Basic-REAL: integrated approach for design, specification and verification of distributed systems

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

Institute of Informatics Systems, Russian Academy of Sciences, Siberian Division,
6, Lavrentiev ave., 630090, Novosibirsk, Russia,
{vnep,shilov,bodin,kozura}@iis.nsk.su

Abstract. We suggest a three-level integrated approach to design, specification and verification of distributed system. The approach is based on a newly designed specification language Basic-REAL (bREAL) and comprises (I) translation of a high-level design of distributed systems to executional specifications of bREAL, (II) presentation of high-level properties of distributed systems as logical specifications of bREAL, (III) problem-oriented compositional deductive reasoning coupled with model-checking. The paper presents syntax and semantics of bREAL in formal and informal levels, some meta-properties of this language (namely, stuttering invariance and interleaving concurrency), proof-principles and model-checking for progress properties. An illustrative example (Passenger and Vending Machine) is also presented.

1 Introduction

The standard formal description techniques (FDT) such as SDL are widely used in practice for design of distributed systems. Verification of FDT specifications, i.e., proving their safety, progress and other properties, is an actual research problem. Verification approaches are based on formal semantics of FDT specifications, suitable language for properties presentation and on feasible and sound verification techniques.

Concise formal semantics has been suggested for some particular fragments of SDL [5, 11, 12, 19, 22], though a complete formal semantics of SDL-96 has more than 500 pages. Due to lack of feasible formal semantics of SDL, a popular approach to verification of properties of SDL specifications comprises a property-conservative translation/simulation of the specifications to another intermediate formalism, and formal verification of the translated specifications against the desired properties [3, 4], but soundness of this simulation is usually justified by informal or quasi-formal arguments.

Temporal logic is a popular formalism for presentation of the properties of SDL specifications. Different temporal logics have been suggested for representation of safety and progress properties: linear temporal logic LTL [3], branching temporal logic CTL [7, 19], metric temporal logic MTL [16], temporal logic of actions TLA+ [13], etc. We would like to remark that these formalisms are external for SDL lead to complicated formulae for presentation of real-time properties.

Two techniques are mainly used for proving properties of the formal models of distributed systems, namely, model-checking and deductive reasoning. The standard model-checking has been successfully used for finite systems, while deductive reasoning is oriented to parameterized and infinite state systems. The automatic nature is the main advantage of the model-checking, while deductive reasoning implies a manual design of proof-outlines and the level of automatization is limited by proof-checking.

As follows from the above arguments, the intermediate formalism is a very critical issue for sound verification of SDL specifications. We suppose that a proper intermediate formalism should meet certain requirements, some of which are enumerated below:
1. SDL-like syntax and semantics,
2. modest blow up in size against SDL specifications,
3. integration with presentation of properties,
4. verification-oriented and human-friendly formal semantics,
5. opportunity for proving meta-level properties and reasoning,
6. support for a variety of verification techniques and their integration.

Let us discuss from viewpoint of these requirements some intermediate formalisms which have been used for SDL. A formal language φ^-SDL [1] is a limited version of SDL. Its semantics is based on process algebras [1]. This language does not cover all structural aspects of SDL, and it is not integrated with presentation of properties. Another intermediate language IF [4] is used for modeling static SDL specifications. It has a considerable expressive power and formal operational semantics, but its meta-level reasoning remains quasi-formal. Although the input language Promela of model-checker SPIN is a convenient formalism for presentation of static SDL specifications [3], and it is well-integrated with linear temporal logic, its complex semantics is not oriented to meta-level reasoning and it has been designed for model-checking temporal properties of finite-state systems only. Thus we can summarize that none of the discussed intermediate formalisms simultaneously meets the constraints listed above.

The aim of the paper is to present our approach to verification of distributed systems specified in SDL-like style. The approach is based on a new intermediate high-level specification language Basic-REAL (bREAL) and meets the requirements listed above. The rest of the paper consists of six sections. The main constructs of bREAL are informally explained in Section 2, while formal context-free syntax is presented in Appendix 1. Section 2 also presents an illustrative example "Passenger and Vending Machine", while the corresponding SDL specification and executional bREAL specification are presented in Appendices 2 and 3. The structural operational semantics of bREAL is sketched in Sections 3, 4 and 5. Two theorems about important meta-properties of bREAL semantics are given in Section 6. Verification techniques based on finite-state model-checking and deductive-like reasoning are discussed in Section 7. Logical specification and verification of the example “Passenger and Vending Machine” are presented in Section 8. In Conclusion the results and prospects of our approach are discussed.

2 Outline of Basic-REAL

Informal[1] introduction of syntax and semantics. bREAL has executional and logical specifications. Logical specifications are used for properties presentation. They have hierarchical structure based on predicates. The predicates can be grouped into formulae. Formulae, in turn, make up higher level formulae. Distributed systems are presented as executional specifications. They have a hierarchical structure based on processes. The processes can be grouped into blocks. Blocks, in turn, make up higher level blocks. Channels are used for communication between such entities as processes, blocks, and the external environment. From the viewpoint of a process, each of its channels is external (input or output one). From the viewpoint of a block, each of its channels is inner if the channel connects its subblocks. A channel is external (input or output one) for a block if it connects a subblock with outside environment (for example, other blocks).

In general, a specification (executional or logical) consists of a head, a scale, a context, a scheme, and subspecification(s). The head defines the specification name and kind: an executional specification is either a process or a block, and a logical one is either a predicate or a formula. Processes and predicates are elementary specifications, blocks and formulae are composite specifications.

Every activity in bREAL has an associated time interval. Time intervals are expressed in terms of time units. Every time unit is a tick of a special clock. A collection of special clocks is a multiple clock. A scale is a finite set of linear integer (in)equalities where time units are variables. A priory, ticks of all clocks are asynchronous and have uninterpreted values, but scales introduce some synchronization. For example, a scale $\{\ 60\ sec\ =\ 1\ min;\ 60\ min\ =\ 1\ hour;\ 24\ hour\ =\ 1\ day;\}$ does not define any particular value for time units *sec*, *min*, *hour* and *day*, but impose the standard relation between them.

A context of a specification consists of type definitions, variable and channel declarations. Let us note that every variable can be declared as a program or a quantifier variable but not both. The values of the quantifier variables can not be changed in executional specifications, they can be varied by the (universal and existential) quantifiers in logical specifications. The values of program variables can be changed by assignments and parameter passing in executional specifications. Channels are intended for passing signals with possible parameters. They can have different inner structures, viz., queues, stacks, bags.

A scheme of a logical specification consists of a diagram and a system list. The system list consists of the name(s) of the executional (sub)specification(s) whose properties are described by the logical specification. There are four kinds of predicate diagrams: relations between values, locators of the control flow, emptiness/overflow controllers for channels, checkers for signals in channels. A

[1] Formal definition of bREAL syntax can be found in [21]. The context-free syntax is given in Appendix 1. Please refer for syntax details if necessary.

diagram of a formula is constructed from name(s) of (sub)specification(s) by propositional combinations, quantification over quantifier variables, and special dynamic/temporal expressions. Dynamic/temporal expressions are bREAL formalism for reasoning about every/some fair behaviours of distributed systems and every/some time moment in temporal intervals. This formalism expands the branching temporal logic CTL [9] by means of explicit time intervals in terms of time units in temporal modalities and explicit references to executional specifications in action modalities in style of dynamic logic [14]. There are several reasons why we prefer the explicit style of dynamic logic instead of the implicit style of CTL. The main ones are opportunities for explicit transformations, optimizations, (bi)simulation and compositional reasoning for executional specifications to be verified.

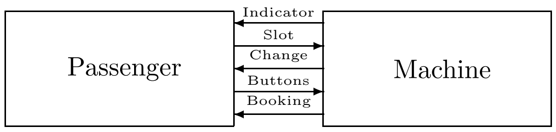
A scheme of an executional specification consists of a diagram and a finite set of fairness conditions. Fairness conditions restrict a set of behaviours: bREAL considers only the "fair behaviours", i.e., the behaviours where every fairness condition holds infinitely often.

A block diagram consists of channel routes. A route connects a subblock with another subblock or (exclusively) with the (external) environment. In the last case the channel is called an *input* or *output* channel. Otherwise, the channel is called an *inner* channel. All sub-blocks of a block work in parallel, and they interact with each other and with the external environment by sending/reading signals with parameters via channels.

A process diagram consists of transitions. Every transition consists of a control state, a body, a time interval, and a (non-deterministic, maybe) jump to next control states. The body of a transition determines the following actions: to read a signal from an input channel (and to assign the values of the parameters to program variables); to write a signal into an output channel (with parameter passing by value); to clean an input or output channel; to execute a (non-deterministic, maybe) sequential program. Each of the actions is atomic, but it can happen only within the time interval specified for the transition. Each process is sequential. A process communicates with the other processes and the environment by means of input/output channels.

An illustrative example: "Passenger and Vending machine". Let us consider the following example: a protocol Σ of serving a passenger by a vending-machine. The vending-machine keeps the money received from the passenger and has the following features: a keyboard with stations, return, and request buttons; a slot for coins; an indicator for showing a sum; a tray for change; a booking window. The passenger knows a desired destination, has enough money and can: press buttons on the keyboard; drop coins into the slot; see readings of the indicator; get coins from the change tray; get a ticket from the booking window. We consider a good passenger who has enough money to pay for the ticket and requests the ticket only when enough money has been paid. Vending machine and passenger can be time-independent (time-free case) or time-dependent (timed case). In the former case no time constraints are imposed on any activity or event, in the later case stricture constraints are imposed on every activity and event. Thus in timed case a slow passenger may not get the ticket after all.

An SDL specification of the time-free protocol is presented in Appendix 2. The top-level view is presented below:



This top-level view is also a block diagram for the corresponding bREAL specification. Appendix 3 presents other fragments of the corresponding bREAL specification. A complete bREAL specification can be assembled from these fragments and the block diagram by adding context and fairness conditions to the machine diagram and a context to the block diagram.

A protocol Σ is specified as a block which consists of two processes, namely, the process `passenger` and the process `machine`. These processes are connected by five channels. Two of them (`buttons` and `slot`) are directed from the passenger to the machine. The other three channels (`indicator`, `change`, and `booking`) are from the machine to the passenger. A behaviour of the process `machine` is fair if the process cannot stay forever in a state other than waiting for input signals. A behaviour of the process `passenger` is fair if the process cannot stay forever in a state other than waiting for input signals. It should be noted also that there are two different cases for Σ – Σ_{fin} and Σ_{par}: in the former case of Σ_{fin}, ticket price is fixed; in the later case of Σ_{par}, ticket price is a parameter. The specification of the time-free protocol has explicit time intervals `FROM NOW TILL FOREVER`. This case is presented in [21]. The specification of timed protocol is a variation of time-free specification where time intervals are parameterized as `FROM` $const'$ `TILL` $const''$.

3 Foundations of semantics of Basic-REAL

Data domains and channel structures. A model with channel structures is a triple $M = (DOM, INT, CHS)$, where a non-empty set is the domain DOM of the model, INT is an interpretation of relation and operation symbols over DOM, and CHS is a collection of available data structures for channels. Moreover, DOM includes integers and INT provides the standard interpretation for integer arithmetic operations and relations. DOM includes arrays and INT provides the standard interpretation for array operations *UpDate* and *Apply*. A data structure of a channel is a mapping which binds every channel with a set of possible contents. Every content is a finite oriented graph whose vertices are marked by signals with vectors of parameter values. For every particular channel structure $DAT \in CHS$, two monadic relations (EMP and FUL), two partial non-deterministic operations (PUT and GET) and one constant (INI) are defined. Let DOM^{PAR} be a set of parameter value vectors from DOM, *dom* and *rng* be the domain and range of the operations, and SIG be a finite set of all signals admissible for the channel. Then

$EMP = \{INI\}$, $PUT : (DAT \times SIG \times DOM^{PAR}) \to DAT$, $dom(PUT) = (DAT \setminus FUL) \times SIG \times DOM^{PAR}$, $rng(PUT) = DAT \setminus EMP$, $GET : DAT \to (DAT \times SIG \times DOM^{PAR})$, $dom(GET) = DAT \setminus EMP$, $rng(GET) = ((DAT \setminus FUL) \times SIG \times DOM^{PAR})$.
If $graph \in DAT \setminus FUL$, $signal \in SIG$ and $vector \in DOM^{PAR}$, then the graph $PUT(graph, signal, vector)$ is constructed by adding a new vertex with several adjacent edges and by marking the new vertex by the pair $(signal, vector)$. If $graph \in DAT \setminus EMP$, then the triple $(graph', signal, vector)$ is in $GET(graph)$ iff $graph'$ results from removing some vertex with all adjacent edges. Standard (un)bounded queues, stacks, and multisets (bags) are compatible with this notion of CHS. For example, let us consider a channel with stack access discipline to data for three admissible signals $\{a, b, c\}$ with a single integer parameter. A particular structure $DAT \in CHS$ for this channel is the set of finite chains (including the empty chain Λ) $sig_1(int_1) \longleftarrow ...sig_n(int_n)$, where $n \geq 0$, and $sig_i \in \{a, b, c\}$, $int_i \in Int$ for all $i \in [1..n]$. Relation EMP holds on the empty chain only, while another relation FUL does not hold at all. PUT and GET operations are illustrated below:

$$\begin{tikzcd} \Big(sig_1(int_1) \longleftarrow \; ... \; sig_n(int_n) \;,\; sig(int) \Big) \arrow[d, shift left=8ex, "PUT"] \\ sig_1(int_1) \longleftarrow \; ... \; sig_n(int_n) \longleftarrow \; sig(int) \arrow[u, shift right=8ex, "GET"] \end{tikzcd}$$

Time and Clocks. Informally speaking, every time unit is a tact of a special clock. All clocks are used for measuring time after some fixed moment in the past. A scale is a set of linear (in)equalities with time units as variables. Scales are used for synchronization of speed of clocks. Formally speaking, a time measure is a positive integer solution of the scale as a system of (in)equalities. We will identify a time measure MSR with a mapping which associates each time *unit* with its value $MSR(unit)$. With a fixed time measure MSR, an indication of a multiple clock T is a mapping of time units into integer numbers so that there exists an integer t such that for every time *unit*, we have: $T(unit) = [\frac{t}{MSR(unit)}]$. Just for example let us consider a scale $\{ 60\, sec = 1\, min;\ 60\, min = 1\, hour;\ 24\, hour = 1\, day; \}$. A possible time measure MSR is $\{ sec \mapsto 2,\ min \mapsto 120,\ hour \mapsto 7200,\ day \mapsto 172800 \}$. Then $T = \{ sec \mapsto 325,\ min \mapsto 5,\ hour \mapsto 0,\ day \mapsto 0 \}$ is a possible indication of the multiple clock $(sec, min, hour, day)$, since there exists an integer t (ex., 651) such that $T(sec) = 325 = [\frac{t}{2}] = [\frac{t}{MSR(sec)}]$, $T(min) = 5 = [\frac{t}{120}] = [\frac{t}{MSR(min)}]$, $T(hour) = 0 = [\frac{t}{7200}] = [\frac{t}{MSR(hour)}]$, $T(day) = 325 = [\frac{t}{172800}] = [\frac{t}{MSR(day)}]$. In contrast, another mapping $T = \{ sec \mapsto 0,\ min \mapsto 0,\ hour \mapsto 5,\ day \mapsto 325 \}$ can not be an indication for the time measure MSR.
Configurations. Let SYS be an executional specification the of bREAL language. Let us fix a model with structures for channels $M = (DOM, INT, CHS)$ and a time measure MSR. Let PR^1, ... PR^k be all processes of SYS. An extended name (of a variable, a state or a channel) is the name itself preceded by the "path" of sub-block names in accordance with the nesting. A configuration

CNF of SYS is a quadruple (T, V, C, S), where T is the indication of the multiple clock, V is the evaluation of variables, C is the contents of the channels, S is the control flow. The evaluation of variables is a mapping that maps every extended name of every variable of the processes PR^1, ..., PR^k to its value from DOM. The current flow is a pair (ACT, DEL) where ACT is a set of extended names of active states and DEL is a mapping that maps extended names of states of the processes PR^1, ..., PR^k to delays (the indication of a special local multiple clocks associated with states). For every individual process there is a single active state.

A *merge operation* $\mathcal{M}(CNF^1$, ..., $CNF^k)$ is said to be possible for the configurations $CNF^1 = (T^1, V^1, C^1, S^1)$, ..., $CNF^k = (T^k, V^k, C^k, S^k)$ of the processes PR^1, ..., PR^k iff $T^1 = \ldots = T^k$ and for every *channel* and for all processes PR^i and PR^j, which shared it as an input/output channel, $C^i(channel) = C^j(channel)$. If the merge is possible, then the result of the merge is a configuration $CNF = (T, V, C, S)$ of the executional specification SYS such that for every $0 \leq i \leq k$: $T = T^i$; the values V^i of the variables of the process PR^i coincides with the values V of their extended names; the contents C^i of the channels of the process PR^i coincides with the contents C of their extended names; the active state and the delays S^i for the states of the process PR^i coincides with the active states and delays for their extended name. When the sub-blocks are considered instead of the processes, the merge is defined in a similar way. If BLK is a sub-block of an executional specification SYS, and $CNF = (T, V, C, S)$ is the configuration of SYS, then the *projection of CNF to BLK* (denoted by CNF/BLK) is a configuration $CNF' = (T'$, V', C', $S')$ of the sub-block BLK such that $T' = T$; for every *variable* $V'(variable) = V(BLK.variable)$; for every *channel* $C'(channel) = C(BLK.channel)$; for every *state* $state \in S'.ACT = BLK.state \in S.ACT$ and $S'DEL(state) = S.DEL(BLK.state$. It is easy to prove by induction over the specification structure that a configuration of an executional specification is the merge of its projections to all its processes, i.e., $CNF = \mathcal{M}(CNF/PR^1, ..., CNF/PR^k)$.

4 The semantics of executional specifications

Step rules and semantics of blocks. The semantics of the executional specifications is defined in terms of events (EVN) and step rules. There are six kinds of events:

1. writing a signal with parameters into a channel (WRiTing),
2. reading a signal with parameters from a channel (ReaDiNg),
3. cleaning an input channel (CLeaNing INput),
4. cleaning an output channel (CLeaNing OUTput),
5. program execution (EXEcution),
6. invisible event (INVisible).

A *firing* is a triple $CNF1 < EVN > CNF2$. If $EVN \neq INV$, then the firing is said to be *active*. Otherwise, it is called *passive*. A step rule has the form CND $\models CNF1 < EVN > CNF2$, where $CNF1 < EVN > CNF2$ is a firing

while CND is a condition on the configurations $CNF1$, $CNF2$ and the event EVN. An intuitive semantics of the step rule is as follows: if the condition CND holds, then the executional specification can transform the configuration $CNF1$ into the configuration $CNF2$ by means of the event EVN. In total, there exist twelve step rules for executional specifications. A countable sequence of configurations is a behaviour of a specification iff, for every successive pair of configurations $CNF1$ and $CNF2$ of this sequence, there exists an event EVN and a condition CND, such that CND $\models CNF1 < EVN > CNF2$ is an instance of a corresponding step rule. Below are presented some steps rules[2]. A behaviour of an executional specification is said to be fair iff each of the fairness conditions of the specification holds infinitely often in the configurations of this behaviour.
Semantics of blocks For blocks there is a single step rule, namely, the composition rule. Informally, a behaviour of a block is an interleaving merge of consistent behaviours of its sub-blocks. Let us fix a model with channel structures M, a scale MSR, and a block B, consisting of the sub-blocks B_1, ..., B_k.

RULE 0 (Composition)

$$\begin{pmatrix} \forall i \in [1..k] \ : \\ CNF1/B_i < EVN/B_i > CNF2/B_i \end{pmatrix} \models CNF1 < EVN > CNF2.$$

Semantics of processes. The other eleven step rules deal with individual processes and the environment. For simplicity of presentation, let us fix a process and the two configurations, $CNF1 = (T1, V1, C1, S1)$ and $CNF2 = (T2, V2, C2, S2)$. Let us use meta-variables *state*, *state′*, *nextstate*, *signal*, *variable*, *variable′*, *channel*, *channel′*, *interval*, *program* and *jumpset* (for sets of states). Let us fix values of *state*, *nextstate*, *signal*, *variable*, *channel*, *program* and *jumpset* so that *nextstate* $\in$ *jumpset*.

Process stuttering and stabilization. The first rule for a process is a stutter rule. Informally, it concerns the case when nothing changes in the process. This rule is essential for the interleaving merge of consistent behaviours of some processes with shared channels into a behaviour of a block.

The second rule deals with stabilization and it means that a process is in a state which does not mark any transition on the process diagram. Thus, the process stabilizes forever, and the configuration of the process cannot change and is called a stable configuration.

Signal reading and writing. The third and fourth rules deal with a process reading a signal with a parameter from an input channel and writing a signal with a parameter into an output channel, respectively. For simplicity let us present and discuss the rule for writing a signal with a single implicit parameter.

RULE 4 (Writing a signal)

$$\begin{pmatrix} \text{The diagram has the transition} \\ state\ \texttt{WRITE}\ signal(variable)\texttt{INTO}\ channel\ interval\texttt{JUMP}\ jumpset, \\ state \in ACT1,\ nextstate \in ACT2,\ DEL1(state) \in interval, \\ T1 = T2, (\forall state'\ :\ DEL2(state') = 0),\ V1 = V2, \\ PUT(C1(channel), signal, V1(variable)) = C2(channel), \\ (\forall channel' \neq channel\ :\ C1(channel') = C2(channel')) \end{pmatrix}$$
$$\models CNF1 < WRT(channel, signal, variable) > CNF2.$$

[2] The complete semantics is given in [21].

This rule can be commented as follows. The process can write the *signal*, if it has a corresponding transition, and a control state of this transition is active ($state \in ACT1$) for a time which is the range of the time interval of the transition ($DEL1(state) \in interval$). Writing is an instant action ($T1 = T2$) which resets delays for all states of the process ($\forall state' \; : \; DEL2(state') = 0$). This action does not change values of variables and content of channels other than a channel to which it writes ($V1 = V2$ and $\forall channel' \neq channel \; : \; C1(channel') = C2(channel')$). In contrast, the content of this channel absorbs the signal with a value of a variable as the parameter value in accordance with structure of the channel ($C2(channel) = PUT(C1(channel), signal, V1(variable))$). The action passes control to some next state ($nextstate \in ACT2$).

Impact of external environment. The fifth and the sixth rules deal with appearance of a new signal with a parameter in an input channel and with disappearance of a signal with a parameter from an output channel. The environment `ENV` is responsible for those actions. The process itself can only observe the appearance of a new signal in an input channel or that some signal disappears from an output channel. *Signal* and *parameter* are legal for an input (output) *channel* if there is a transition with body `READ` *signal*(*parameter*) `FROM` *channel* (resp. `WRITE` *signal*(*parameter*) `INTO` *channel*). In accordance with the composition rule, if a process is combined with other process(es) into a block, then appearance of a new signal in its input channel corresponds to writing this signal into this channel by the partner process. Similarly, if a process is combined with other process(es) into a block, then disappearance of a signal from its output channel corresponds to reading this signal from this channel by the partner process. For simplicity, let us present the rule for appearance of a signal with a single implicit parameter.

RULE 5 (Appearance of a signal in an input channel)

$$\begin{pmatrix} \textit{Signal} \text{ and } \textit{parameter} \text{ are legal for an input } \textit{channel}, \\ T1 = T2,\ V1 = V2,\ ACT1 = ACT2,\ DEL1 = DEL2, \\ PUT(C1(channel), signal, parameter) = C2(channel), \\ (\forall channel' \neq channel \; : \; C1(channel') = C2(channel')) \end{pmatrix}$$
$$\models CNF1 < INV > CNF2.$$

Channel cleaning and program execution. The seventh and the eighth rules are the rules of cleaning the input and the output channels. The ninth rule for a process is the rule of program execution.

Time progress and starvation. The tenth rule deals with time progress. It concerns the case when nothing has changed except the value of the multiple clock and the synchronous local multiple clock of every current delay, and there is a transition marked by the active state such that its current delay has not exceeded the right bound of the corresponding time interval.

RULE 10 (Time progress)

$$\begin{pmatrix} T1 < T2,\ V1 = V2,\ C1 = C2,\ ACT1 = ACT2, \\ (\forall state \in ACT1 \; : \; DEL2(state) = DEL1(state) + T2 - T1), \\ (\forall state \notin ACT1 \; : \; DEL2(state) = 0), \\ (\forall state \in ACT1 \; : \; \exists transition \; : \; transition \text{ is marked by } state, \text{ and} \\ DEL1(state) \text{ does not exceed the right bound of time interval of } transition) \end{pmatrix}$$
$$\models CNF1 < INV > CNF2.$$

The eleventh rule deals with a starvation. It is similar to the time-progress rule, but in this case a current delay of an active state surpasses the right bounds of time intervals of all transitions marked by the active state. It means that the process failed to read or write a signal during the specified time interval.

RULE 11 (Starvation)

$$\left(\begin{array}{l} T1 < T2,\ V1 = V2,\ C1 = C2,\ ACT1 = ACT2, \\ (\forall state \in ACT1\ :\ DEL2(state) = DEL1(state) + T2 - T1), \\ (\forall state \notin ACT1\ :\ DEL2(state) = 0), \\ (\forall state \in ACT1\ :\ \forall transition\ : state \text{ marks } transition \Rightarrow \\ DEL1(state) \text{ exceeds the right bound of time interval of } transition) \end{array}\right)$$
$$\models CNF1 < INV > CNF2.$$

5 The semantics of logical specifications

The semantics of logical specifications is defined in terms of validity in the configurations. For every configuration CNF and every logical specification SPC, $CNF \models SPC$ means that the configuration belongs to the truth set of the logical specification, and $CNF \not\models SPC$ means the negation of this fact. In order to shorten the description of the semantics, let us fix a model with channel structures M, a scale MSR, and a configuration $CNF = (T, V, C, S)$. Let the relation $CNF \models SPC$ be defined by induction on structure of the diagram of a logical specification SPC.

Semantics of predicates. A predicate[3] can be a relation,a locator, a controller, or a checker. If SPC is a relation, then its diagram has the form $R(t1, \ldots, t2)$, where R is a relation symbol, and $t1$, ..., $t2$ are terms constructed from the operation symbols, variables and parameters of channels and $CNF \models SPC \Leftrightarrow (VAL_{CNF}(t1), \ldots, VAL_{CNF}(t2)) \in INT(R)$, where evaluation VAL_{CNF} for terms is defined by the ordinary rules. If SPC is a locator, then its diagram has the form `AT` *state* and $CNF \models SPC \Leftrightarrow state \in S.ACT$. If SPC is a controller, then its diagram has the form EMP *channel* or FUL *channel* and $CNF \models SPC \Leftrightarrow \mathtt{EMP}(C(channel))$, $CNF \models SPC \Leftrightarrow \mathtt{FUL}(C(channel))$, respectively. If SPC is a checker, then its diagram has the form *signal* `IN` *channel* or *signal* `RD` *channel*. In the former case, $CNF \models SPC$ iff there exists a *value* from DOM, such that there exists a pair $(signal, value)$ in $C(channel)$. In the latter case, $CNF \models SPC$ iff there exists a *graph* from DAT and a *value* from DOM, such that $GET(C(channel)) = (graph, signal, value)$.

Semantics of formulae. The diagram of a formula[3] can be a name of a predicate, a propositional combination of subformulae, a quantified subformula, or a dynamic expression.

If the diagram of SPC is a name of a predicate, then $CNF \models SPC \Leftrightarrow CNF \models PRD$, where PRD is the predicate with this name. If the diagram of SPC is a propositional combination, then its value is determined in a natural way.

[3] Please refer the syntax definition in Appendix 1.

If the diagram of SPC is $\forall$ *variable* DGR or $\exists$ *variable* DGR, where DGR is a formula diagram, then $CNF \models SPC$ iff for every/some configuration CNF' which agrees with CNF everywhere but *variable*, the following holds: $CNF' \models SPDGR$, where $SPDGR$ is a formula with diagram DGR.

If the diagram of SPC is M_B SYS M_T I_T DGR, where M_B is a modality `AB` or `EB`, SYS is an executional specification, M_T is a modality `AT` or `ET`, I_T is a time interval, and DGR is a formula diagram, then we have the following. Prefix "`AB/EB` SYS" means "for every/some fair behaviour of SYS". Prefix "`AT/ET` I_T" means "for every/some time moment in I_T".

6 Some meta-properties of Basic-REAL semantics

In [15] a property of invariance under stuttering was introduced. It means that expressible properties are not affected by duplication of some configurations. The following definitions, Theorem 1 and Corollary 1 state that bREAL enjoys the invariance under stuttering.

Let us fix the model M with structures for channels and the time measure MSR. For all behaviours $SEQA$ and $SEQB$, $SEQB$ is said to be obtained from $SEQA$ by copying (configurations) (or $SEQB$ is a *copy-extension* of $SEQA$) iff some configurations in $SEQA$ are duplicated in $SEQB$, i.e.,
$SEQA = CNF_0 \ldots CNF_i \ldots$, $SEQB = CNF_0 \ldots CNF_0 \ldots CNF_i \ldots CNF_i \ldots$.
For all sets of behaviours $SETA$ and $SETB$, $SETB$ is said to be obtained from $SETA$ by copying configurations (or $SETB$ is a *copy-extension* of $SETA$) iff
• every behaviour in $SETB$ is a copy-extension of a behaviour in $SETA$,
• for every behaviour in $SETA$ has a copy-extension in $SETB$.

Theorem 1. *Let SYS be an executional bREAL specifications. For all behaviours $SEQA$ and $SEQB$, if $SEQB$ is a copy-extension of $SEQA$, then $SEQB$ is a (fair) behaviour of SYS iff $SEQA$ is.*

Corollary 1. *For every set of configurations PRP and every interval I_T, for all sets of behaviours $SETA$ and $SETB$, if $SETB$ is a copy-extension of $SETA$, then dynamic/temporal expressions (M_B $SETB$ M_T I_T PRP) and (M_B $SETA$ M_T I_T PRP) are equivalent.*

An interleaving character of bREAL concurrency is stated in the following Theorem 2 and Corollary 2. It implies (in particular) an interleaving access to shared channels, i.e. impossibility of synchronous access to them.

Theorem 2. *Let SYS be an executional bREAL specification, let $CNF1$ and $CNF2$ be its configurations, and PR^1,... PR^k be all its subprocesses, and EVN be an event.*
(2.1) $CNF1 < INV > CNF2$ is a firing of SYS iff $CNF1/PR^i < INV > CNF2/PR^i$ is a firing of PR^i for every i in $1..k$.
(2.2) $CNF1 < EVN > CNF2$ is an active firing of SYS iff there exists a single j in $1..k$ such that $CNF1/PR^j < INV > CNF2/PR^j$ an active firing of PR^j.

Corollary 2. *Let SYS be an executional bREAL specification and PR^1,... PR^k be all its subprocesses. The set of all behaviours of SYS is equal to the set of behaviours CNF_0 ... CNF_n ... ... such that for every i in $1..k$ and for every $n \geq 0$ there exists an event EVN_n^i for which*
(1) $CNF_n/PR^i < EVN_n^i > CNF_{n+1}/PR^i$ is a firing of PR^i,
(2) $EVN_n^i \neq INV \Rightarrow EVN_n^j = INV$ for every $j \neq i$.

Proofs of Theorems 1 and 2 can be found in [21].

7 Verification techniques

Problem-oriented deduction for Basic-REAL. Our deductive approach is inspirited by the problem-oriented approach adopted in [8, 17, 18]. This approach assumes

- classification properties into classes of problems with respect to their semantics and syntax;
- formulation and justification of problem-oriented proof principles for every problem class.

We would like to point out that in general the proof-principles are not inference rules, since they exploit some higher order notions like sets, partitions, well-foundness, etc.

Let us explain our problem-oriented approach to deductive reasoning by a class of time-free progress properties, i.e., the class of the properties which can be presented in the bREAL logical specification by means of formulae with diagrams ($A \Rightarrow$ `AB` SYS `ET FROM NOW TILL FOREVER` B), where A and B are subdiagrams and SYS is a fixed name of an executional specification. Let us denote diagrams of this kind by $A \mapsto B$. That is, for every configuration $CNF \models (A \mapsto B)$ iff $CNF \models A$ implies that for every fair behaviour of SYS that starts from CNF, there exists a configuration CNF' from this behaviour such that $CNF' \models B$.

To formulate the proof principles, let us fix the executional specification SYS. Let SET' and SET'' (with possible subscripts) denote sets of configurations of SYS. The semantics of the expression $SET' \mapsto SET''$ is as follows: for every fair behaviour of SYS, if this behaviour begins in SET', then it contains a configuration in SET''. Thus, if SET' and SET'' are the truth sets of logical specifications with the diagrams A and B, respectively, then $SET' \mapsto SET''$ is equivalent to $A \mapsto B$. Note that the following concept of a *fair firing* is used: a fair firing is a firing that begins a fair behaviour.

1. Subset principle
$SET' \subseteq SET'' \vdash SET' \mapsto SET''$ or in the logical form $A \to B \vdash A \mapsto B$.

2. Partition/Union principle
For every set of indices I
$\{SET_i' \mapsto SET_i'' \ : \ i \in I\} \vdash (U_{i \in I} SET_i') \mapsto (U_{i \in I} SET_i'')$
or in the logical form $\forall i \in I.(A_i \mapsto B_i) \vdash (\exists i \in I \ : \ A_i) \mapsto (\exists i \in I \ : \ B_i)$.

3. Single step principle
$\vdash \{CNF'\} \ \mapsto \ \{CNF''$: there exists a fair firing $CNF' < EVN > CNF''\}$.

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

5. Principle of mapping to a well-founded set
Let SET be a set of configurations. Let WFS be a well-founded set (i.e., a partially ordered set without infinite descending chains) and let MIN be the set of minimal elements of the set WFS. Let $f : SET \overset{\text{partial}}{\longrightarrow} WFS$ be a partial function and let $f^- : WFS \overset{\text{partial}}{\longrightarrow} \mathcal{P}(SET)$ be the inverse function such that $f^-(w) = \{CNF \in SET : f(CNF) = w\}$ for every $v \in WFS$. Then the principle of mapping to a well-founded set is as follows:
$\big(\forall\ v \in WFS \setminus MIN\ :\ f^-(v) \mapsto \cup_{u<v} f^-(u)\big) \vdash f^-(WFS) \mapsto f^-(MIN)$.

Model-checking for Basic-REAL. There are many opportunities to exploit model-checking for verification of bREAL specifications. Let us enumerate some of them:

1. overall verification of finite-space time-free specifications;
2. overall verification of infinite-space time-bounded specifications;
3. validation of conditions of above two types in deductive proofs.

The first case is classical problem domain for model-checking. In the second case, when we would like to verify the time-bounded properties (e.g., that nothing bad happens in, say, first ten hours), the model may be safely cut off when the specification clock exceeds a time limit. It is done by detecting that a special `"time"` variable reaches a given value.

In the last case, a proof is designed manually in terms of problem-oriented proof-principles with extensive finite-space/time-free or infinite-space/time-bounded conditions and then their correctness is model-checked. Thus we integrate model-checking and deductive reasoning neither as a tactics, nor as a decision procedure, but as a checker for applicability of proof principles (see example in the next section).

Our model-checking tool is implemented as an application programming interface (API) and consists of the following three modules: a model-constructor for executional specifications, a translator from logical specifications, a kernel model-checker. Let us note that the model is an extended finite automaton whose states are configurations of the executional specification, and whose transitions are the firings of the transitions of the executional specification. It should be noted also that the kernel model-checker has been developed on the base of the faster model-checking algorithm for the μ-Calculus in finite models [10]. A preliminary tool version for time-free properties only has been presented in [2].

8 Verification of the example

Logical specification. We would like to discuss both time-free and timed properties of the system "Passenger and Vending Machine" presented in Section 2. Let us give at first a logical specification of the following time-free property: a good passenger will eventually get the ticket to the desired station. Or, more formally, for all ticket prices, for every station, the condition A leads to the

condition B where A is the conjunction of the five conditions: the machine is ready, the passenger wants to buy a ticket, no button is stuck, the indicator shows no information, the booking is empty, and B is the conjunction of the two conditions: there is a ticket in the booking window, and a station on the ticket is the needed one. Let us use the abbreviations "p" for "passenger" and "m" for "machine", respectively. The predicates for these (seven) conditions are `st_mach`, `st_pass`, `no_comm`, `no_info`, `no_tick`, `tick_in`, and `prop_st`, respectively. Let us denote the conjunction of the first five predicates by `init`: `st_mach` & `st_pass` & `no_comm` & `no_info` & `no_tick`. In this notation the time-free property can be expressed as a logical specification $SPEC1$ with diagram
$\forall$ `m.expenses` : $\forall$ `p.station` : $\big($`init`$\mapsto$`tick_in` & `prop_st`$\big)$.

Let us turn to timed property, namely: a quick passenger will eventually get the ticket to the desired station, a sluggish passenger either will never get the ticket to the desired station or may get it sometimes. Let $RushTime$ and $SlowTime$ be (parameterized) integers
$RushTime = Const1 * MaxPrice/MinCoin + Const2$,
$SlowTime = Const3 * MaxPrice/MinCoin + Const4$,
with MaxPrice for the maximal possible expenses, MinCoin for the smallest coin in use, $const1$–4 for parameters. Let $pFAST$ express the fact that the passenger is fast enough to drop coins in time, let $pSLOWEST$ mean that the passenger can't drop even the first coin because of a slow speed, and let $pSLOW$ express the intermediate case where the passenger may fail and may succeed. For each particular case only one of these predicates is true. Then let $SPEC2$ be a logical specification with the following diagram :

$\forall$ `m.expenses` : $\forall$ `p.station` :
((`init` & pFAST$\Rightarrow$ `AB` Σ `ET FROM NOW TILL` $RushTime$(`tick_in` & `prop_st`))&
(`init` & pSLOWEST$\Rightarrow$ `AB` Σ `ET FROM NOW TILL` $SlowTime$(`AT m.retcoin`))&
(`init` & pSLOW$\Rightarrow$ `EB` Σ `ET FROM NOW TILL` $SlowTime$(`tick_in` & `prop_st`))&
(`init` & pSLOW$\Rightarrow$ `EB` Σ `ET FROM NOW TILL` $SlowTime$ (`AT m.retcoin`)).

Coupling deduction and model-checking. For the case Σ_{fin} (when expenses are fixed integers) the model-checking technique has been applied to both time-free and timed variants. For example, the μ-formula corresponding to the SPEC1 is:
(`st_mach` & `st_pass` & `no_comm` & `no_info` & `no_tick`
$\Rightarrow \mu\, x.$ ((`tick_in` & `prop_st`) & ($\langle a\rangle true$ & $[a]x$))).

For the time-free case Σ_{par} (when expenses are parameters), coupling model-checking with deductive technique from Section 7 has been applied. In this case a proof of progress property `init` $\mapsto$ (`tick_in` & `prop_st`) has been decomposed in [21] (according to the transitivity principle) onto proofs of some "local" progress properties: `init` $\equiv P \mapsto P1 \mapsto P2 \mapsto P3 \mapsto Q \equiv$ (`tick_in`&`prop_st`). The last local property $P3 \mapsto Q$ is due to the subset principle. The properties $P \mapsto P1$ and $P2 \mapsto P3$ have been proved using the model-checking. But the step $P1 \mapsto P2$ is essentially inductive, since it has an uninterpreted parameter, viz., the price of the required ticket. Details of a proof of this progress property using the principle of mapping to a well-founded set with aid of model-checking are given in [21].

9 Conclusion

We presented the distributed systems specification language Basic-REAL as a formalism for the description of asynchronous systems. It has not been designed with the aim to replace formal description techniques such as SDL, but as a intermediate representation for them (in particular, for SDL).

The bREAL meets constraints 1–6 mentioned in Section 1. Indeed, a static subset of SDL (i.e., without dynamic process generation) can be translated into executional bREAL specifications in a property- and structure-conservative manner [23]. We implemented a prototype SDL2REAL translator which generates operationally equivalent REAL specifications from SDL ones. Given the static nature of bREAL, this translation does not cover the dynamical features of SDL (e.g., creation of process instances). An optimization is also performed to reduce the state space size: the unreachable generated state are removed from the bREAL specification. Due to these reasons, the equivalent specifications have approximately the same size and the same safety/progress properties. A presentation of this translation is a subject of a forthcoming publication. The formal semantics of bREAL described in Sections 3, 4 and 5 is simple, and it can be used to derive the important meta-properties like properties formulated in Section 6. Automatic proof of these and some other meta-properties of bREAL (e.g., bisimulation of synchronous communication between processes, i.e., rendezvous) is a topic for forthcoming research. The verification techniques discussed in Section 7, demonstrate that bREAL is a language suitable for representation and verification of properties. Study of problem classes other than progress properties and corresponding problem-oriented proof-principles is also a topic for further research.

The bREAL has the following advanced features for specifying distributed systems and their properties:

- The new time concept based on uninterpreted time units extends expressiveness of the language, and the time intervals associated with transitions allow the shortcoming of the timer concept in SDL [3, 6] to be overcome.
- The logical specification language is rather expressive due to the extension of real-time variants of CTL by time intervals and first order dynamic logic constructs.
- The expressive power of executional specifications is essentially increased by using fairness conditions and communication via channels with different structures (i.e., (un)bounded queues, stacks, bags, etc.).
- The formal semantics of bREAL provides the interaction of processes with the external environment, which simplifies specification and verification of distributed systems.

The project REAL is under development since 1991. Initial presentation of the project is in [20] where a sketch of formal semantics of language REAL was given. A complete definition of bREAL is presented in a technical report [21]. We intend to extend the bREAL language by dynamic process generation. The new REAL version — Dynamic-REAL — will increase the expressive power of bREAL and essentially extend the SDL subset which can be naturally translated to REAL.

Acknowledgements. The authors would like to thank the anonymous reviewers for valuable comments and suggestions.

References

1. Bergstra J.A., Middelburg C.A., Usenko Y.S. Discrete time process algebra and the semantics of SDL. Technical report SEN-R9809, CWI, June 1998.
2. Bodin E.V., Kozura V.E., Shilov N.V. Experiments with model checking for μ-calculus in specification and verification project REAL. Proc. of the Fifth New Zealand Formal Program Development Colloquium, IIMS Technical Report 99-1, 1999, 1–18.
3. Bošnački D. et al. Model checking SDL with Spin, TACAS/ETAPS 2000, Lect. Notes in Comp. Sci., 2000, 363–377.
4. Bozga M. et al. IF: An intermediate representation and validation environment for timed asynchronous systems, FM'99, Vol.I, Lect. Notes in Comp. Sci., 1999, v.1708, 307–327.
5. Broy M. Towards a formal foundation of the specification and description language SDL, Formal Aspects of Computing, v.3, n.1, 1991, 21–57.
6. Broy M., Grosu R. Klein C. Reconciling real-time with asynchronous message passing, Lect. Notes in Computer Sci., 1997, v.1313, 182–200.
7. Cavalli A.R., Horn F. 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.
8. Chandy K.M., Misra J. Parallel program design, Addison-Wesley, 1988.
9. Clarke E.M., Emerson E.A., Sistla A.P. Automatic verification of finite state concurrent systems using temporal logic specifications, ACM Trans. Programming Languages & Systems, 1986, **8, n. 2**, 244–263.
10. Cleaveland R., Klein M., Steffen B. Faster model checking for modal mu-calculus, Proceedings of CAV-92, Montreal, Canada, Lect. Notes in Comp. Sci., v.663, p.410-422.
11. Eschbach R. et al. On the formal semantics of SDL-2000: A compilation approach based on an abstract SDL machine, ASM 2000, Lect. Notes in Comp. Sci., 2000, v.1912, 242–265.
12. Gammelgaard A., Kristensen J.E. A correctness proof of a translation from SDL to CRL, Proc. of the 6th SDL Forum, 1993, 205–219.
13. Gibson P., Mery D. Telephone feature verification: translating SDL to TLA+, Report CRIN, Nancy, Dec. 1996.
14. Harel D. First-order dynamic logic, Lect. Notes in Comp. Sci., v.68, 1979.
15. Lamport L. Verification and specification of concurrent programs. - Lect. Notes in Comp. Sci., 1994, v.803, 347–374.
16. Leue S. Specifying real-time requirements for SDL specifications — A temporal logic-based approach, Proc. 15-th IFIP Intern. Symp. on Protocol Spec. Test. and Verif., 1995, Warsaw, p.19–34.
17. Manna Z., Pnueli A. The temporal logic of Reactive and Concurrent Systems. Springer-Verlag, Berlin/New York, 1991.
18. Manna Z., Pnueli A. Temporal verification of reactive systems: safety. Springer-Verlag, Berlin/New York, 1995.

19. Mery D., Mokkedem A. CROCOS: An integrated environment for interactive verification of SDL specifications, Lect. Notes in Comp. Sci., 1993, v.663, 343–356.
20. Nepomniaschy V.A., Shilov N.V. Real92: A combined specification language for systems and properties of real-time communicating processes, Proc. Int. Conf. on Formal Methods in Programming and Their Applications, Novosibirsk, 1993, Lect. Notes in Comp. Sci., v.735, 1993, 377-393.
21. Nepomniaschy V.A., Shilov N.V., Bodin E.V. A new language Basic-REAL for specification and verification of distributed system models, Report Nr. 65 of A.P. Ershov's Institute of Informatics Systems, Novosibirsk, 1999, 39 p. (also available at `http://www.iis.nsk.su/preprints/shilov/bre99/`)
22. Orava F. Formal semantics of SDL specifications, Proc. of 8-th IFIP Intern. Symp. on Protocol Spec. Test., and Verif., 1988, 143–157.
23. Valiullin R. Translation of static SDL specifications to Basic-REAL, Bachelor thesis. Novosibirsk State University, Department of Information Technologies, 2001 (in Russian).

Appendix 1. Formal syntax of Basic-REAL

specification:: executional-specification | logical-specification
executional-specification:: process | block
logical-specification:: predicate| formula
process:: process-head scale context fairness-conditions process-diagram
block:: block-head scale context fairness-conditions block-diagram subblocks
predicate:: predicate-head scale context systems predicate-diagram
formula:: formula-head scale context systems formula-diagram subformulae
process-head:: name : `PROCESS`
block-head:: name : `BLOCK`
predicate-head:: name : `PREDICATE`
formula-head:: name : `FORMULA`
scale:: {linear-equality-over-time-units | linear-inequality-over-time-units }*
context:: {type-definition | object-declaration }*
type-definition:: `TYPE` type `IS` type-expression
type-expression:: predefined-type | enumerated-type |
 type-expression `ARRAY OF` type-expression |
 type-expression `QUEUE OF` type-expression |
 type-expression `STACK OF` type-expression |
 type-expression `BAG OF` type-expression
predefined-type:: `INT` | `STR`
enumerated-type::name | name, enumerated-type | name; enumerated-type
subblocks:: { executional-specification }*
subformulae:: { logical-specification }*
object-declaration:: variable-declaration | channel-declaration
variable-declaration:: location appointment `VAR` variable `OF` type
appointment:: `QU` | `PR`
channel-declaration:: location role organization `CHN` channel `FOR` signal {`WITH PAR` parameter `OF` type-expression }*

role:: INP | OUT | INN
organization:: capacity structure | ELEMENTARY
capacity:: number-ELM | UNB
structure:: QUE | STACK | BAG
process-diagram:: { transition }*
transition:: state [**:**] body interval jump
body:: EXE program | READ signal-with-parameters-1 FROM channel |
    WRITE signal-with-parameters-2 INTO channel | CLEAN channel
program:: operator {; operator }*
operator:: variable :=expression | SKIP | ABRT | IF condition THEN program [ELSE program] FI | WHILE condition DO program OD | CASE program [ALT program] ESAC | LOOP program POOL
signal-with-parameters-1:: signal [(variable-list)
variable-list:: variable {, variable }*
signal-with-parameters-2:: signal [(value-list)
value-list:: expression {, expression }*
expression :: constant | variable | (expression) |expression operation-sign expression
operation-sign:: + | **-** | $*$ | / | APPLY | UPDATE
interval:: left-bound linear-time-expression right-bound linear-time-expression
left-bound:: AFTER | FROM
right-bound:: UNTIL | UPTO
jump:: JUMP state-list.
state-list:: state {, state }*.
block-diagram:: { route }*
route:: source CHN channel destination
source:: name | ENV
destination:: name | ENV
predicate-diagram:: relation | locator | controller | checker
relation:: expression relation-sign expression
locator:: AT state
controller:: EMP channel | channel IS EMPTY | FUL channel | channel IS OVERFULL
checker:: signal IN channel | signal RD channel
formula-diagram:: name | (propositional-combination) |
    (quantifier variable formula-diagram) |
    (behavioural-modality system temporal-modality interval formula-diagram)
quantifier:: $\forall$ | $\exists$
behavioural-modality:: EACH | SOME | AB | EB
temporal-modality $\Box$ | $\Diamond$ | AT | ET
state:: name
channel:: name
parameter:: name

Appendix 2. SDL-specification of "Passenger and Vending machine"

```
SYSTEM All;
BLOCK Passenger_and_Machine;
 SIGNAL coin (integer), light (integer), sigchange (integer),
  ticket (integer), sigstation (integer), request, return;
 SIGNALROUTE buttons FROM Passenger TO Machine
   WITH sigstation, return, request;
 SIGNALROUTE slot FROM Passenger TO Machine WITH coin;
 SIGNALROUTE indicator FROM Machine TO Passenger WITH light;
 SIGNALROUTE change FROM Machine TO Passenger WITH sigchange;
 SIGNALROUTE booking FROM Machine TO Passenger WITH ticket;
PROCESS Passenger;
  DCL sum, nominal, station, gotstation, gotsum integer;
  START; 'Initialize (constant) station';
   TASK decision := station;
   OUTPUT sigstation(decision); NEXTSTATE look;
  STATE look; INPUT light(sum); DECISION sum;
    (<=0): OUTPUT request; NEXTSTATE get;
    ELSE: /* choose a random coin from range 1..10 */
         TASK nominal := RANDOM(10);
         OUTPUT coin(nominal); NEXTSTATE look; ENDDECISION;
  ENDSTATE;
  STATE get; INPUT ticket(gotstation); INPUT sigchange(gotsum);
   STOP; ENDSTATE; ENDPROCESS;
PROCESS Machine;
  NEWTYPE price ARRAY (integer, integer) ENDNEWTYPE;
  DCL sum, nominal, station integer, expenses price;
  START; 'Initialize the (constant) expenses array';
  NEXTSTATE get_station;
  STATE get_station;
   INPUT sigstation(station); TASK sum := expenses(station);
   NEXTSTATE showcount;
  STATE showcount; OUTPUT light(sum); NEXTSTATE getcoin;
  STATE getcoin;
   INPUT coin(nominal); TASK sum := sum - nominal;
    NEXTSTATE showcount;
   INPUT return; OUTPUT sigchange(expenses(station) - sum); STOP;
   INPUT request; DECISION sum;
    (>0): OUTPUT light(sum); NEXTSTATE showcount;
    ELSE: OUTPUT ticket(station); OUTPUT sigchange( - sum );
     STOP; ENDDECISION;
  ENDSTATE; ENDPROCESS;
ENDBLOCK;
ENDSYSTEM;
```

Appendix 3. Basic-REAL specification of “Passenger and Vending machine”

Specification of the process “passenger”

```
passenger : PROCESS
OUT CHN buttons FOR sigstation WITH PAR name OF 1..100,
      FOR return, FOR request ;
OUT CHN slot FOR coin WITH PAR nominal OF integer ;
INP CHN indicator FOR light WITH PAR sum OF integer ;
INP CHN change FOR sigchange WITH PAR value OF integer ;
INP CHN booking FOR ticket WITH PAR name OF 1..100' ;
PR VAR sum, nominal OF integer,
PR VAR decision, gottenstation, station OF 1..100 ;
   { Fairness conditions}
   ¬AT start ; ¬AT press ; ¬AT continue ;
   ¬AT request ; ¬AT chcoin ; ¬AT drop ;
start EXE decision := station JUMP press.
press WRITE sigstation(decision) INTO buttons JUMP look.
look READ light(sum) FROM indicator JUMP continue.
continue EXE (sum <= 0)? JUMP request.
continue EXE (sum > 0)? JUMP chcoin.
chcoin EXE nominal := RANDOM(10) ; JUMP drop.
drop WRITE coin(nominal) INTO slot JUMP look.
request WRITE request INTO buttons JUMP get.
get READ ticket(gottenstation) FROM booking JUMP getchange.
getchange READ sigchange(sum) FROM change JUMP satisfaction.
```

Diagram of the process “machine”

```
start READ sigstation(station) FROM buttons JUMP defcount.
defcount EXE sum := expenses[station] JUMP showcount.
showcount WRITE light(sum) INTO indicator JUMP getcoin.
getcoin READ coin(nominal) FROM slot JUMP add.
getcoin READ return FROM buttons JUMP retcoin.
getcoin READ request FROM buttons JUMP check.
add EXE sum := sum - nominal ; JUMP showcount.
retcoin WRITE sigchange(expenses[station] - sum) INTO change
      JUMP finish.
check EXE (sum <= 0)? ; JUMP give.
check EXE (sum > 0)? JUMP showcount.
give WRITE ticket(station) INTO booking JUMP givechange.
givechange WRITE sigchange( - sum ) INTO change JUMP finish.
```