Formal semantics and verification of distributed systems presented by Basic–REAL specifications*

V.A. Nepomniaschy, N.V. Shilov, E.V. Bodin

The specification language Basic–REAL consists of executional and logical specification sublanguages. The first one is based on SDL and differs from it by multiple clock concept, time intervals associated with actions, non-determinism, and rich collection of channel structures. The second one is based on temporal and dynamic logics extended by time intervals. The language includes fairness conditions. The structural operational semantics of Basic–REAL is given by means of transition systems. Throughout the paper an example “Passenger and Slot–Machine” is considered. Our approach to verification of Basic–REAL specifications is based on inductive proof principles supported by model-checking and exploits refinement, fairness conditions, and time constraints.

Introduction

The role of formal description techniques in development of distributed systems increases since such systems become more complex and require more efforts for their documentation, testing and verification. In this connection it is sufficient to refer to the development of the well-known specification language SDL. But despite a number of merits making SDL popular, this language has some shortcomings. Firstly, SDL has no means for describing properties – either basic ones, like deadlock, starvation, etc., or more complex ones. Secondly, development of formal semantics lags behind the language development; still there is no universally recognized, mathematically strict description for semantics of the latest versions of SDL.

To overcome the first shortcoming, some authors ((4, 10, 11)) use external formalisms for describing properties. To overcome the second one, a number of authors ((3, 7, 11, 14)) extract some fragment of SDL syntax, sometimes varying its informal semantics; and for this fragment they build complete formal semantics.

The starting point of our research was the original version of REAL [12] consisting of a SDL-like executional specification language and a logical specification language based on temporal logic CTL [6]. A formal syntax,

* Partially supported by INTAS–RFBR under Grant 95–0378.
informal semantics and a sketch of formal semantics of REAL were described in [12].

Logical specification language of REAL cannot be strictly reduced to CTL, it is an extension of CTL with time intervals and first-order dynamic logic constructions [8].

Three main approaches are used to model real-time systems: discrete-time, fictitious-clock, and dense-time. Our approach to real-time is a multiple clock variant of the fictitious-clock approach [15] when a special tick transition counts time steps.

However, REAL semantics was rather complicated. This is why the idea arose at first to construct a simplified level of REAL and then to develop its complete formal semantics.

In [13] a simplified level of REAL called Basic–REAL was introduced and its formal semantics was presented. As compared with REAL, the Basic–REAL uses fairness conditions, predefined types as well as abstract data types, predefined structures and semi-abstract channel structures. We express structural operational semantics [16] of Basic–REAL specifications by means of transition systems. Basic–REAL is a verification-oriented version of REAL. This allows us to use Basic–REAL for verification of distributed systems.

The Basic–REAL language is intended to the representation of finite as well as parameterized and infinite systems (the parameterized example “Passenger and Slot–Machine” see below). Parameterized and infinite systems cannot be automatically verified by a straightforward application of the model-checking method. This is why we combine model-checking with inductive reasoning in our example.

We consider proving distributed systems properties in style of [9]:

- classify properties into classes of problems with respect to the syntactical structure of the corresponding logical specifications,
- formulate and validate problem-oriented proof principles for arbitrary transition systems,
- apply the problem-oriented proof principles to the transition systems generated by the operational semantics of the verified executable specifications.

In practice, the third item requires preliminary simplification of the verified executable specifications so that the verified specifications are a refinement of the simplified specifications. In the framework of presented approach, the verification procedure of the simplified specifications is semi-automatic: general proof outlines are designed manually but the application of problem-oriented proof principles is supported by model-checking.
The rest of the paper consists of five sections and Appendix. The general concepts of the specification language are stated in Section 1. The main constructs of Basic−REAL language are explained with the help of an example (Passenger and Slot−Machine) in Section 2. Foundations of Basic−REAL semantics and logical specifications semantics are described in Section 3. In Section 4 semantics of executional specifications is explained. The verification example is considered in Section 5. In conclusion the results and further research directions are discussed. Appendix contains an example of Basic−REAL executable specification.

1. General concepts of Basic−REAL

Basic−REAL language consists of executional and logical specifications sublanguages.

The executional specification language is intended for describing the structure of distributed systems, while the logical specification language describes their properties.

Basic−REAL has a two-level hierarchy: an executional specification is a process or a block consisting of processes; a logical specification is a predicate or a formula consisting of predicates. Basic−REAL allows only local communication via channels. Channels are intended for signals with parameter passing. The channels themselves are semi−abstract data structures: they are not abstract data structures in general, there exist some restrictions for their interpretation, but these restrictions permit some standard data structures such as queue, stack, multiset (bag), and so on. These standard data structures are predefined in the language. Similarly, Basic−REAL exploits the abstract data types for variables and parameters of signal, but there exist some predefined types (e.g., integer) and type constructors (e.g., array). Properties of non−predefined data structures can be specified by means of logical specifications. The elementary properties that may be expressed by means of the logical specifications, are relations on variables and signal parameters, control state locators, emptiness and overfull controllers, presence and readiness checkers for signals in channels. The formulae are constructed from the elementary properties by means of propositional combinations, variable quantification, temporal interval modalities (□ and ◇), and behaviour modalities (EACH and SOME).

The behaviours of the executional specifications may be restricted by the fairness conditions. The fairness conditions mean that we consider not all behaviour space, but only the behaviours in which the fairness conditions hold infinitely often.

A specification (both executional and logical one) consists of a head, a scale, a context, a scheme, and subspecifications.
The head of the specification determines its name and kind: the executional specification is a process (PROC) or a block (BLCK), and the logical one is a predicate (PRED) or a formula (FORM). The processes and the predicates are elementary specifications, and the blocks and the formulae are composite specifications consisting of processes and predicates, respectively.

The scale of the specification is a finite set of homogeneous linear (in)equalities with positive integer coefficients on uninterpreted time units, and the special symbol $\infty$ can be used for infinity.

The context of the specification is a finite set of type definitions and object (variable and channel) declarations.

There are some predefined types (at least, integer numbers) and some type-constructors (at least, arrays).

The scheme of a block consists of fairness conditions and channel routes connecting its subspecifications (i.e., processes) to each other and to the external environment. The scheme of a process consists of fairness conditions and a process diagram. The scheme of a formula is somewhat average of what in mathematical logic is called formula and formula scheme (see the example below). And the scheme for a predicate is its name with the substituted actual parameters.

Remarks: (1) we do not require all objects of all specifications to be declared in the context of the specification; (2) all states, variables, channels with their parameters are identified in logical specifications by their extended names which consist of the process name and its own name separated by point, e.g., $gp.station$ is a variable station from a process $gp$.

2. Basic–REAL specifications of distributed systems

In order to illustrate the general concept of the language Basic–REAL, let us consider the following example: Passenger is buying a railway ticket in an automatic booking-office (Slot–Machine).

A description of the example.
The passenger can:
- see the remaining sum on the indicator,
- receive the coins returned by the Slot–Machine from the special change window,
- drop coins into the slot,
- get a ticket with a station name from the special booking window,
- press buttons with station names or commands of ticket request/cancellation.
The Slot–Machine can:

- get from the Passenger a station name or a command of ticket cancel-
  ling, or requesting a ticket through the buttons.
- show the remaining sum on the indicator,
- return all dropped coins through the change window,
- issue a ticket with a station name printed on it,
- receive coins through the slot.

Suppose that the Slot–Machine can execute 30–100 operations per sec-
ond, while the Passenger is slower, he/she can make 1–20 operations (actions
or decisions) per minute.

There are coins with nominals 1, 5, 10, 20 and 50, and there are three
stations: a, b and c.

We will call the Passenger ‘good’ if he/she follows a natural protocol of
buying a ticket to a desired station, otherwise we will call the Passenger
‘bad’.

Informally, the natural protocol of buying a ticket by the ‘good’ Pass-
enger is as follows. The good Passenger presses the button with a station
name and, while looking at the indicator, drops coins (having enough coins
of all nominals). Once the indicator shows 0, the ‘good’ Passenger presses
the request ticket button and then gets the ticket.

An example of ‘bad’ Passenger is as follows: he/she may try to get the
ticket while the indicator is not 0 yet, but when it is 0, he/she keeps dropping
coins.

We will consider two properties:

- “progress property”: the ‘good’ Passenger is sure to get the ticket,
- “safety property”: the ‘bad’ Passenger will never get the ticket.

**Formal specification.** Let us specify this informally described system
as execution specifications of the Basic–REAL language. It will be pre-

tented by two blocks containing specifications of the protocol for the ‘good’
and the ‘bad’ Passengers.

The first block consists of two processes: the good–passenger and the
Slot–Machine. The head of the block is `good-passenger_and_slot-machine`:

```
BLCK.
```

The scale of the block deals with the time units, such as a minute
(min), a second (sec), Passenger’s ingenuity (ing) and Slot–Machine opera-
tions (opr), where 1 min = 60 sec; 1 ing ≤ 1 min ≤ 20 ing; 30 opr
≤ 1 sec ≤ 100 opr.

The context begins with the type definitions. Let us describe them in
Pascal style: `value = {1, 5, 10, 20, 50}`; `region = {a, b, c};`
The declaration of a channel shows whether it is an INPut channel (i.e., going from the environment to one of the block processes), OUTPut channel (from a process to the environment), or INNer channel (from one process to another); its capacity and structure, its name and the set of possible signals with their lifetimes, names and types of their parameters.

For example, the inner channel buttons is a so-called elementary buffer (i.e., one-element queue) with the possible signal a button, the parameter station of the type region and the following possible signals: cancel (to return all coins dropped so far) and request (to give the ticket) without parameters. All these signals have the lifetime “until a request” (i.e., they are removed from the channel after the first reading). Then the inner channels follow: the unbounded queue booking with the signal ticket with parameter station of type region, the unbounded queue slot with signal coin with parameter nominal of type value, the unbounded queue change with signal coin with parameter nominal of type value, elementary buffer indicator with signal info with integer parameter left.

Then there is the scheme of the block good-passenger and slot-machine (Figure 1) and the subspecifications: the processes Slot-Machine and good-passenger.

Two processes “inherit” the scale and the context of the block to which they belong with the obvious correction: for example, the channel buttons in the process Slot-Machine becomes input (INP role) in the process Slot-Machine, and output (OUT role) in the process good-passenger. The contexts of the processes also contain variable declarations. For example, the context of the process Slot-Machine defines the following variables: expenses, natural-valued array indexed by region, for the prices to the corresponding station; coins, natural-valued array indexed by value, for counting the coins of corresponding nominal received from the Passenger; station ranging over region, for storing the name of the station that the Slot-Machine has accepted; left, the sum to be received; and nominal, the nominal of the last received coin.

The schemes of the processes Slot-Machine and good-passenger are presented in SDL-like graphical form in Figure 2 and in Figure 3, respectively.
Here, a name of a state in frame \( \square \) means that in a transition marked with this state an execution of non-deterministic program is done, \( \rightarrow \) means sending of a signal, \( \leftarrow \) means acceptance of a signal, and \( \Rightarrow \) means cleaning a channel.

Final states (they do not mark transitions but present in some JUMPs of transitions) are framed with \( \square \).

Arrows denote possible JUMPs from the states. An arrow directed down to its unique successor may be omitted.

When there are no explicit temporal restrictions on the behaviour of the Passenger and the Slot–Machine, it is necessary to use the explicit fairness conditions stating that the Slot–Machine cannot stay forever at a state (with the exception of waiting for the Passenger's actions). For the ‘timed’ Slot–Machine and Passenger, the fairness conditions are unnecessary, since they are provided by the temporal restrictions on the inner actions of the Slot–Machine and the Passenger.

For simplicity and readability, let us use the abbreviations “gp” and “sm” for "good–passenger" and “Slot–Machine”, respectively, in the extended names in the following examples and in the Section 4.
Timeless progress property 1: the ‘good’ Passenger is sure to get the ticket. It is expressed as follows:

\[(\text{AT } \text{gp.start}) \& (\text{AT } \text{sm.start}) \& (\text{EMP buttons}) \Rightarrow (\text{ticket}(\text{gp.station}) \in \text{booking})\].

Here \((\text{AT } \text{gp.start}) \& (\text{AT } \text{sm.start})\) means that the initial states of the Passenger and the Slot–Machine belong to the set of active states, \((\text{EMP buttons})\) denotes the emptiness of the channel buttons, \((\text{ticket}(\text{gp.station}) \in \text{booking})\) denotes that the signal ticket (with the parameter equal to the value of variable \(\text{gp.station}\)) is in the channel booking.

The \(\Rightarrow\) symbol is an abbreviation for the construct \(\Rightarrow \text{EACH good-passenger\_and\_slot\_machine} \Diamond \text{FROM NOW UNTIL } \infty\) which means that if the formula before this symbol is true then for each (EACH) behaviour of the block good-passenger\_and\_slot\_machine there exists a time moment \((\Diamond)\) between zero (FROM NOW) and infinity (UNTIL \(\infty\)), such that the formula after the symbol \(\Rightarrow\) is true.

When the time interval of a transition is FROM NOW UNTIL \(\infty\), we omit it.

Timed progress property 1: the ‘good’ Passenger will get the ticket by the (pre-known) time moment \(T = \text{Const} \times (\text{MaxTicketPrice} / \text{MinCoinValue}) \text{ min where min means minutes}\). It can be obtained from timeless property 1 by replacing the symbol \(\infty\) by \(T\).

Safety property 2: the ‘bad’ Passenger will never get the ticket.

\[(\text{AT bp.start}) \& (\text{AT sm.start}) \& (\text{EMP booking}) \Rightarrow \text{EACH bad-passenger\_and\_slot\_machine} \Box \text{FROM NOW UNTIL } \infty (\text{EMP booking}),\] where EACH bad-passenger\_and\_slot\_machine means “for each behaviour of the block bad-passenger\_and\_Slot\_Machine”, \(\Box \text{FROM NOW UNTIL } \infty\) means “for each time moment between zero and infinity”.
3. Foundations of Basic–REAL semantics

In order to define semantics of specifications (both executational and logical) of Basic–REAL, it is necessary to define the concept of a model with data structures and the concept of configuration. After that, a set of all behaviours in the space of configurations (i.e., countable sequences of configurations) will be associated with every executational specification and a truth set will be associated with every logical specification in the space of configurations.

A model with data structures is a triple \( M = (\text{DOM}, \text{INT}, \text{DTS}) \), where a non-empty set \( \text{DOM} \neq \emptyset \) is the domain of the model (i.e., the union of all the data type domains defined in the specification); \( \text{INT} \) is the interpretation of relation and operation symbols from the specification by the relations and the operations over \( \text{DOM} \); and \( \text{DTS} \) is a finite set of data structures for the channels defined in the specification.

The data structure is a set of finite oriented graphs, each of which is marked with a pair (signal, parameter), where parameter \( \in \text{DOM} \). Two relations \( \text{EMP} \) and \( \text{FUL} \), and two partial operations \( \text{PUT} \) and \( \text{GET} \) are defined for the data structure \( \text{DAT} \), such that: \( \text{PUT} : (\text{DAT} \times \text{SIG} \times \text{DOM}) \rightarrow \text{DAT} \), \( \text{dom(PUT)} = (\text{DAT} \setminus \text{FUL}) \times \text{SIG} \times \text{DOM} \), \( \text{val(PUT)} = \text{DAT} \setminus \text{EMP} \), and \( \text{GET} : \text{DAT} \rightarrow (\text{DAT} \times \text{SIG} \times \text{DOM}) \), \( \text{dom(GET)} = \text{DAT} \setminus \text{EMP} \), \( \text{val(GET)} = ((\text{DAT} \setminus \text{FUL}) \times \text{SIG} \times \text{DOM}) \).

For a concrete data structure \( \text{DAT} \), a graph \( \text{st} \in \text{DAT} \setminus \text{FUL} \), a signal \( sg \in \text{SIG} \), and an element \( el \in \text{DOM} \), a graph \( \text{PUT}(\text{st}, sg, el) \) is constructed by adding a new vertex and several new edges connecting the new vertex with the old ones, and marking the new vertex with the pair \( (sg, el) \). For a data structure \( \text{DAT} \), a graph \( \text{st} \in \text{DAT} \setminus \text{EMP} \), \( \text{GET}(\text{st}) \) is a triple \( (\text{st}', sg, el) \), where \( \text{st}' \) differs from \( \text{st} \) by the absence of the vertex with all the edges connecting it with others, so that \( (sg, el) \) is the mark of the removed vertex in \( \text{st} \). (The rules of removing are determined by the structure \( \text{DAT} \) itself.)

For example, if the data structure is a queue, then \( \text{DAT} \) is a set of all finite sequences of pairs (signal, parameter). The relation \( \text{EMP} \) is true on the empty sequence, the relation \( \text{FUL} \) is always false.

Let us fix a model with data structures \( M = (\text{DOM}, \text{INT}, \text{DTS}) \). For simplicity, let \( \text{DTS} \) consist of the only data structure \( \text{DAT} \), i.e., \( \text{DTS} = \{\text{DAT}\} \).

The configuration space \( \text{SPC}_M \) (or \( \text{SPC} \), while \( M \) is fixed) is a set of configurations \( \text{CNF} \), i.e., quadruples \( (T, V, C, S) \), where

- \( T \) is a value of the multiple clock (see below);
- \( V \) is an evaluation of variables, i.e., mapping which connects every variable with its current value from \( \text{DOM} \);
• \( C \) is a current content of channels, i.e., mapping which associates every channel with a marked oriented graph from \( DAT \);

• \( S \) is a current control state (i.e., mapping \( DEL \) which connects each state with its current delay presented by local clocks) and the set \( ACT \) of current active states.

Let us fix a scale. Let \( unit_1, \ldots, unit_n \) be all time units occurring in the scale. Then the scale is a system of homogeneous linear inequalities with the variables \( unit_1, \ldots, unit_n \). The integer positive solutions of this system will be called the speeds (of the clocks for the time units \( unit_1, \ldots, unit_n \)). The observation of the multiple clock \( T \) is the vector \( (t, t_1, \ldots, t_n) \) of non-negative integers such that \( t_1 = t/m_1, \ldots, t_n = t/m_n \), where \( m_1, \ldots, m_n \) are the speeds of the clocks for \( unit_1, \ldots, unit_n \) and \( \div \) is the integer division. If \( T = (t, t_1, \ldots, t_n) \) is the observation of the multiple clocks, then \( t \) is called the global time, \( t_1 \) is the time of the clock for \( unit_1 \), \ldots, and \( t_n \) is the time of the clock for \( unit_n \). For the observations of the multiple clocks \( T_1 \) and \( T_2 \) we will write \( T_1 \leq T_2 \) iff the value \( t_1 \) of the global clock in \( T_1 \) is less than or equal to the value \( t_2 \) of the global clock in \( T_2 \).

For example, let the scale be \( 10 \text{ tact} \leq 11 \text{ tick}, 10 \text{ tick} \leq 11 \text{ tact} \), \( 60 \text{ sec} = 1 \text{ min} \). The time units occurring in the scale are \( \text{tact}, \text{tick}, \text{sec} \) and \( \text{min} \). Then \( m_1 = (100, 102, 10^3, 6 \times 10^4) \) and \( m_2 = (105, 101, 20, 1200) \) are possible variants of the speeds of the clocks, while \( m_3 = (100, 111, 10^3, 10^5) \) is not, because the inequation \( 10 \times 111 \leq 11 \times 100 \) and the equation \( 60 \times 10^4 = 10^3 \) are wrong. The vector \((532, 5, 5, 0, 0)\) as well as the vector \((908, 9, 8, 0, 0)\) may be an observation of the multiple clock in the model with speeds \( m_1 \), while the vector \((908, 9, 9, 15, 0)\) cannot be an observation because \( 908/102 \neq 9 \).

If a variable is declared in a specification with a type, then \( DOM \) must include this type and the values of this variable must always belong to this type. Similarly, if a channel is declared in a specification with a structure and capacity, then the corresponding \( DAT \) must be a graphical representation of this declaration.

Let us define semantics of logical specifications. For any configuration \( CNF \) and any logical specification \( SPC \), the fact that the configuration belongs to the truth set of the logical specification \( SPC \) is denoted by \( CNF \models SPC \), and its negation is denoted by \( CNF \not\models SPC \). In order to shorten the description of semantics of logical specifications, let us fix a configuration \( CNF = (T, V, C, S) \) where \( S = (DEL, ACT) \). The relation \( CNF \models \) is defined by induction on the structure of the scheme of logical specification \( SPC \).

**Induction basis:** \( SPC \) is a predicate. If \( SPC \) is a relation, then its scheme (in prefix form) is \( R(t_1, \ldots, t_2) \), where \( R \) is a relation symbol, and \( t_1, \ldots, t_2 \) are terms constructed from operation symbols, variables and parameters of channels. Then \( CNF \models SPC \) iff \( VAL_{CNF}(t_1), \ldots, VAL_{CNF}(t_2) \)
are in the relation INT(R) where VALCNF(t₁), ..., VALCNF(t₂) are determined according to the ordinary rules.

If SPC is a locator, then its scheme has the form AT state. Then CNF |= SPC ⇔ state ∈ ACT.

If SPC is a controller, then its scheme has the form EMPchan or 0VFchan. Then CNF |= SPC ⇔ EMP(C(chan)) or CNF |= SPC ⇔ FUL(C(chan)).

If SPC is a checker, then its scheme has the form sig ∈ chan or sig ⊠ chan. Then in the first case: CNF |= SPC ⇔ ∃ val ∈ DOM such that ∃ (sig, val) ∈ C(chan); in the second case: CNF |= SPC ⇔ ∃ graph ∈ DAT and val ∈ DOM, such that GET(C(chan))=(graph, sig, val).

**Induction step.** If the scheme of SPC is a name of a predicate pred, then CNF |= SPC ⇔ CNF |= pred.

If the scheme of SPC is a propositional combination, then its value is determined in the natural way. For example, if the scheme of SPC has the form ¬F, where F is the scheme of a formula, then CNF |= SPC ⇔ CNF |≠ SPF, where SPF differs from SPC by the scheme only, which is F.

If the scheme of SPC is ∀z.F (∃x.F), where F is the scheme of a formula, then CNF |= SPC ⇔ for each (resp., some) configuration CNF' differing from CNF at most by the evaluation of variable x, the following holds: CNF' |= SPF, where SPF differs from SPC by the scheme only, which is F.

If the scheme of SPC is M₁ SYS M₂ DURATION A, where M₁ is modality EACH or SOME, SYS is an executional specification, M₂ is modality ⊥ or ⊤, duration is a time interval, and A is a scheme of a formula, then EACH means “for each fair behaviour”, SOME means “there exists a fair behaviour”, ⊥ means “for all time moments”, ⊤ means “there exists a time moment”.

For example: CNF |= EACH SYS ⊤ duration A holds iff for each behaviour of the executional specification SYS starting from the configuration CNF, there exists a time moment T' ∈ duration, such that CNF' |= SPF holds, where CNF' is a configuration in the behaviour in which T' is the observation of the multiple clock, and SPF differs from SPC by the scheme only, which is F.

4. **Semantics of executional specifications**

The semantics of executional specifications will be discussed in terms of events and step rules. Basic–REAL language has five kinds of events:

- WRT, putting a signal with parameters into a channel (WRiTIng),
- RDN, getting a signal with parameters from a channel (ReaDiNg),
- CLEAN, cleaning a channel,
- EXE, program execution,
• INVIS, an INVISible event (a clock tick without changing the channels, variables, and states).

A step rule has the form CND $\models CNF_1 < event > CNF_2$, or

$$
\frac{\text{CND}}{CNF_1 < event > CNF_2}
$$

where CND is a condition on the configurations $CNF_1$ and $CNF_2$ and on the event. An intuitive semantics of the step rule is as follows: if the condition CND holds, then the executional specification can be transformed by the event from the configuration $CNF_1$ into the configuration $CNF_2$. A countable sequence of configurations is a behaviour of an executable specification iff for each successive pair $CNF_1$ and $CNF_2$ from the sequence there exist an event and a condition CND, so that CND $\models CNF_1 < event > CNF_2$ is an instance of appropriate step rule.

For blocks there is a unique step rule, namely, the composition rule. Informally, a behaviour of a block is a simultaneous behaviour of all its processes with interleaving access to channels. Formally, let a block B contain processes $P_1, \ldots, P_k$ as its subblocks. Then

**RULE 0 (Composition)**

for all $i = 1, \ldots, k$ $CNF_1 < event/P_i > CNF_2$ 

$$
\frac{CNF_1 < event > CNF_2}
$$

where event/$P_i$ for each process $P_i$ is the event itself, if the event is either reading from an input channel or writing a signal with parameters into an output channel of the process $P_i$ or cleaning an output channel of the process $P_i$, and it is INVIS otherwise.

The remaining nine step rules deal with processes and an external environment. To be short, let us fix a process and a pair of configurations $CNF_1 = (T_1, V_1, C_1, S_1 = (ACT_1, DEL_1))$, $CNF_2 = (T_2, V_2, C_2, S_2 = (ACT_2, DEL_2))$.

For any value of the multiple clock $T$ and any interval we shall say that the value of the multiple clock belongs to the interval, iff it does not exceed either left or right bounds of the interval.

The first rule for process is a stutter rule. Informally, it concerns the case when nothing changes in the process, except the value of the multiple clock and the delay counter of the active state.

The second and third rules deal with deadlock and stabilization. Deadlock rule means that in appropriate time intervals (specified by the process diagram) the process has failed to fulfill reading or writing, i.e., a long starvation has lead to the deadlock. Stabilization rule means that the process is
in a state which marks no transition on the process diagram, so the process stabilizes. The first three rules deal with the event INVIS.

The fourth and fifth rules deal with the process reading a signal with a parameter from an input channel and writing a signal with a parameter into an output channel, respectively.

The sixth and seventh rules deal with appearing of a new signal with a parameter in an input channel and with disappearing of a signal with a parameter from an output channel.

The eighth one is the rule of cleaning a channel. The ninth rule for process is the rule of program execution. It is represented by the binary relation IO (Input-Output) on the set of variable values.

Let us consider the following rules: stuttering and reading.

At first we formulate conditions that we will use in the step rules. Note that all quantifications are made over all process variables.

\[
\begin{align*}
\text{TIME.CONST} & \quad T_1 = T_2 \\
\text{TIME.STEP} & \quad T_1 \leq T_2 \\
\text{VAR.CONST} & \quad \forall x. V_1(x) = V_2(x) \\
\text{VAR.STEP}(x) & \quad \forall y \neq x. V_1(y) = V_2(y) \\
\text{CHAN.CONST} & \quad \forall \text{chan}. C_1(\text{chan}) = C_2(\text{chan}) \\
\text{CHAN.STEP}(\text{chan}) & \quad \forall \text{chan}' \neq \text{chan}: C_1(\text{chan}') = C_2(\text{chan}') \\
\text{CHAN.HEAD}(\text{chan}, \text{sig}, x) & \quad \text{GET}(C_1(\text{chan})) = (C_2(\text{chan}), \text{sig}, V_2(x)) \\
\text{DEL.ZER} & \quad \forall \text{state}. \text{DEL}_2(\text{state}) = 0 \\
\text{DEL.NOT-OUT} & \quad \forall \text{state}. \text{if } \text{state} \in \text{ACT}_1, \text{then there is a transition in the process diagram state body interval jump}, \text{so that } \text{DEL}_1(\text{state}) \text{ does not exceed the right bound of the interval.} \\
\text{DEL.IN}(\text{state}, \text{interval}) & \quad \text{DEL}_1(\text{state}) \in \text{interval.} \\
\text{DEL.PROGR} & \quad \forall \text{state}. \text{if } \text{state} \in \text{ACT}_1, \text{then } \text{DEL}_2(\text{state}) = \text{DEL}_1(\text{state}) + T_2 - T_1, \text{else } \text{DEL}_2(\text{state}) = 0. \\
\text{ACT.CONST} & \quad \forall \text{state} \text{.state} \in \text{ACT}_1 \Leftrightarrow \text{state} \in \text{ACT}_2. \\
\text{ACT.UNIQUE} & \quad \exists \text{ unique state.state } \in \text{ACT}_1. \\
\text{ACT.ACT}(\text{state}) & \quad \text{state} \in \text{ACT}_1, \forall \text{state}' \neq \text{state}. \text{state}' \notin \text{ACT}_1. \\
\text{ACT.NEXT}(\text{next}) & \quad \text{next} \in \text{ACT}_2, \forall \text{state}' \neq \text{next}. \text{state}' \notin \text{ACT}_2. \\
\text{RTR}(\text{state}, \text{sig}, x, \text{chan}, \text{interval}, \text{next}) & \quad \text{the process diagram contains the transition state READsig(x) FROM chan interval JUMP Set, where Set is the set of states such that next } \in \text{Set.} \\
\end{align*}
\]

Now we can formulate the step rules.

**RULE 1 (Stuttering)***

\[
\begin{align*}
\text{TIME.STEP, VAR.CONST, CHAN.CONST, ACT.UNIQUE, ACT.CONST, DEL.PROGR, DEL.NOT-OUT} \\
\text{CNF1 < INVIS > CNF2}
\end{align*}
\]
RULE 4 (Reading)

\[
\text{TIME.CONST, DEL.ZER, } \exists \text{ state, interval, next: VAR.STEP}(x), \\
\text{CHAN.HEAD}(\text{chan}, \text{sig}, x), \text{ CHAN.STEP}(\text{chan}), \text{ ACT.ACT}(\text{state}), \\
\text{ACT.NEXT}(\text{next}), \text{ DEL.IN}(\text{state}, \text{interval}), \text{ RTR}(\text{state, sig, z, chan,} \\
\text{\quad interval, next}) \\
\]

\[
\text{CNF1 } \prec \text{ RDN(chan, sig, z) } \succ \text{ CNF2}
\]

5. Verification of progress properties

5.1. The proving method

In the framework of our approach we consider verification as proving properties (presented by logical specifications) of systems presented by executional specifications. Let us illustrate proving the timeless progress property 1 for the system good-passenger_and-slot-Machine with fairness conditions. We will apply the approach of [9] and then exploit the refinement for proving the same property for the timed variant of the same system. So the variant of the system with the fairness conditions is a simplification of the timed variant of the same system.

When being applied to our example, the refinement technique gives the following. Because of the time restrictions, the original (timed) system good-passenger_and_slot-machine cannot stay forever in any state of the processes good-passenger and slot-machine in which these processes do not wait for input signals; each behaviour of the timed system is a fair behaviour of the system with the fairness conditions. And since the progress property 1 deals with the modality EACH on all (fair) behaviours, the progress property 1 for the system with the fairness conditions implies the same property for the original system.

As for the approach of [9], it consists of classification of properties on the kind of formulae they are specified by, development and justification of proof principles for formulae of special kinds, and application of these proof principles. It should be noted that the proof principles in general differ from inference rules, because the inference rules are purely syntactical and they are used in the framework of an axiomatic theory, while the proof principles are semantical and they work in the framework of a metatheory usually including set theory or arithmetics.

Let us fix an arbitrary executional specification SYS.

Now we can formulate the proof principles for the progress properties. We formulate them for the sets of configurations SETCNF, SETCNF', possibly, with subscripts. The semantics of \( \text{SETCNF'} \to \text{SETCNF''} \) is as follows: for any configuration \( \text{CNF}' \) from \( \text{SETCNF}' \) and for any fair behaviour of SYS, if this behaviour starts from \( \text{CNF}' \), it contains a configu-
ration \( CNF'' \in SETCNF'' \). So, if \( SETCNF' \) and \( SETCNF'' \) are validity sets of logical specifications with the schemes \( A \) and \( B \), respectively, then \( SETCNF' \rightarrow SETCNF'' \) is equivalent to \( A \rightarrow B \). When formulating the principles, the concept of a fair firing is used. By a firing we mean a triple \( CNF' < EVN > CNF'' \), where \( CNF' \) and \( CNF'' \) are configurations, and \( EVN \) is an event obtained according to the step rules for the executional specification \( SYS \). A fair firing is a firing which is the beginning of a fair behaviour of the system. Each of the principles is rather evident, so we present them without proofs. We have to remark that our principles are similar to the proof rules and the Inductive Principle for “Leads-To” from [5, Section 3.6.3], but they are more flexible than those.

1. Subset principle:
\( SETCNF' \subseteq SETCNF'' \models SETCNF' \rightarrow SETCNF'' \) or in the logical form \( (A \rightarrow B) \models A \rightarrow B \).

2. Union principle:
\( \{SETCNF'_{i}| \rightarrow SETCNF''_{i}| i \in I \} \models (U_{i \in I} SETCNF'_{i}) \rightarrow (U_{i \in I} SETCNF''_{i}) \)
or in the logical form \( (\forall i \in I. A_{i} \rightarrow B_{i}) \models (\exists i \in I. A_{i} \rightarrow B_{i}) \) for any finite set \( I \).

3. One step principle:
\( SETCNF = \{CNF''| \exists \text{ a fair firing } CNF' < EVN > CNF'' \} \models \{CNF'\} \rightarrow SETCNF \).

4. Transitivity principle:
\( SETCNF', SETCNF'' \rightarrow SETCNF''' \models SETCNF' \rightarrow SETCNF''' \)
or, in the logical form, \( A \rightarrow B, B \rightarrow C \models A \rightarrow C \).

5. Principle of partial mapping to well-founded set:
Let \( WFS \) be a well-founded set, i.e., a set with a partial order \( < \) and without infinite descending sequences. Let \( MIN \) be the set of minimal elements of \( WFS \). Let \( f \) be a partial function from the set of configurations \( SETCNF \) to the well-founded set \( WFS \). Let \( f^{-} \subseteq WFS \times SETCNF \) be the inverse of \( f \). (The inverse \( f^{-} \) is a set of pairs \( \{(C1,C2)|C1 \in WFS, C2 \in SETCNF \text{ such that } f(C2) = C1\}. \)
\( \forall v \in WFS \setminus MIN, f^{-}\{v\} \rightarrow f^{-}\{u|v > u\} \models f^{-}(WFS) \rightarrow f^{-}(MIN) \).

We would like to have a sufficient criterion for the principle of partial mapping to well-founded set: if the following conditions (INV) and (DEC) hold, then the function \( f \) meets the principle of partial mapping to well-founded set.

\( (INV) \forall v \in WFS \setminus MIN, \forall \text{ fair firing } CNF' < EVN > CNF'', \text{ if } f(CNF') = v, \text{ then } f(CNF'') \leq v; \)

\( (DEC) \forall \text{ fair behaviour, if } f \text{ in the initial configuration does not take a minimal value, then in this behaviour there is a pair of configurations where } f \text{ takes different values.} \)
5.2. Example “Passenger and Slot–Machine”

Our aim is to apply the principles described above to verification of the progress property 1 of the timeless system with fairness conditions from the section 2. Let us remind that for simplicity in the extended names in this section we use the abbreviation “gp” for “good–passenger” and “sm” for “Slot–Machine”, respectively.

According to the transitivity principle for proving progress property, $(\forall \text{gp.start} \& (\forall \text{sm.start} \& (\text{buttons IS EMPTY} \& (\text{indicator IS EMPTY}) \& (\text{sm.expenses[gp.station]} > 0) \Rightarrow \{\text{ticket WITH gp.station IN booking}\}$ it is sufficient to prove the correctness of each of the “local” progress properties of $P1 \Rightarrow P2 \Rightarrow P3 \Rightarrow P4 \Rightarrow P5$, where:

$P1$ is $(\forall \text{gp.start} \& (\forall \text{sm.start} \& (\text{buttons IS EMPTY} \& (\text{sm.expenses[gp.station]} > 0);$

$P2$ is $(\forall \text{gp.s’} \mid s’ \in S’ = \{\text{look, continue, choin, drop}\}) \& \forall (\forall \text{sm.s”} \mid s” \in S” = \{\text{show1, show2, getcoins, add}\}) \& (\text{sm.station = gp.station}) \& (\text{sm.left} \leq \text{gp.left}) \& (\text{buttons IS EMPTY}) \& (\forall \text{sm.show2}) \& \forall (\text{sm.left} \leq \text{indicator.left} \leq \text{gp.left});$

$P3$ is $(\forall \text{gp.continue}) \& \forall (\forall \text{sm.s”} \mid s” \in S” = \{\text{show1, show2, getcoins, add}\}) \& (\text{sm.station = gp.station}) \& (\text{gp.left} \leq 0) \& (\text{sm.left} \leq 0) \& (\text{buttons IS EMPTY});$

$P4$ is $(\forall \text{gp.get2} \& (\forall \text{sm.init1}) \& (\text{sm.station = gp.station}) \& (\text{buttons IS EMPTY}) \& (\text{ticket WITH sm.station IN booking});$

$P5$ is $(\text{ticket WITH gp.station IN booking}).$

But the property $P4 \Rightarrow P5$ is evident because it is a particular case of the subset principle.

Properties $P1 \Rightarrow P2$ and $P3 \Rightarrow P4$ can be proved by a model-checker because the set of all possible values of variables and parameters is restricted by the set of their values in an initial configuration, i.e., a configuration where $P1$ ($P3$, respectively) holds, so that only finite information is changed. But the step $P2 \Rightarrow P3$ is inherently parameterized by the price of the ticket required; therefore, this step is inductive.

Now let us consider the proof of the progress property $P2 \Rightarrow P3$.

Let $A$ be the precondition of this progress property, and $B$, its postcondition. As a well-founded set let us take the set of pairs of natural numbers with the following partial order: $(a_1, b_1) < (a_2, b_2)$ iff either (a) $a_1 \leq a_2$ and $b_1 < b_2$, or (b) $a_1 < a_2$ and $b_1 \leq b_2$. Then $MIN = \{(0, 0)\}$. Let $\text{SETCNF}$ be the set of configurations such that $CNF \models A$.

As the partial function $f : \text{SETCNF} \rightarrow WFS$ let us take the function defined as follows: $f(CNF) = (\text{POS(sm.left)}, \text{POS(gp.left)})$, if $CNF \models A$, and undefined otherwise. Here $\text{POS}$ is the operation of taking the positive part of an integer number, i.e., $\text{POS}(c) = c$, if $c > 0$, and 0 otherwise. Let us prove that $f^{-1}(WFS) \rightarrow f^{-1}(MIN)$ applying the sufficient criterion
for the principle of partial mapping to well-founded set.

(INV) Let us choose \( v = (a, b) \in WFS \setminus MIN \) and a fair firing \( CNF' < EVN > CNF'' \), such that \( f(CNF) = v \). Since \( CNF' \models A \), \( CNF' \models (\forall (AT \ gp.s' | s' \in S0')) \) and \( CNF'' \models (\forall (AT \ sm.s'' | s'' \in S1')) \). Thus, the following events are possible in the configuration \( CNF' \): RD(info, gp.left, indicator), EXE(IF gp.left \leq 0 THEN SKIP ELSE ABORT), EXE(IF gp.left > 0 THEN SKIP ELSE ABORT), EXE("choosing the value for gp.nominal"), WRT(coin, gp.nominal, slot), CLN(indicator), WRT(info, sm.left, indicator), RD(coin, sm.nominal, slot), EXE("decrementing the value of sm.left").

By virtue of \( CNF' \models A \), in the configuration \( CNF' \) holds: \( sm.left \leq gp.left \) and (AT sm.show2) \( \lor (sm.left \leq indicator.left \leq gp.left) \).

Therefore, \( f(CNF'') \in (a, b), (a, d), (a - c, b) \) so, \( f(CNF'') \leq (a, b) \), where \( c \) is the value of sm.nominal in \( CNF'' \), and \( d \) is the value of indicator.left in \( CNF'' \).

(DEC) Let \( v \in WFS \setminus MIN \). According to the rules of the structural operational semantics and the property (INV), we have:

\[
\{CNF_0 | f(CNF_0) = v\} \rightarrow SETCNF' = \{CNF_1 | f(CNF_1) \leq v \text{ and } CNF_1 \models (AT \ sm.getcoin)\}.
\]

Let \( SETCNF' = \{CNF_2 | f(CNF_2) \leq v, CNF_2 \models (AT \ sm.getcoin) \} \) and \( CNF_3 \models (slot ISFULL) \), and \( SETCNF' = \{CNF_3 | f(CNF_3) \leq v, CNF_3 \models (AT \ sm.getcoin) \} \), and \( SETCNF'' = \{CNF_4 | f(CNF_4) \leq v \} \) and \( SETCNF'' = \{AT \ sm.add\} \). Then \( SETCNF' = SETCNF' \ U SETCNF'' \), in an obvious way, and \( SETCNF' \rightarrow SETCNF'' \), according to the rules of the structural operational semantics of the executional specifications and the property (INV).

For \( SETCNF'' \) we have:

\[
SETCNF'' \rightarrow \{CNF_5 | f(CNF_5) \leq v \} \text{, and in } CNF_5 \text{ holds:}
\]

\[
(slot IS EMPTY), (AT \ sm.getcoin), (AT \ gp.drop) \rightarrow SETCNF'.
\]

Therefore, \( SETCNF' \rightarrow SETCNF'' \).

Analogously to the proof of the progress property (1), we can show that (by virtue of the fairness conditions (\( ^* \) AT sm.add)) \( SETCNF'' \rightarrow \{CNF_5 | f(CNF_5) < v \} \) holds.

Therefore, for each fair behaviour \( CNF_0 \ldots CNF_i \ldots \), if \( f(CNF) = v \), then \( \exists i \geq 0 \), such that \( f(CNF) < v = f(CNF_0) \).

Therefore, \( f^-(WFS) \rightarrow f^-(MIN) \). Moreover, analogously to (1), one can show that \( \{CNF_0 | f(CNF_0) < v \} \rightarrow \{CNF | CNF \models B\} \). To complete the proof of progress property (2), it is sufficient to apply the transitivity principle:

\[
\{CNF | CNF \models A\} \rightarrow f^-(WFS), f^-(WFS) \rightarrow f^-(MIN), f^-(MIN) \rightarrow \{CNF | CNF \models B\} \models A \rightarrow B.
\]

Let us note that all "local" \( \rightarrow \) in the proof of progress property (2) can also be proved by the model-checking technique.
6. Conclusion

Thus Basic–REAL is presented as a language for executable specifications of distributed systems and logical specifications of their properties. The complete structural operational semantics for Basic–REAL is presented too. On the base of this semantics a proving technique for properties of a special kind ("progress properties") is designed. This technique combines proof principles and refinement, fairness conditions and time constrains. The Basic–REAL style of specifications and the verification technique are illustrated by specification and verification of a new example of a distributed system (good-passenger and Slot–Machine) protocol. Specification and verification of another example (a variant of the alternating bit protocol from [5]) in a framework of Basic–REAL and described proving technique are given in [2].

The semantics of time in Basic–REAL language is close to the fictitious clock semantics [1]. The semantics of [15] can be described in terms of Basic–REAL language using a scale including a unique time unit (tick). It is sufficient to extend a specification of the (distributed) system with a (specification of a) process consisting of two transitions such that (at least) one of them has a non-zero lower bound of its time interval, and fairness conditions ensure that the transition fires infinitely often.

Our research has considerable perspectives. We are going to generalize the proof technique for proving properties depending on time constraints and to develop compositional proof principles. An important problem is to develop a method for translation of SDL specifications annotated by logical formulae in equivalent Basic–REAL specifications. We intend to develop an abstract–real language with second-order quantifiers over sets of configurations and to develop proving technique for this abstract language combining proof principles and model-checking.

References


Appendix: Basic–REAL specification of the example

Good_Passenger: PROCESS

1 min = 60 sec;
1 ing <= 1 min <= 20 ing;

TYPE value IS GRAPH 1, 5, 10, 20, 50;
TYPE region IS GRAPH a, b, c, none;

PR VAR station, station2 OF region.
PR VAR left, val OF INT.

INP UNB QUE CHN Decision
  FOR s
    WITH PAR stat OF region.
    LIFE 1REQ.

OUT 1-ELM QUE CHN Buttons
  FOR s_button
    WITH PAR stat OF region.
    LIFE 1REQ.
  FOR cancel
    LIFE 1REQ.
  FOR ticket_req
    LIFE 1REQ.

INP UNB QUE CHN Booking
  FOR ticket
    WITH PAR stat OF region.
    LIFE 1REQ.

OUT UNB QUE CHN Slot
  FOR coin
    WITH PAR nom OF value.
    LIFE 1REQ.

INP UNB QUE CHN Change
  FOR coin
    WITH PAR nom OF value.
    LIFE 1REQ.

INP 1-ELM QUE CHN Table
  FOR info
    WITH PAR left OF INT.
    LIFE INF REQ.
start
   EXE READ s(station) FROM Decision
   FROM NOW UPTO 1 ing;
   JUMP press.
press
   WRITE s_button(station) INTO Buttons
   FROM NOW UPTO 1 ing;
   JUMP look.
look
   READ info(left) FROM Indicator
   FROM NOW UPTO 1 ing;
   JUMP continue.
continue
   IF left <= 0 THEN JUMP get1
   ELSE JUMP chcoin
get1
   WRITE ticket_req INTO Buttons
   FROM NOW UPTO 1 ing;
   JUMP get2.
get2
   READ ticket(station2) FROM Booking
   FROM NOW UPTO 1 ing;
   JUMP satisfaction.
chcoin
   EXE IF left >= 50 THEN nominal = 50
      ELSE IF left >= 20 THEN nominal = 20
      ELSE IF left >= 10 THEN nominal = 10
      ELSE IF left >= 5 THEN nominal = 5
      ELSE nominal = 1
      FI FI FI FI
      FROM NOW UPTO 1 ing;
      JUMP drop.
drop
   WRITE coin(nominal) INTO Slot
   FROM NOW UPTO 1 ing;
   JUMP look.