Joint NCC & IIS Bull., Comp. Science, 7 (1997), 35–55
© 1997 NCC Publisher

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 fictious-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 ($\Box$ and $\Diamond$), 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 ∞ 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 cancelling, 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 second, 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' Passenger 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 executional specifications of the Basic–REAL language. It will be presented 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 operations (`opr`), where `1 min = 60 sec; 1 ing` $\leq$ `1 min` $\leq$ `20 ing; 30 opr` $\leq$ `1 sec` $\leq$ `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)`;

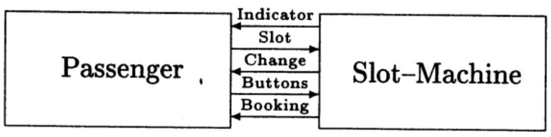

Figure 1. Block: good–passenger_and_Slot–Machine

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 `s_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 "untila one 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.

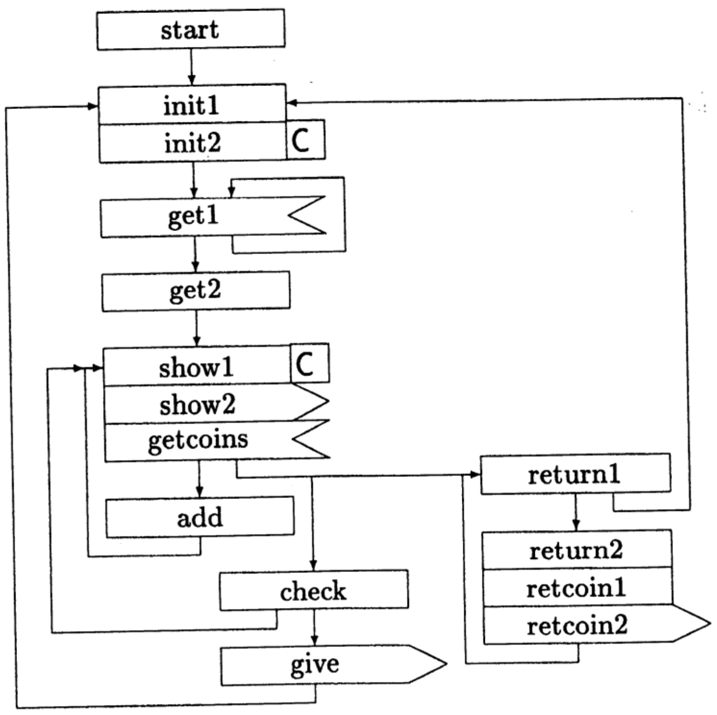

Figure 2. The Slot–Machine

Here, a name of a state in frame ▭ means that in a transition marked with this state an execution of non-deterministic program is done, ▭> means sending of a signal, ▭< means acceptance of a signal, and ▭▣ means cleaning a channel.

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

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 .

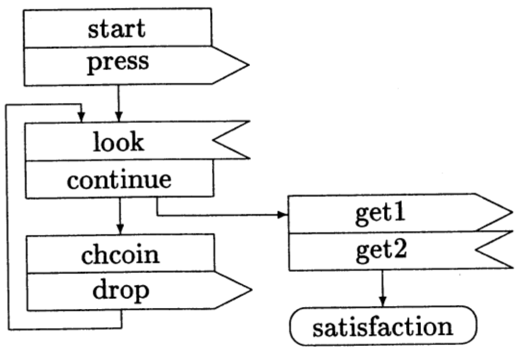

Figure 3. The Good Passenger

Timeless progress property 1: the 'good' Passenger is sure to get the ticket. It is expressed as follows:

(AT *gp.start*) & (AT *sm.start*) & (EMP *buttons*) $\mapsto$ (*ticket*(*gp.station*) $\in$ *booking*).

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

The $\mapsto$ symbol is an abbreviation for the construct $\Rightarrow$ EACH *good-passenger_and_slot-machine* $\Diamond$ FROM *NOW* UNTIL ∞ 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 ∞), such that the formula after the symbol $\mapsto$ is true.

When the time interval of a transition is FROM *NOW* UNTIL ∞, we omit it.

Timed progress property 1: the 'good' Passenger will get the ticket by the (pre-known) time moment $T = Const * (MaxTicketPrice \,/\, MinCoinValue)$ min where min means minutes. It can be obtained from timeless property 1 by replacing the symbol ∞ by T.

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

(AT *bp.start*) & (AT *sm.start*) & (EMP *booking*) $\Rightarrow$ EACH bad-passenger_and_slot-machine $\Box$ FROM *NOW* UNTIL ∞ (EMP *booking*), where EACH bad-passenger_and_slot-machine means "for each behaviour of the block bad-passenger_and_Slot--Machine", $\Box$ FROM *NOW* UNTIL ∞ means "for each time moment between zero and infinity".

3. Foundations of Basic–REAL semantics

In order to define semantics of specifications (both executional 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 executional 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 $M = (DOM, INT, DTS)$, where a non-empty set $DOM \neq \emptyset$ is the domain of the model (i.e., the union of all the data type domains defined in the specification); INT is the interpretation of relation and operation symbols from the specification by the relations and the operations over DOM; and 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 $\in DOM$. Two relations EMP and FUL, and two partial operations PUT and GET are defined for the data structure DAT, such that: $PUT : (DAT \times SIG \times DOM) \to DAT$, $dom(PUT) = (DAT \setminus FUL) \times SIG \times DOM$, $val(PUT) = DAT \setminus EMP$, and $GET : DAT \to (DAT \times SIG \times DOM)$, $dom(GET) = DAT \setminus EMP$, $val(GET) = ((DAT \setminus FUL) \times SIG \times DOM)$.

For a concrete data structure DAT, a graph $st \in DAT \setminus FUL$, a signal $sg \in SIG$, and an element $el \in DOM$, a graph $PUT(st, sg, el)$ 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 DAT, a graph $st \in DAT \setminus EMP$ $GET(st)$ is a triple (st', sg, el), where st' differs from 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 st. (The rules of removing are determined by the structure DAT itself.) For example, if the data structure is a queue, then DAT is a set of all finite sequences of pairs (signal, parameter). The relation EMP is true on the empty sequence, the relation FUL is always false.

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

The configuration space SPC_M (or SPC, while M is fixed) is a set of configurations 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 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 "/" 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 $T1$ and $T2$ we will write $T1 \leq T2$ iff the value t^1 of the global clock in $T1$ is less than or equal to the value t^2 of the global clock in $T2$.

For example, let the scale be `10 tact` $\leq$ `11 tick, 10 tick` $\leq$ `11 tact, 60 sec = 1 min`. The time units occurring in the scale are *tact*, *tick*, *sec* and *min*. Then m1 = $(100, 102, 10^3, 6*10^4)$ and m2 = (105, 101, 20, 1200) are possible variants of the speeds of the clocks, while m3 = $(100, 111, 10^3, 10^4)$ 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 $m1$, 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 $VAL_{CNF}(t_1)$, …, $VAL_{CNF}(t_2)$ are determined according to the ordinary rules.

If SPC is a locator, then its scheme has the form **AT** *state*. Then $CNF \models SPC \Leftrightarrow$state$\in ACT$.

If SPC is a controller, then its scheme has the form **EMP***chan* or **OVF***chan*. Then $CNF \models SPC' \Leftrightarrow$ EMP(C(*chan*)) or $CNF \models SPC \Leftrightarrow$ FUL(C(*chan*)).

If SPC is a checker, then its scheme has the form *sig* $\in$ *chan* or *sig* **RD** *chan*. Then in the first case: $CNF \models SPC \Leftrightarrow \exists$ *val* $\in$ DOM such that $\exists$ (*sig*, *val*) $\in$ C(*chan*); in the second case: $CNF \models SPC \Leftrightarrow \exists$ *graph* $\in$ DAT and *val* $\in$ DOM, such that GET(C(*chan*))=(*graph*, *sig*, *val*).

Induction step. If the scheme of SPC is a name of a predicate *pred*, then $CNF \models SPC \Leftrightarrow$ CNF $\models$ *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 $CNF \models SPC \Leftrightarrow$ CNF $\not\models$ SPF, where SPF differs from SPC by the scheme only, which is F.

If the scheme of SPC is $\forall x$.F ($\exists x$.F), where F is the scheme of a formula, then $CNF \models SPC \Leftrightarrow$ for each (resp., some) configuration CNF' differing from CNF at most by the evaluation of variable x, the following holds: $CNF' \models SPF$, 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 $\Box$ or $\Diamond$, 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", $\Box$ means "for all time moments", $\Diamond$ means "there exists a time moment".

For example: $CNF \models$ EACH SYS $\Diamond$ *duration* A holds iff for each behaviour of the executional specification SYS starting from the configuration CNF, there exists a time moment $T' \in$ *duration*, such that $CNF' \models SPF$ 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 $\models CNF1 < event > CNF2$, or

$$\frac{\text{CND}}{CNF1 < event > CNF2}$$

where CND is a condition on the configurations $CNF1$ and $CNF2$ 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 $CNF1$ into the configuration $CNF2$. A countable sequence of configurations is a behaviour of an executable specification iff for each successive pair $CNF1$ and $CNF2$ from the sequence there exist an event and a condition CND, so that CND $\models CNF1 < event > CNF2$ 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)

$$\frac{\text{for all } i = 1, \ldots, k\ CNF1 < \text{event}/P_i > CNF2}{CNF1 < event > CNF2}$$

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 $CNF1 = (T1, V1, C1,\ S1 = (ACT1, DEL1))$, $CNF2 = (T2, V2, C2, S2 = (ACT2, DEL2))$.

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.

TIME.CONST $T1 = T2$

TIME.STEP $T1 \leq T2$

VAR.CONST $\forall x.V1(x) = V2(x)$.

VAR.STEP(x) $\forall y \neq x.V1(y) = V2(y)$.

CHAN.CONST $\forall chan.C1(chan) = C2(chan)$.

CHAN.STEP($chan$) $\forall$ $chan' \neq chan$: $C1(chan') = C2(chan')$.

CHAN.HEAD($chan, sig, x$) $GET(C1(chan)) = (C2(chan), sig, V2(x))$.

DEL.ZER $\forall state.DEL2(state) = 0$.

DEL.NOT-OUT $\forall state$. if $state \in ACT1$, then there is a transition in the process diagram *state body interval jump*, so that $DEL1(state)$ does not exceed the right bound of the *interval*.

DEL.IN($state, interval$) $DEL1(state) \in interval$.

DEL.PROGR $\forall state$. if $state \in ACT1$, then $DEL2(state) = DEL1(state) + T2 - T1$, else $DEL2(state) = 0$.

ACT.CONST $\forall state.state \in ACT1 \Leftrightarrow state \in ACT2$.

ACT.UNIQUE $\exists$ unique $state.state \in ACT1$.

ACT.ACT($state$) $state \in ACT1$, $\forall state' \neq state.state' \notin ACT1$.

ACT.NEXT($next$) $next \in ACT2$, $\forall state' \neq next.state' \notin ACT2$.

RTR(*state, sig, x, chan, interval, next*) the process diagram contains the transition *state* `READ`*sig(x)* `FROM`*chan* *interval* `JUMP`*Set*, where *Set* is the set of states such that $next \in Set$.

Now we can formulate the step rules.

RULE 1 (Stuttering)

$$\frac{\text{TIME.STEP, VAR.CONST, CHAN.CONST, ACT.UNIQUE, ACT.CONST, DEL.PROGR, DEL.NOT-OUT}}{CNF1 < \text{INVIS} > CNF2}$$

RULE 4 (Reading)

$$\frac{\text{TIME.CONST, DEL.ZER, } \exists\ \mathit{state, interval, next}\text{: VAR.STEP}(x)\text{, CHAN.HEAD}(\mathit{chan}, \mathit{sig}, x)\text{, CHAN.STEP}(\mathit{chan})\text{, ACT.ACT}(\mathit{state})\text{, ACT.NEXT}(\mathit{next})\text{, DEL.IN}(\mathit{state}, \mathit{interval})\text{, RTR}(\mathit{state, sig, x, chan, interval, next})}{CNF1 < \text{RDN}(\mathit{chan}, \mathit{sig}, x) > CNF2}$$

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 $SETCNF' \mapsto SETCNF''$ is as follows: for any configuration CNF' from $SETCNF'$ and for any fair behaviour of SYS, if this behaviour starts from CNF', it contains a configuration $CNF'' \in SETCNF''$. So, if $SETCNF'$ and $SETCNF''$ are validity sets of logical specifications with the schemes A and B, respectively, then $SETCNF' \mapsto SETCNF''$ is equivalent to $A \mapsto B$. 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:
$SETCNF' \subseteq SETCNF'' \vdash SETCNF' \mapsto SETCNF''$ or in the logical form $(A \to B \vdash A \mapsto B)$.

2. Union principle:
$\{SETCNF'_i \mapsto SETCNF''_i | i \in I\} \vdash (U_{i \in I} SETCNF'_i) \mapsto (U_{i \in I} SETCNF''_i)$ or in the logical form $(\forall i \in I.A_i \mapsto B_i \vdash \exists i \in I.A_i \mapsto B_i)$ for any finite set I.

3. One step principle:
$SETCNF = \{CNF'' | \exists$ a fair firing $CNF' <$ EVN $> CNF''\} \vdash \{CNF'\}$ $\mapsto SETCNF$.

4. Transitivity principle:
$SETCNF' \mapsto SETCNF''$, $SETCNF'' \mapsto SETCNF''' \vdash SETCNF' \mapsto$ $SETCNF'''$ or, in the logical form, $A \mapsto B,\ B \mapsto C \ \vdash \ A \mapsto C$.

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 $\{(C1, C2) | C1 \in WFS, C2 \in SETCNF$ such that $f(C2) = C1\}$.)
$\forall v \in WFS \setminus MIN. f^-\{v\} \mapsto f^-\{u | v > u\} \vdash f^-(WFS) \mapsto f^-(MIN)$.

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)	$\forall\ v \in WFS \setminus MIN$, $\forall$ fair firing $CNF' < EVN > CNF''$, if $f(CNF') = v$, then $f(CNF'') \leq v$;
(DEC)	$\forall$ fair behaviour, if f in the initial configuration does not take a minimal value, then in this behaviour there is a pair of configurations where f 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, (**AT** *gp.start*) & (**AT** *sm.start*) & (*buttons IS EMPTY*) & (*indicator IS EMPTY*) & (*sm.expenses*[*gp.station*] > 0) $\longmapsto$ (*ticket WITH gp.station IN booking*) it is sufficient to prove the correctness of each of the "local" progress properties of $P1 \mapsto P2 \mapsto P3 \mapsto P4 \mapsto P5$, where:

$P1$ is (**AT** *gp.start*) & (**AT** *sm.start*) & (*buttons IS EMPTY*) & (*indicator IS EMPTY*) & (*sm.exspenses* [*gp.station*] > 0);

$P2$ is $\vee$(**AT** *gp.s'* $\mid s' \in S0' = \{$*look, continue, chcoin, drop*$\}$) & $\vee$(**AT** *sm.s''* $\mid s'' \in S1'' = \{$*show*1, *show*2, *getcoins, add*$\}$) & (*sm.station* = *gp.station*) & (*sm.left* $\leq$ *gp.left*) & (*buttons IS EMPTY*) & ((**AT** *sm.show*2) $\vee$(*sm.left* $\leq$ *indicator.left* $\leq$ *gp.left*));

$P3$ is (**AT** *gp.continue*) & $\vee$(**AT** *sm.s''* $\mid s'' \in S1'' = \{$*show*1, *show*2, *getcoins, add*$\}$) & (*sm.station* = *gp.station*) & (*gp.left* ≤ 0) & (*sm.left* ≤ 0) & (*buttons IS EMPTY*);

$P4$ is (**AT** *gp.get*2) & (**AT** *sm.init*1) & (*sm.station* = *gp.station*) & (*buttons IS EMPTY*) & (*ticket WITH sm.station IN booking*);

$P5$ is (*ticket WITH gp.station IN booking*).

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

Properties $P1 \mapsto P2$ and $P3 \mapsto P4$ 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 $P2 \mapsto P3$ 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 $P2 \mapsto P3$:

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: (a1, b1) < (a2, b2) iff either (a) a1 $\leq$ a2 and b1 < b2, or (b) a1 < a2 and b1 $\leq$ b2. Then $MIN = \{(0, 0)\}$. Let $SETCNF$ be the set of configurations such that $CNF \models A$.

As the partial function $f : SETCNF \to WFS$ let us take the function defined as follows: $f(CNF) = (POS(sm.left), POS(gp.left))$, if $CNF \models A$, and undefined otherwise. Here POS is the operation of taking the positive part of an integer number, i.e., $POS(c) = c$, if $c > 0$, and 0 otherwise. Let us prove that $f^{-}(WFS) \mapsto f^{-}(MIN)$ 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 $CNF' \models A$, $CNF' \models (\vee(AT\ \ gp.s'|s' \in S0'))$ and $CNF' \models (\vee(AT\ sm.s''|s'' \in S1''))$. 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) $\vee$ (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:

$\{CNF_0 | f(CNF_0) = v\} \mapsto SETCNF' = \{CNF_1 |\ f(CNF_1) \leq v$ and $CNF_1 \models (AT\ sm.getcoin)\}$.

Let $SETCNF1' = \{CNF_2 |\ f(CNF_2) \leq v,\ CNF_2 \models (AT\ sm.getcoin)$ and $CNF_2 \models (slotISFULL)\}$, and $SETCNF2' = \{CNF_3 | f(CNF_3) \leq v$, $CNF_3 \models (AT\ sm.getcoin)$ and $CNF_3 \models (slot\ IS\ EMPTY)\}$, and $SETCNF'' = \{CNF_4 \mid f(CNF_4) \leq v$ and $CNF_4 \models (AT\ sm.add)\}$. Then $SETCNF' = SETCNF1'\ U\ SETCNF2'$, in an obvious way, and $SETCNF1' \mapsto SETCNF''$, according to the rules of the structural operational semantics of the executional specifications and the property (INV). For $SETCNF2'$ we have:

$SETCNF2' \mapsto \{CNF_5 | f(CNF_5) \leq v$, and in CNF_5 holds:
$(slot\ IS\ EMPTY)$, $(AT\ sm.getcoin)$, $(AT\ gp.drop)\} \mapsto SETCNF1'$.

Therefore, $SETCNF' \mapsto SETCNF''$.

Analogously to the proof of the progress property (1), we can show that (by virtue of the fairness conditions (˜ AT *sm.add*)) $SETCNF'' \mapsto \{CNF_6 | f(CNF_6) < v\}$ holds.

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

Therefore, $f^-(WFS) \mapsto f^-(MIN)$. Moreover, analogously to (1), one can show that $\{CNF_6 | f(CNF_6) < v\} \mapsto \{CNF | CNF \models B\}$. To complete the proof of progress property (2), it is sufficient to apply the transitivity principle:

$\{CNF | CNF \models A\} \ \mapsto f^-(WFS)$, $f^-(WFS) \mapsto f^-(MIN)$, $f^-(MIN) \mapsto \{CNF |\ CNF \models B\} \vdash A \mapsto B$.

Let us note that all "local" $\mapsto$ 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 fictious 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

[1] R. Alur, T.A. Henzinger, *Logics and Models of Real Time: A Survey*, Lecture Notes in Computer Science, **600**, 1992, 74–106.

[2] E.V Bodin, *Approaches to the verification of specifications on language REAL*, Specification and verification problems for concurrent systems, Novosibirsk, 1995 (in Russian).

[3] M. Broy, *Towards a formal foundation of the specification and description language SDL*, Formal Aspects of Computing, **3**, No. 1, 1991, 21–57.

[4] A.R. Cavalli, F. Horn, *Proof of specification properties by using finite state machines and temporal logic*, Proc. of 7-th IFIP Conf. on Protocol Specifications, Testing, and Verification, 1987, 221–233.

[5] K.M. Chandy, J. Misra, *Parallel Program Design*, Reading a.o., Addison-Wesley, 1988.

[6] E.M. Clarke, E.A. Emerson, A.P. Sistla, *Automatic verification of finite state concurrent systems using temporal logic specifications*, ACM Trans. Programming Languages Systems, **8**, No. 2, 1986, 244–263.

[7] A. Gammelgaard, J.E. Kristensen, *A correctness proof of a translation from SDL to CRL*, Proc. of 6-th SDL Forum, North-Holland, 1993, 205–219.

[8] D. Harel, *First-order dynamic logic*, Lecture Notes in Computer Science, **68**, 1979.

[9] A. Henzinger, Z. Manna, A. Pnueli, *Temporal proof methodologies for real-time systems*, Proc. of Symp. on POPL, 1991, 353–366.

[10] S. Leue, *Specifying real-time requirements for SDL specifications – A temporal logic-based approach*, Proc. of 15-th IFIP Intern. Symp. on Protocol Spec., Test., and Verif., Warsaw, 1995, 19–34.

[11] D. Mery, A. Mokkedem, *CROCOS: An integrated environment for interactive verification of SDL specifications*, Lecture Notes in Computer Science, **663**, 1993, 343–356.

[12] V.A Nepomniaschy, N.V. Shilov, *REAL92: A combined specification language for real-time concurrent systems and properties*, Lecture Notes in Computer Science, **735**, 1993, 377–389.

[13] V.A. Nepomniaschy, N.V. Shilov, E.V. Bodin, *A concurrent systems specification language based on SDL & CTL*, Proc. of Workshop on Concurrency, Specifications & Programming, Berlin, Humboldt University, Informatik–Bericht, No. 36, 1994, 15–26.

[14] F. Orava, *Formal semantics of SDL specifications*, Proc. of 8-th IFIP Intern. Symp. on Protocol Spec. Test., and Verif., 1988, 143–157.

[15] J.S. Ostroff, *Automated verification of timed transition models*, Lecture Notes in Computer Science, **407**, 1990, 247–256.

[16] G.D. Plotkin, *A structure approach to operational semantics*, Technical report FN–19, Aarhus University, DAIMI, Denmark, 1981.

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.
```