        produce O
    end
until X
```

is ambiguous. We must define what it means to exit several *run-until* statements simultaneously.

Priorities between run-until statements — only the outermost *run-until* statement matters, the other ones are discarded.

In the above example the internal *run-until* is discarded and *O* is not produced.

- *Statement local*:

  local S {, $\mathrm{S_i}$} in stat end

where S and S_i are signals and *stat* is any statement. It declares a lexically scoped signal S {, S_i} that can be used for internal broadcast communication within *stat*.

At each reaction, a signal has a single status — *present* or *absent*. The following law determines the status of local and output signals: *A local or output signal is present in a reaction if and only if it is produced by executing a produce statement in that reaction.* The default status of a signal is to be absent.

3. THE BEHAVIOURAL SEMANTICS OF THE *PURE SLRS*

This semantics defines program execution reaction by reaction using Structural Operational Semantics technique [6]. It defines transitions of the form

$$\mathrm{P} \xrightarrow[\mathrm{I,\,O}]{} \mathrm{P}',$$

where P is a program, I is an input event, O is the corresponding output event, and P' is the new program, i.e. the new state of P after reaction to I. The sequence

$$\mathrm{P} \xrightarrow[\mathrm{I_1,\,O_1}]{} \mathrm{P_1} \xrightarrow[\mathrm{I_2,\,O_2}]{} \cdots \mathrm{P_n} \xrightarrow[\mathrm{I_{n+1},\,O_{n+1}}]{} \cdots$$

defines the reaction $O_1, O_2, \ldots, O_n, \ldots$ to an input sequence $I_1, I_2, \ldots, I_n, \ldots$ The programs P_i are called *derivations of* P.

The transition

$$\mathrm{P} \xrightarrow[\mathrm{I,\,O}]{} \mathrm{P}'$$

is defined using the following auxiliary relation:

$$\mathrm{stat} \xrightarrow[\mathrm{E,\,E',\,t,\,S}]{} \mathrm{stat}',$$

where *stat* is the body of P, *stat'* is the body of P', E is the current event in which *stat* reacts, E' is the event composed of the signals produced by *stat*, t is an integer ($t \geq 0$) that codes the way in which *stat* terminates or exits, and S is a set of integers. S is called a *stopset* and t — a *termination level.* They are defined below. The current event E is composed of all signals that are present at a given reaction. By the law, which determines the state of local and output signals, E must contain the set E' of produced signals. The auxiliary relation is defined by structural induction on statements by means of inductive rules.

The connection between the transition and the auxiliary relation is as follows:

$$\mathrm{P} \xrightarrow[\mathrm{I,\,O}]{} \mathrm{P}' \text{ if stat } \xrightarrow[\mathrm{I \cup O \cup \{tic\},\,O,\,t,\,S}]{} \mathrm{stat}'$$

for some t and S.

Termination level. To determine the termination level, it is useful to label the *exit* X part of a *run-until* X statement with the corresponding level $t+2$, where t ($t \geq 0$) is an integer and is equal to the number of the *run-until* statements which one must traverse to reach the *run-until* X statement [2].

Example.

```
run
    parallel
        exit X : 2,
        run
```

```
        parallel
          exit X : 3,
          exit Y : 2
        end
      until Y
    end
  until X
```

The first *exit X* and the *exit Y* are labelled 2 since there is not intermediate *run-until* statement to traverse, while the second *exit X* is labelled 3 since one must traverse the *run-until Y* statement to reach the *run-until X* statement.

Definition. The *termination level t of a statement stat* is defined as *t(stat)*, where:

$$\begin{aligned}
&\mathrm{t(skip)} = 0,\\
&\mathrm{t(stop)} = 1,\\
&\mathrm{t(produce\ X)} = 0,\\
&\mathrm{t(sequence\ stat_1,\ stat_2\ end)} = \begin{cases} \mathrm{t(stat_1)} & \text{if } \mathrm{t(stat_1)} > 0,\\ \mathrm{t(stat_2)} & \text{if } \mathrm{t(stat_1)} = 0,\end{cases}\\
&\mathrm{t(parallel\ stat_1,\ stat_2\ end)} = \max\{\mathrm{t(stat_1)},\ \mathrm{t(stat_2)}\},\\
&\mathrm{t(cycled\ stat\ end)} = 1 \ \text{ if } \mathrm{t(stat)} = 0,\\
&\mathrm{t(cycled\ stat\ end)} = \mathrm{t(stat)} \ \text{ if } \mathrm{t(stat)} > 0,\\
&\mathrm{t(watching\ X\ do\ stat\ end)} = \mathrm{t(stat)},\\
&\mathrm{t(run\ stat\ until\ X)} = \begin{cases} 0 & \text{if } \mathrm{t(stat)} = 0 \text{ or } \mathrm{t(stat)} = 2,\\ 1 & \text{if } \mathrm{t(stat)} = 1,\\ \mathrm{i} - 1 & \text{if } \mathrm{t(stat)} = \mathrm{i},\ \mathrm{i} > 2,\end{cases}\\
&\mathrm{t(exit\ X : i)} = \mathrm{i},\\
&\mathrm{t(local\ X\ in\ stat\ end)} = \mathrm{t(stat)}.
\end{aligned}$$

The termination level of the statement of the above example is 0.

Stopset. We number all occurrences of the *stop* statement in *stat* by different integers from 0 to n, $n > 0$. A *stopset* S is a subset of $[0..n]$ that satisfies the following condition: If $stat_1$ and $stat_2$ are the two statements of a *sequence* or two branches of an *ifp-then-else-end* statement, then S cannot contain an occurrence of *stop* in $stat_1$ together with an occurrence of *stop* in $stat_2$. Notice that $S = \emptyset$ when $t \neq 1$ and $S \neq \emptyset$ when $t = 1$.

Inductive Rules:

(IR1) $\mathrm{skip} \xrightarrow[\mathrm{E},\ \emptyset,\ 0,\ \emptyset]{} \mathrm{skip};$

(IR2) $\mathrm{stop : i} \xrightarrow[\mathrm{E},\ \emptyset,\ 1,\ \{i\}]{} \mathrm{stop : i};$

(IR3) $\mathrm{produce\ X} \xrightarrow[\mathrm{E},\ \{X\},\ 0,\ \emptyset]{} \mathrm{skip};$

$$\text{(IR4)}\quad \frac{\text{stat}_1 \xrightarrow[\text{E, E}'_1,\ 0,\ \varnothing]{} \text{stat}'_1 \quad \text{and} \quad \text{stat}_2 \xrightarrow[\text{E, E}'_2,\ \text{t}_2,\ \text{S}_2]{} \text{stat}'_2}{\text{sequence stat}_1,\ \text{stat}_2 \text{ end} \xrightarrow[\text{E, E}'_1 \cup \text{E}'_2,\ \text{t}_2,\ \text{S}_2]{} \text{stat}'_2};$$

$$\text{(IR5)}\quad \frac{\text{stat}_1 \xrightarrow[\text{E, E}'_1,\ \text{t}_1,\ \text{S}_1]{} \text{stat}'_1, \quad \text{t}_1 > 0}{\text{sequence stat}_1,\ \text{stat}_2 \text{ end} \xrightarrow[\text{E, E}'_1,\ \text{t}_1,\ \text{S}_1]{} \text{sequence stat}'_1,\ \text{stat}_2 \text{ end}};$$

$$\text{(IR6)}\quad \frac{\text{stat}_1 \xrightarrow[\text{E, E}'_1,\ \text{t}_1,\ \text{S}_1]{} \text{stat}'_1 \quad \text{and} \quad \text{stat}_2 \xrightarrow[\text{E, E}'_2,\ \text{t}_2,\ \text{S}_2]{} \text{stat}'_2}{\text{parallel stat}_1,\ \text{stat}_2 \text{ end} \xrightarrow[\text{E, E}'_1 \cup \text{E}'_2,\ \max(\text{t}_1,\text{t}_2),\ \text{S}]{} \text{parallel stat}''_1,\ \text{stat}''_2 \text{ end}},$$

where $\text{S} = \begin{cases} \text{S}_1 \cup \text{S}_2, & \text{if } \max(\text{t}_1,\text{t}_2) \le 1, \\ \varnothing, & \text{if } \max(t_1,t_2) > 1 \end{cases}$ and $\text{stat}''_\text{i} = \begin{cases} \text{stat}'_\text{i}, & \text{if t}_\text{i} \ne 0, \\ \text{skip}, & \text{if t}_\text{i} = 0; \end{cases}$

$$\text{(IR7)}\quad \frac{\text{stat} \xrightarrow[\text{E, E}',\ \text{t},\ \text{S}]{} \text{stat}', \quad \text{t} > 0}{\text{cycled stat end} \xrightarrow[\text{E, E}',\ \text{t},\ \text{S}]{} \text{sequence stat}',\ \text{cycled stat end end}};$$

$$\text{(IR8)}\quad \frac{\text{X} \in \text{E and stat}_1 \xrightarrow[\text{E, E}'_1,\ \text{t}_1,\ \text{S}_1]{} \text{stat}'_1}{\text{ifp X then stat}_1 \text{ else stat}_2 \text{ end} \xrightarrow[\text{E, E}'_1,\ \text{t}_1,\ \text{S}_1]{} \text{stat}'_1};$$

$$\text{(IR9)}\quad \frac{\text{X} \notin \text{E and stat}_2 \xrightarrow[\text{E, E}'_2,\ \text{t}_2,\ \text{S}_2]{} \text{stat}'_2}{\text{ifp X then stat}_1 \text{ else stat}_2 \text{ end} \xrightarrow[\text{E, E}'_2,\ \text{t}_2,\ \text{S}_2]{} \text{stat}'_2};$$

$$\text{(IR10)}\quad \frac{\text{stat} \xrightarrow[\text{E, E}',\ \text{t},\ \text{S}]{} \text{stat}'}{\text{watching X do stat end} \xrightarrow[\text{E, E}',\ \text{t},\ \text{S}]{} \text{ifp X else watching X do stat}' \text{ end end}};$$

$$\text{(IR11)}\quad \frac{\text{stat} \xrightarrow[\text{E, E}',\ \text{t},\ \varnothing]{} \text{stat}' \quad \text{and} \quad \text{t} = 0 \text{ or t} = 2}{\text{run stat until X} \xrightarrow[\text{E, E}',\ 0,\ \varnothing]{} \text{skip}};$$

$$\text{(IR12)}\quad \frac{\text{stat} \xrightarrow[\text{E, E}',\ \text{t},\ \text{S}]{} \text{stat}' \quad \text{and} \quad (\text{t} = 1 \text{ and t}' = 1) \text{ or } (\text{t} > 2 \text{ and t}' = \text{t} - 1)}{\text{run stat until X} \xrightarrow[\text{E, E}',\ \text{t}',\ \text{S}]{} \text{run stat}' \text{ until X}};$$

$$\text{(IR13)}\quad \text{exit X : i} \xrightarrow[\text{E},\ \varnothing,\ \text{i},\ \varnothing]{} \text{stop};$$

$$\text{(IR14)} \quad \frac{X \notin E' \text{ and stat} \xrightarrow[E,\ E' \cup \{X\},\ t,\ S]{} \text{stat}'}{\text{local X in stat end} \xrightarrow[E,\ E',\ t,\ S]{} \text{local X in stat}' \text{ end}};$$

$$\text{(IR15)} \quad \frac{X \notin E' \text{ and stat} \xrightarrow[E-\{X\},\ E',\ t,\ S]{} \text{stat}'}{\text{local X in stat end} \xrightarrow[E,\ E',\ t,\ S]{} \text{local X in stat}' \text{ end}}.$$

Definition. A program is *locally correct* if its body and its substatements are such that each local and output signal can have a single status for any input event that satisfies the input relations.

Definition. A program is *correct* if all its derivations are locally correct.

Correctness obviously implies determinism. In the sequel, we will consider a correct program P. For technical reasons (see Theorem 1 below), we assume also that the body of P never terminates, adding a trailing *stop* if it is necessary. This does not change the observable behaviours.

Let *stat* be a statement, S — a stopset, and *stat'* — a derivation of *stat*. We will define term *R(stat:S)* equal to *stat'*, i.e. by means of the operator R we recover the derivation *stat'* from *stat* and S. The argument of the operator R is a term labelled S. A labelled term *stat:S* is obtained by labelling the subterms of *stat* either $S+$, or $S-$. A subterm is labelled $S+$ if and only if it contains at least one occurrence of *stop* which number is in S, otherwise, the subterm is labelled $S-$. The labels are redundant, but they make the proofs simpler to write.

Definition. R(stat:S−) = stat
R(skip:S) = skip
R((stop:i):S) = stop:i
R((produce X):S) = skip
R(sequence stat_1:S+, stat_2:S− end) = sequence R(stat_1:S+), stat_2 end
R(sequence stat_1:S−, stat_2:S+ end) = R(stat_2:S+)
R(parallel stat_1:S+, stat_2:S+ end)= parallel R(stat_1:S+), R(stat_2:S+) end
R(parallel stat_1:S+, stat_2:S− end) = parallel R(stat_1:S+), skip end
R(parallel stat_1:S−, stat_2:S+ end) = parallel skip, R(stat_2:S+) end
R(ifp X then stat_1:S+ else stat_2:S− end) = R(stat_1:S+)
R(ifp X then stat_1:S− else stat_2:S+ end) = R(stat_2:S+)
R(cycled stat:S+ end) = sequence R(stat:S+), cycled stat end end
R(watching X do stat:S+ end) = ifp X else watching X do R(stat:S+) end end
R((run stat until X):S) = run R(stat:S) until X
R((local X in stat end):S) = local X in R(stat:S) end.

Theorem 1. Let *stat* be the body of a correct program and *stat* never terminate. Let S be a stopset in *stat*. Then for any transition of the form

$$\text{R(stat:S)} \xrightarrow[E,\ E',\ 1,\ S']{} \text{stat}'$$

the stopset S' contains only stops occurring in *stat'* and *stat'* = *R(stat:S')*.

Proof. Let E is a given current event. The proof is by structural induction on *stat*. All cases are similar, so we will consider the *sequence* and the *watching-do* statements as examples.

(i) Let *stat* = *sequence* $stat_1$, $stat_2$ *end*. There are two main subcases:

— If *stat:S* = *stat:S+* = *sequence* $stat_1$*:S−*, $stat_2$*:S+ end*, then *R(stat:S)* = *R($stat_2$:S+)*. By correctness and by the hypothesis that *stat* stops, *R($stat_2$:S+)* has a unique transition

$$R(stat_2{:}S+) = R(stat{:}S) \xrightarrow[E,\ E',\ 1,\ S']{} stat',$$

where S' is a non-empty stopset that contains only stops in $stat_2$. By induction,

$$stat' = R(stat_2{:}S') \tag{1}$$

and S' contains only stops in *stat'*. Since S' is non-empty and is a stopset in $stat_2$,

$$R(stat_2{:}S') = R(sequence\ stat_1{:}S'-,\ stat_2{:}S' + end) = R(stat{:}S'). \tag{2}$$

The result is achieved as a consequence of (1) and (2).

— If *stat:S* = *stat:S+* = *sequence* $stat_1$*:S+*, $stat_2$*:S− end*, then *R(stat:S)* = *sequence R($stat_1$:S+)*, $stat_2$ *end*. By correctness and by the hypothesis that *stat* stops, *R($stat_1$:S+)* has a unique transition

$$R(stat_1{:}S+) \xrightarrow[E,\ E',\ 1,\ S']{} stat'_1,$$

where S' is a non-empty stopset that contains only stops in $stat_1$. By induction,

$$stat'_1 = R(stat_1{:}S') \tag{3}$$

and S' contains only stops in $stat'_1$. By (IR5) we have

$$\begin{aligned} &sequence\ R(stat_1{:}S+),\ stat_2\ end \xrightarrow[E,\ E',\ 1,\ S']{} \\ &\qquad sequence\ stat'_1,\ stat_2\ end = stat'. \end{aligned} \tag{4}$$

From (3) and (4)

$$\begin{aligned} stat' &= sequence\ stat'_1,\ stat_2\ end = sequence\ R(stat_1{:}S'),\ stat_2\ end \\ &= R((sequence\ stat_1,\ stat_2\ end){:}S') = R(stat{:}S') \end{aligned}$$

and the result is achieved.

(ii) Let *stat* = *watching X do* $stat_1$ *end*. There are also two main subcases:

— If *stat:S* = *stat:S−*, then *R(stat:S−)* = *stat*.

By correctness and by the hypothesis that *stat* stops, $stat_1$ has a unique transition

$$R(stat_1{:}S) = stat_1 \xrightarrow[E,\ E',\ 1,\ S']{} stat'_1,$$

where S' is a non-empty stopset that contains only stops in $stat_1$. By (IR10) we have

$$stat \xrightarrow[E,\ E',\ 1,\ S']{} ifp\ X\ else\ watching\ X\ do\ stat'_1\ end\ end = stat'.$$

By induction,

$$stat'_1 = R(stat_1{:}S'),$$

and by the fact that S' is a non-empty stopset that contains only stops in $stat_1$,

$$\begin{aligned} \mathrm{stat'} &= \mathrm{ifp\ X\ else\ watching\ X\ do\ stat'_1\ end\ end} \\ &= \mathrm{ifp\ X\ else\ watching\ X\ do\ R(stat_1 : S'+)\ end\ end} \\ &= \mathrm{R(stat : S')}. \end{aligned}$$

— If $stat{:}S = stat{:}S+$, then $R(stat{:}S+) =$ *ifp X else watching X do* $R(stat_1{:}S+)$ *end end*. By correctness and by the hypothesis that *stat* stops, $R(stat_1{:}S+)$ has a unique transition

$$\mathrm{R(stat_1{:}S+)} \xrightarrow[\mathrm{E,\ E',\ 1,\ S'}]{} \mathrm{stat'_1},$$

where S' is a non-empty stopset that contains only stops in $stat_1$. By induction,

$$\mathrm{stat'_1} = \mathrm{R(stat_1{:}S')}$$

and S' contains stops in $stat'_1$. By (IR10) and (IR9) ($X \notin E$) we have

$$\begin{aligned} \mathrm{R(stat{:}S+)} &= \mathrm{ifp\ X\ else\ watching\ X\ do\ R(stat_1{:}S+)\ end\ end} \\ &\xrightarrow[\mathrm{E,\ E',\ 1,\ S'}]{} \mathrm{ifp\ X\ else\ watching\ X\ do\ stat'_1\ end\ end} = \mathrm{stat'}. \end{aligned}$$

Then

$$\mathrm{stat'} = \mathrm{ifp\ X\ else\ watching\ X\ do\ R(stat_1{:}S')\ end\ end} = \mathrm{R(stat{:}S')}.$$

Theorem 2. Let P be a correct program and *stat* be its body. Then any derivation *stat'* of *stat* is equal to *R(stat:S)* for some stopset S and there are only finitely many derivations.

Proof. We shall use induction on the length of a transition sequence. Let the derivative *stat'* of *stat* be produced by means of the following sequence:

$$\mathrm{stat} = \mathrm{stat_1} \xrightarrow[\dots]{} \cdots \xrightarrow[\dots]{} \mathrm{stat_n} \xrightarrow[\mathrm{E_n,\ E'_n,\ 1,\ S_n}]{} \mathrm{stat'}.$$

If $n = 0$, $stat' = stat = R(stat{:}\varnothing)$ and the result is achieved.

Let $stat_n = R(stat{:}S')$ for some stopset S'. Then

$$\mathrm{R(stat{:}S')} \xrightarrow[\mathrm{E_n,\ E'_n,\ 1,\ S_n}]{} \mathrm{stat'}.$$

By Theorem 1,

$$\mathrm{stat'} = \mathrm{R(stat{:}S_n)}$$

and the result is achieved.

The finiteness property is obvious since there are only finitely many possible stopsets in *stat*.

We can therefore completely replace a program P by its reaction graph considered as a finite state automaton with derivatives as states.

REFERENCES

1. Manna, Z., A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer — Verlag, 1991.

2. Berry, G., G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation.
`http://www.time-rover.com/reactive.html/islip_ps.gz`.
3. Berry, G., S. Ramesh, R. Shyamasunder. Communicating Reactive Processes.
`http://www.time-rover.com/reactive.html`.
4. Halbuachs, N., J-C. Fernansez, A. Bouajjani. An executable temporal logic to express safety properties and its connection with the language Lustre. Proc. in ISLIP'93, 1993.
5. Guernic, P., M. Borgne, T. Gauthier, C. Maire. Programming real time applications with Signal. Proc. of the IEEE, 1991.
6. Plotkin, G. A Structural Approach to Operational Semantics. Technical Report D AIMI FN-19, University of Aarhus, 1981.

Received November 9, 1998
Revised January 15, 1999

Faculty of Mathematics and Informatics
"St. Kliment Ohridski" University of Sofia
5 James Bourchier Blvd.
BG-1164 Sofia, Bulgaria
E-mail: `magda@fmi.uni-sofia.bg`