Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

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

A NEW LANGUAGE BASIC-REAL FOR SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEM MODELS

Preprint
65

Novosibirsk 1999

To design distributed systems the standard formal description techniques (FDT), such as Specification and Description Language (SDL), Extended State Transition Language (ESTELLE), Language of Temporal Ordering Specification (LOTOS) are used. The verification of FDT specifications (proving their safety, progress, and other properties) is still an open problem. The logical approach to the problem consists in the development of mathematical semantics for FDT, in a choice of a logical language for property presentation, and in the development of a corresponding proving methodology.

Our approach to verification of SDL specifications uses two-level scheme which combines the translation of SDL to a model high-level language with a verification method for the models. We use especially designed Basic-REAL (bREAL) specification language. This language with rigorous mathematical semantics is a version of the specification language REAL based on SDL and CTL.

The paper presents syntax and semantics of bREAL in both formal and informal levels. Some semantical properties of bREAL specifications (including invariance under stuttering and the interleaving property of concurrency) are proved. A specification and verification example ("Good Passenger and Slot-Machine") is also given.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

В. А. Непомнящий, Н. В. Шилов, Е. В. Бодин

НОВЫЙ ЯЗЫК BASIC-REAL ДЛЯ СПЕЦИФИКАЦИИ И ВЕРИФИКАЦИИ МОДЕЛЕЙ РАСПРЕДЕЛЕННЫХ СИСТЕМ

Препринт
65

Новосибирск 1999

Для разработки распределенных систем широко используются методы формального описания (МФО), такие как Specification and Description Language (SDL), Extended State Transition Language (ESTELLE), Language of Temporal Ordering Specification (LOTOS). Верификация МФО-спецификаций (доказательство свойств безопасности, прогресса, etc.) — актуальная научная проблема. Логический подход к этой проблеме подразумевает разработку точной математической семантики для МФО, выбор логического языка представления свойств и выбор соответствующей методики доказательства.

Предлагаемый подход к верификации SDL-спецификаций использует двухуровневую схему. На первом этапе SDL-спецификации транслируются в модельный язык высокого уровня, а на втором этапе верифицируются свойства модельных спецификаций. В качестве модельного языка используется специально разработанный язык Basic-REAL (bREAL). Этот язык имеет точную математическую семантику и является версией ранее описанного языка REAL, который основан на SDL и CTL.

В препринте на формальном и неформальном уровне описаны синтаксис и семантика языка bREAL, доказано несколько математических свойств этой семантики (включая устойчивость к заиканию и интерливинговый характер параллелизма), приведен пример спецификации и верификации системы “Касса-Пассажир”.

1. INTRODUCTION

The standard formal description techniques (FDT), such as SDL (Specification and Description Language) [27], ESTELLE (Extended State Transition Language) [12], LOTOS (Language of Temporal Ordering Specification) [11] are used for design of distributed systems. The verification of FDT specifications (proving their safety, progress, and other properties) is an actual open problem. The logical approach to the problem consists in the development of mathematical semantics for FDT, in a choice of a logical language for the property presentation and in the development of a corresponding proving methodology. Implementation of FDT specifications is another important problem and we would like to separate it from the verification problem.

The development of mathematical semantics lags behind the Formal Description Techniques evolution. A challenging Object-Oriented advance of FDTs in the 90-ies widens the gap between the modern FDTs and the mathematical background, between the distributed system specification and their mathematical verification. For example, the latest version of SDL-92 has no formal semantics oriented to verification [14] although interesting results have been obtained for fragments of SDL-88 [5, 13, 19, 24]. As far as concerns the property presentation of SDL specifications, linear (LTL) and branching time (CTL) temporal logics [7, 19], metric temporal logic (MTL) [1, 2, 18], Lamport's logic TLA+ [14], process algebra (CRL) [13] have been used. We would like to remark that using these formalisms for real-time properties often leads to complicated formulae.

A popular approach to proving properties of SDL specifications is a simulation of SDL specifications by special models which preserve the properties under consideration. Finite automata and other finite transition systems, Petri nets, UNITY programs, etc. are used as models of the specifications. Two methods are mainly used for proving properties of the models, namely, model-checking and inductive reasoning. The standard model-checking method [9] is successfully employed for finite systems, while inductive reasoning is oriented to parameterized and infinite state systems. The automatic nature is the main advantage of the model-checking technique, while inductive reasoning implies a manual design of proof outlines and the level of automatization is limited by the proof-checking. The combination of model-checking with abstraction (i.e., a homomorphical compression of a large, or parameterized, or infinite model into a relatively small finite model) [10] seriously extended a sphere of application of the model-checking method. But the abstraction method requires much additional work for each nontrivial case of study in order to ensure that the compression mapping is really homomorphical. The inductive reasoning methods have been proposed for proving safety properties of infinite transition systems [16] and properties of UNITY programs [8]. We suppose that a development of an SDL-oriented verification technique is of considerable importance

Our approach to verification of SDL specifications uses two-level scheme which combines the translation of SDL to a model high-level language with a verification method for the models. Since the use of a low-level language (like Petri nets or transition systems) restricts an expressive power of SDL, we use an especially designed Basic-REAL (bREAL) specification language. This language with rigorous mathematical semantics is a version of the specification language REAL [20, 21] based on SDL and CTL. A preliminary version of bREAL was shortly described in [22]. First verification experiments were described in [3].

The rest of the paper consists of five sections and Appendix. The main constructs of bREAL are explained informally in Section 2. Foundations of its formal semantics are described in Section 3. Semantical properties of bREAL specifications including invariance under stuttering and the interleaving property of concurrency are proved in Section 4. An example "Good Passenger and Slot-Machine" is given in Section 5. The method of verification of progress properties and the verification of the example is considered in Section 6. In conclusion the results and prospects of our approach are discussed. Appendix contains the formal syntax, semantics of bREAL, and specification of the system from the example.

The research is supported by the INTAS-RFBR grant under contract 95-0378.

2. THE GENERAL CONCEPTS OF BREAL

2.1. Representation of the distributed systems

Distributed systems are presented as executional specifications. The executional specifications have a hierarchical structure based on processes. The processes can be grouped into blocks which, in turn, make up higher level blocks.

Channels are used to describe the communication between such entities as processes, blocks, and the environment. Channels are intended for passing signals with possible parameters. Channels can have different inner structures, viz., queues, stacks, bags. A process specification in bREAL language, as well as an SDL process, describes a sequence of assignments, signal and parameter reading/writing from/into a channel, a channel cleaning.

In contrast to SDL, each action in bREAL is associated with a time interval, which specifies the duration of the action. Non-deterministic flow of control is admissible.

The bREAL language employs a new time concept, i.e., the concept of multiple clock synchronized by means of linear inequalities on its speeds.

The set of behaviours of an executional specification can be restricted by fairness conditions. We will consider only the “fair behaviours”, i.e., the behaviours in which each fairness condition holds infinitely often.

2.2. Representation of the properties

The properties of executional specifications are presented by logical specifications. Logical specifications expand the branching temporal logic CTL[9] by means of time intervals and dynamic logic constructs[15]. The formulae of this language are constructed from predicates by means of Boolean operations, variable quantification, and two kinds of modalities. The first kind is behavioural modality with the meaning “for all behaviours” and “there exists a behaviour”. The second kind of modalities is temporal modalities on multiple time interval with the meaning “for all time moments” and “there exists a time moment”.

The predicates are of the following four kinds:

1. relations between values;
2. locators of the control flow;
3. emptiness/fullness controllers for channels;
4. checkers of signals for channels.

2.3. Time concept

A scale is a finite set of linear (in)equalities, where time units are variables, and integer numbers are coefficients.

Each time unit presented in the scale is a tick of a special clock. A collection of special clocks is a multiple clock. The scale defines a set of restrictions for synchronization of special clock speeds in the multiple clock. The asynchronous clock system synchronized by means of the scale can be used as a multiple clock measuring time in multiticks. A multiinterval is a finite set of intervals with its boundaries presented by linear expressions with integer coefficients over time units. (The boundaries `NOW` and `INF` also can be used.) There are three popular semantical formalisms for time: discrete time, fictitious tick, and continuous time[1]. From a conceptual point of view, our approach is close to the fictitious clock model[25].

2.4. An outline of the syntax

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.

The concepts of time and scale were discussed above.

The context of a specification consists of type definitions, variable and channel declarations. Let us note that there are program and quantor variables in the bREAL language. The values of the quantor 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 in executional specifications.

The scheme of an executional specification consists of a diagram (described in the following sections) and a finite set of fairness conditions. The concepts of a fairness condition and fair behaviour were discussed above.

The 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.

A diagram of a formula is constructed from name(s) of (sub)specification(s) by propositional combinations, quantification over a quantor variable, and dynamic/temporal expressions.

2.5. Block diagrams

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. A block diagram can have both graphical and linear (textual) form. The graphical form of the block diagram is a marked graph whose vertices are rectangles marked by names of subblocks or the environment symbol, and the edges correspond to channels. The linear form of a block diagram is a list of edges of the graphical form. The two forms of the block diagram representation are equal in rights.

An informal semantics of blocks is as follows. All subblocks 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. As follows from the formal semantics of bREAL, a concurrent access to channels is impossible, i.e., at each moment only one subblock or the environment can change the contents of a channel.

2.6. Process diagrams

A process diagram is a generalization of a program scheme. It consists of transitions. Each of them consists of a control state, a body, a time interval, and a non-deterministic jump to next control states. The body of a transition determines the action to perform, such as

- 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 sequential program.

Each of the actions is performed instantly, but it can happen only within the time interval specified for the transition.

A process diagram can have both graphical and linear forms. The graphical form of a process diagram is auxiliary and can be omitted. It is an oriented graph, whose vertices are marked by the control states, while the edges represent the control flow. The shape of a vertex depends on the kind of the correspondent transition: ▭▷ ("pendant"), if the state marks a transition with a signal sending; ▭< ("flag"), if the state marks a transition with a signal reading; ▭▣ ("standard"), if the state marks a transition with a signal cleaning; ▭ ("box"), if the state marks a transition with a program execution; ⬭ ("oval"), if the state marks no transition. In contrast to the graphical form, the linear form of a process diagram is mandatory. It is a complete description of all transitions of the process.

Each process is non-deterministic but sequential, i.e., no concurrency is allowed inside a process. A process communicates with the other processes and the environment by means of input/output channels.

3. THE FORMAL SEMANTICS OF BREAL SPECIFICATIONS

3.1. Models with channel structures

To describe the semantics of the bREAL specifications (both executional and logical), we need the concept of a model with structures for channels.

Let us fix a bREAL specification. 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 the data structures for channels, i.e., a mapping which binds the channels with their contents. The structure of a channel is a set of all possible contents for the channel. We would like to consider each possible content as a finite oriented graph whose vertices are marked by signals with vectors of parameter values. For each channel structure DAT from CHS, two relations (EMP and FUL), two partial non-deterministic operations (PUT and GET) and one constant (INI) are defined below. (Here DOM^{PAR} is a set of parameter value vectors from DOM, *dom* and *ran* are the domain and the range of the functions, SIG is the set of signals admissible for the channel.

$EMP = \{INI\}$,
$PUT : (DAT \times SIG \times DOM^{PAR}) \to DAT$,
$dom(PUT) = (DAT \setminus FUL) \times SIG \times DOM^{PAR}$,
$ran(PUT) = DAT \setminus EMP$,
$GET : DAT \to (DAT \times SIG \times DOM^{PAR})$,
$dom(GET) = DAT \setminus EMP$,
$ran(GET) = ((DAT \setminus FUL) \times SIG \times DOM^{PAR})$,

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 and several edges connecting the new vertex to the other vertices 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 $GET(graph)$

$$\Updownarrow$$

$graph'$ differs from $graph$
by absence of a vertex and
all edges connecting it with the other vertices,
and $(signal, vector)$
is the marking of the removed vertex in $graph$.

The rules of removing are determined by the channel structure DAT itself. (Un)bounded queues, stacks, and multisets are standard channel structures with the standard semantics. For example, if a structure for a channel is a queue with admissible signals X, Y, and Z with the only parameter being integer-valued, then DAT is the set of finite sequences (chains, in terms of graphs) of pairs of the form (a signal, an integer). In this case the relation EMP is true on the empty sequence, and FUL is always false. Let *graph* = (X,10) $\longrightarrow$ (Y,-3) $\longrightarrow$ (Z,8). Then

PUT(*graph*,X,3) = (X,10) $\longrightarrow$ (Y,-3) $\longrightarrow$ (Z,8) $\longrightarrow$ (X,3),
GET(*graph*) = ((Y,-3) $\longrightarrow$ (Z,8), X, 10).

An example of a non-standard channel structure with the same set of signals is the structure DAT', consisting (as well as DAT) of all finite sequences of pairs of the form (a signal, an integer) with the same relations $EMP' = EMP$ and $FUL' = FUL$, but with the operation PUT' of insertion of a new vertex into the middle of the sequence and the operation GET' of removing a vertex from either of the ends of the sequence. The operation PUT' is non-deterministic on all sequences with odd length, and the operation GET' is non-deterministic on all sequences with at least two elements. Thus, for the same sequence *graph* we have:

PUT'(*graph*,X,3) = (X,10) $\longrightarrow$ (X,3) $\longrightarrow$ (Y,-3) $\longrightarrow$ (Z,8) and
PUT'(*graph*,X,3) = (X,10) $\longrightarrow$ (Y,-3) $\longrightarrow$ (X,3) $\longrightarrow$ (Z,8),
GET'(*graph*) = ((Y,-3) $\longrightarrow$ (Z,8), X, 10) and
GET'(*graph*) = ((X,10) $\longrightarrow$ (Y,-3), Z, 8).

3.2. Time scales and multiple clocks

A scale is a set (a system) of integer linear homogeneous (in)equalities with time units as variables. A time measure is a positive integer solution of the system. Thus, we will identify a time measure MSR with a mapping which associates each time *unit* with its value MSR(*unit*). For example, the scale

$1ing \leq 1min \leq 20ing, 30opr \leq 1sec \leq 100opr, 1min = 60sec$

has a solution

$sec = 100, min = 6000, opr = 2, ing = 600,$

and, therefore,

$MSR : sec \to 100, min \to 6000, opr \to 2, ing \to 600$

is a time measure in this scale. 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 each time *unit*, we have: T(*unit*)

$= [\frac{t}{MSR(unit)}]$. Thus, in the time measure fixed above, the following mapping $T: sec \rightarrow 246, min \rightarrow 4, opr \rightarrow 12315, ing \rightarrow 246$
is an indication of a multiple clock, since there exist even two integer numbers $t = 24630$ and $t = 24631$ such that
$T(\text{sec}) = [\frac{t}{MSR(sec)}] = 246$, $T(\text{min}) = [\frac{t}{MSR(min)}] = 4$, $T(\text{ing}) = [\frac{t}{MSR(ing)}]$ $= 41$, $T(\text{opr}) = [\frac{t}{MSR(opr)}] = 12315$. The mapping

$T': sec \rightarrow 24, min \rightarrow 4, opr \rightarrow 123, ing \rightarrow 46$
is not an indication of a multiple clock, since $T'(\text{min}) \times MSR(\text{min}) > (T'(\text{sec})$ $+ 1) \times MSR(\text{sec})$. Thus, each time unit in the time measure is a tact of a special clock, and the scale itself impose some synchronization for the special clock speeds in the multiple clock in order to ensure that all the inequalities hold.

3.3. Configurations of executional specifications

Let SYS be an executional specification the of bREAL language. Let us fix a model with structures for channels $M = (DOM, INT, CHS)$ and the time measure MSR. According to the syntax, SYS finally consists of the processes combined into (sub)blocks. Let PR^1, ... PR^k be all processes of SYS. An extended name (of a variable, state or channel) is the name itself preceded by the "path" of subblock names in accordance with the nesting. For example, if a system contains a subblock `b1` which in turn contains a subblock `b2` which contains a process `p` with a variable `v`, then the extended name of this variable in the system is `b1.b2.p.v`.

A configuration CNF of an executional specification SYS is a quadruple (T, V, C, S), where

- T is the indication of the multiple clock;
- V is the valuation of variables;
- C is the current contents of the channels;
- S is the current state of the control flow.

The valuation of variables is a mapping that maps each extended name of each variable of the processes PR^1, ..., PR^k to its current value from DOM. The current state of the control flow is a pair of mappings (ACT, DEL) that maps each extended name of each state of the processes PR^1, ..., PR^k to the Boolean characteristic of its current activity and the current delay (the indication of a special local multiple clock associated with each state). For an individual process, the Boolean characteristic of the control state activity is a characteristic function of a one-element set.

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 each *channel* and for all processes PR^i and PR^j, if *channel* is their common channel (an input channel for one of the processes and an output one for the other) then $C^i(channel) = C^j(channel)$. 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 each $0 \le i \le k$

- $T = T^i$;
- the current value V^i of the variables of the process PR^i coincides with the value V of its extended name;
- the contents C^i of the channels of the process PR^i coincides with the contents C of its extended name;
- the activity characteristic and the current delay S^i for the states of the process PR^i coincides with the activity characteristic and the current delay for its extended name.

When the subblocks are considered instead of the processes, the merge is defined in a similar way.

If BLK is a subblock 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 subblock BLK such that

- $T' = T$;
- for each *variable* $V'(variable) = V(BLK.variable)$;
- for each *channel* $C'(channel) = C(BLK.channel)$;
- for each *state* $S'(state) = S(BLK.state)$.

It is easy to prove by induction over the specification structure that a configuration of an executional specification is a merge of its projections to all its processes, i.e., $CNF = \mathcal{M}(CNF/PR^1, ..., CNF/PR^k)$.

3.4. The semantics of executional specifications

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

- sending a signal with parameters into a channel (WRiTing),
- receiving a signal with parameters from a channel (ReaDiNg),
- cleaning an input channel (CLeaNing INput),
- cleaning an output channel (CLeaNing OUTput),
- program execution (EXEcution),
- 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 each 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. A behaviour of an executional specification is said to be fair iff each of the specification's fairness conditions holds infinitely often in the configurations of this behaviour.

For blocks there is a unique step rule, namely, the composition rule. Informally, a behaviour of a block is an interleaving merge of consistent behaviours of its subblocks. Let us fix a model with channel structures $M = (DOM, INT, CHS)$ and a scale MSR. Then the composition rule for the block B, consisting of the subblocks B_1, ..., B_k, is as follows.
RULE 0 (Composition)
For each $i = 1, \ldots, k$, $\mathrm{CNF1}/\mathrm{B}_i < EVN/\mathrm{B}_i > \mathrm{CNF2}/\mathrm{B}_i$
$\models CNF1 < EVN > CNF2$.

The other eleven step rules deal with individual processes and the environment. They are given in Appendix 2 while an informal survey and examples are given below. For simplicity of presentation, 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*.

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.

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.

Let us fix a process and the two configurations, $CNF1 = (T1, V1, C1, S1)$ and $CNF2 = (T2, V2, C2, S2)$. For simplicity let us consider the case when each signal has the only parameter whose name is not used explicitly.
RULE 4 (Writing a signal)
The diagram has the transition
state `WRITE` *signal*(*variable*) `INTO` *channel interval* `JUMP` *jumpset*,
$T1 = T2$, $\forall state' DEL2(state') = 0$, $V1 = V2$,
$PUT(C1(channel), signal, V1(variable)) = C2(channel)$,
$\forall$ $channel' \neq channel$: $C1(channel') = C2(channel')$,
$state \in ACT1$, $nextstate \in ACT2$, $DEL1(state) \in interval$
$\models CNF1 < WRT(channel, signal, variable) > CNF2.$

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. 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.
RULE 5 (Appearance of a signal)
The diagram has the transition
state `WRITE` *signal*(*variable*) `INTO` *channel interval* `JUMP` *jumpset*,
$T1 = T2$, $V1 = V2$, $ACT1 = ACT2$,
PUT(C1(channel), signal, V1(variable)) $=$ C2(channel),
$\forall$ $channel' \neq channel$: $C1(channel') = C2(channel')$,
$DEL1(state) \in interval$, $\forall$ *state′ DEL2(state′)=0*
$\models CNF1 <$ INV $> CNF2$

The seventh and the eighth rules are the rules of cleaning the input and the output channels, respectively.

The ninth rule for a process is the rule of program execution.

RULE 9 (Program Execution)
The diagram has the transition *state* `EXE` *program interval* `JUMP` *jumpset*,
$T1 = T2$, $C1 = C2$, $(V1, V2) \in IO(program)$,
$state \in ACT1$, $nextstate \in ACT2$,
$DEL1(state) \in interval$, $\forall$ *state′: DEL2(state′)*=0
$\models CNF1 < EXE(program) > CNF2$

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 each 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.

The eleventh rule deals with a so called starvation. It is similar to the time-progress rule, but in this case all transitions marked by the active state have their current delays exceeded the right bound of the corresponding time interval. It means that the process failed to read or write a signal during the specified time interval.

3.5. The semantics of logical specifications

The semantics of logical specifications is defined in terms of validity in the configurations. For each configuration CNF and each logical specification SPC, CNF$\models$ SPC means that the configuration CNF belongs to the truth set of the logical specification SPC, 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 $M = (DOM, INT, CHS)$, a scale MSR, and a configuration CNF=(T, V, C, S). Let the relation $CNF \models$ be defined by induction on structure of the diagram of a logical specification SPC.

Induction basis: SPC is a predicate.

If SPC is a relation, then its diagram has the form $R(t1, \ldots, t2)$, where R is a relation symbol, and $t1, \ldots, t2$ are terms constructed from the operation symbols, variables and parameters of channels. Then $CNF \models$ SPC $\Leftrightarrow$ $(VAL_{CNF}(t1), \ldots, VAL_{CNF}(t2) \in INT(R)$, where the values $VAL_{CNF}(t1), \ldots, VAL_{CNF}(t2)$ are determined according to the ordinary rules.

If SPC is a locator, then its diagram has the form `AT` *state*. Then $CNF \models SPC \Leftrightarrow S(state) = true$.

If SPC is a controller, then its diagram has the form EMP *channel* or FUL *channel*. Then CNF$\models$ SPC $\Leftrightarrow$ `EMP`(C(channel)), CNF$\models$ SPC $\Leftrightarrow$ `FUL`(C(channel)), respectively.

If SPC is a checker, then its diagram has the form *signal* `IN` *channel* or *signal* `RD` *channel*. Then CNF$\models$ SPC iff there exists a *value* from DOM, such that there exists a vertex $(signal, value)$ in $C(channel)$; CNF$\models$ SPC $\Leftrightarrow$ there exists a *graph* from DAT and a *value* from DOM, such that $GET(C(channel)) = (graph, signal, value)$, respectively.

Induction step. 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. For example, if the diagram of SPC has the form $\neg$ A, where A is the diagram of a formula, then CNF$\models$ SPC $\Leftrightarrow$ CNF$\not\models$ SPA, where SPA is a formula differing from SPC with the diagram only, which is A.

If the diagram of SPC is $\forall$ *variable* A ($\exists$ *variable* A), where A is a formula diagram, then $CNF \models$ SPC iff for each (some, respectively) configuration CNF' differing from CNF at most with the evaluation of *variable*, the following holds: $CNF' \models$ SPA, where SPA is a formula differing from SPC with the diagram only, which is A.

If the diagram of SPC is M1 SYS M2 IT A, where M1 is a modality `AB` or `EB`, SYS is an executional specification, M2 is a modality `AT` or `ET`, IT is a time interval, and A is a formula diagram, then we have the following. Modality `AB` means "for All Behaviours" and `EB` means "there Exists a Behaviour". In general, M1 SB ranges over a set of behaviours SB and, in particular, this set consists of all fair behaviours of SYS. `AT` means "for All Time moments" and `ET` means "there Exists a Time moment". In general, M2 ST they range over a set of time moments ST and, in particular, this set consists of all time moments of IT.

For example: $CNF \models$ `AB` SYS `ET` IT A iff each fair behaviour of SYS starting with CNF, for some moment of time $T' \in$ IT, $CNF' \models$ SPA, where CNF' is a configuration from the behaviour with T' as the multiple clock and SPA is a formula differing from SPC with the diagram only, which is A.

If a logical specification has a diagram $\Rightarrow$ `EACH` SYS $\Diamond$`FROM NOW TILL` ∞ where SYS is the only element of a system list of the specification, then let us use the following logical macro leadsto ($\mapsto$) for presentation of the diagram of the specification. That is, $CNF\models$ (A $\mapsto$ B) iff $CNF \models$ A implies that for each fair behaviour of the executional specification SYS that starts from CNF, there exists a configuration CNF' from this behaviour such that $CNF'\models$ B.

4. THE PROPERTIES OF THE BREAL SEMANTICS

4.1. Invariance under stuttering

In [17] a property of invariance under stuttering was introduced. It means that the value of a formula is not affected when a finite number of copies of the elements of the sequence are added. The bREAL specification has a similar property.

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
$SEQA = CNF_0 \,...\, CNF_i \,...\, ...\, ...,$
$SEQB = CNF_0 ... CNF_0 \,...\, CNF_i ... CNF_i \,...\, ...\, ...,$
i.e., when some (possibly none) configurations from *SEQA* can be repeated several times in *SEQB*. For all sets of behaviours *SETA* and *SETB*, *SETB* is said to be obtained from *SETA* by copying configurations iff
• each behaviour in *SETB* is obtained from a behaviour from *SETA* by copying configurations,
• for each behaviour from *SETA* there exists a behaviour in *SETB* which is obtained from the behaviour from *SETA* by copying configurations.

The semantics of executional and logical bREAL specifications guarantees the invariance under stuttering in the following sense.

Theorem 1
Let SYS and PRP be an executional and a logical bREAL specifications.
1.1 For all behaviours *SEQA* and *SEQB*, if *SEQB* is obtained from *SEQA* by copying the configuration, then *SEQA* is a behaviour (fair behaviour) of the executional specification SYS iff *SEQB* is a behaviour (resp., a fair behaviour) of the same specification SYS.
1.2 For all sets of behaviours *SETA* and *SETB*, if *SETB* is obtained from *SETA* by copying the configuration, then for all modalities M1 and M2, each time interval DRTN (DuRaTioN) the validity sets of the logical specifications M1 *SETA* M2 DRTN PRP and M1 *SETB* M2 DRTN PRP are the same. (Let us remind that in general M1 can range over a set of behaviours as well as M2 can range over a set of time moments.)

Proof.

1.1 Let the conditions of the theorem hold. Hence, the behaviours *SEQA* and *SEQB* can be presented as
$SEQA = CNF_0 \,...\, CNF_i \,...\, ...\, ...\ ,$

$SEQB = CNF_0...CNF_0$ ... $CNF_i...CNF_i$ ... ...

If $SEQA$ is a behaviour of SYS, then there is a sequence of the events EVN$_0$ ... EVN_i ... ... such that the sequence
$CNF_0 < EVN_0 > CNF_1$, ... $CNF_i < EVN_i > CNF_{i+1}$, ...
is a sequence of firings. But for each configuration CNF the triple $CNF < INV > CNF$ is a firing due to the stutter rule. Therefore, the sequence
$CNF_0 <$ INV $> CNF_0$, ..., $CNF_0 < EVN_0 > CNF_1$,
...
$CNF_i <$ INV $> CNF_i$, ..., $CNF_i < EVN_0 > CNF_{i+1}$,
... ...
is a sequence of firings. Hence, $SEQB$ is a behaviour of SYS. If $SEQA$ is a fair behaviour of SYS, then, for each fairness condition CND of SYS, there is a countable sequence of indices $i_0, \dots i_j, \dots \dots \dots$ such that $CNF_{i_0} \models$ CND, ... $CNF_{i_j} \models$ CND, ..., and, therefore, $SEQB$ is also a fair behaviour of SYS.

If $SEQB$ is a behaviour of SYS, then there is a sequence of events
EVN$_{0,0}$ ... EVN_{0,k_0} EVN_0 ... ... EVN$_{i,0}$ ... EVN_{0,k_i} EVN_0 ... ...
such that the sequence
$CNF_0 < EVN_{0,0} > CNF_0$, ... $CNF_0 < EVN_{0,k0} > CNF_0$,
$CNF_0 < EVN_0 > CNF_1$,
... ...
$CNF_i < EVN_{i,0} > CNF_i$, ... $CNF_i < EVN_{i,ki} > CNF_i$,
$CNF_i < EVN_i > CNF_{i+1}$,
... ...
is a sequence of firings. Then the sequence
$CNF_0 < EVN_0 > CNF_1$, ... $CNF_i < EVN_0 > CNF_{i+1}$, ...
is a sequence of firings. Therefore, $SEQA$ is a behaviour of SYS. If $SEQB$ is a fair behaviour of SYS, then, for each fairness condition CND of SYS, there is a countable sequence of indices $i_0, \dots i_j, \dots \dots \dots$ such that $CNF_{i_0} \models$ CND, ... $CNF_{i_j} \models$ CND, ..., and, therefore, $SEQA$ is also a fair behaviour of SYS.

1.2 Let the conditions of the theorem hold. All the cases of different M1 and M2 are considered in a similar way, so, let M1 be `SOME`, and let M2 be $\Box$. Let CNF $= (T, V, C, S)$ be an arbitrary configuration.

If CNF $\models$ `SOME` $SETA$ $\Box$ DRTN PRP, then in $SETA$ there is a behaviour $SEQA = CNF_0$ ... CNF_i ... ..., such that CNF $= CNF_0$ and for each configuration $CNF_i = (T_i, V_i, C_i, S_i)$ in this behaviour with $T_i \in T_0 +$ DRTN holds $CNF_i \models$ PRP. Since $SETB$ is a copy-extension of $SETA$, then in *SETB* there is a behaviour $SEQB = CNF_0...CNF_0$ ... $CNF_i...CNF_i$ ... ... Since CNF $= CNF_0$ and for each configuration $CNF_i = (T_i, V_i, C_i, S_i)$ in this behaviour with $T_i \in T_0 +$ DRTN holds $CNF_i \models$ PRP, then CNF $\models$ `SOME` *SETB* $\Box$ DRTN PRP.

If CNF $\models$ `SOME` *SETB* $\Box$ DRTN PRP, and *SETB* is a copy-extension of *SETA*, then there is a behaviour *SEQB* in *SETB* which can be represented as $CNF_0...CNF_0$ ... $CNF_i...CNF_i$ ... ..., where CNF $= CNF_0$, for each configuration $CNF_i = (T_i, V_i, C_i, S_i)$ in this behaviour with $T_i \in T_0 +$ DRTN holds $CNF_i \models$ PRP, and CNF_0 ... CNF_i ... ... is a behaviour *SEQA* from *SETA*. Therefore, CNF $\models$ `SOME` *SETA* $\Box$ DRTN PRP.
$\Box$

4.2. The interleaving property of concurrency

Let us fix the model with channel structures and the time measure.

Theorem 2
Let SYS be an executional bREAL specification, $CNF1$ and $CNF2$ be a pair of configurations of SYS, PR^i, $i = 1, ..., k$ be all the processes of SYS, and EVN be an event.

(2.1) $CNF1 < INV > CNF2$ is a firing of SYS iff for each i, $1 \leq i \leq k$, $CNF1/\mathrm{PR}^i < INV > CNF2/\mathrm{PR}^i$ is a firing of the process PR^i.

(2.2) $CNF1 < EVN > CNF2$ is an active firing of SYS iff there exists j, $1 \leq j \leq k$, such that
$CNF1/\mathrm{PR}^j < EVN/\mathrm{PR}^j > CNF2/\mathrm{PR}^j$ is an active firing of the process PR^j, and for each i, $i \neq j$, $1 \leq i \leq k$, $CNF1/\mathrm{PR}^i < INV > CNF2/\mathrm{PR}^i$ is a passive firing of the process PR^i.

Proof is made by induction over the hierarchical structure of the executional specifications.

Induction base: SYS is a process. Then SYS coincides with PR^1, $k = 1$, and the statements (2.1) and (2.2) are evident.

Induction step: SYS is a block and it consists of subblocks $\mathrm{B}_1, \ldots, \mathrm{B}_m$ for which the statement of the theorem holds. Let the block B_i consist of the processes $\mathrm{PR}^{i,j}$, $1 \leq j \leq k_i$. Let us note that the set of all processes $\mathrm{PR}^{i,j}$ coincides with the set of all processes PR^i, i.e., the double indexing only distributes the processes over the topmost subblocks, and it is only a renumeration of the processes.

(2.1) Let $CNF1 < INV > CNF2$ be a firing of SYS. Then the composition rule has been applied (since this is the only possible rule for the blocks), and, hence, its premise is true. Therefore,

$\forall i, i = 1, ..., m$, $CNF1/\mathrm{B}_i < INV > CNF2/\mathrm{B}_i$ is a firing of B_i.
Due to the induction assumption, for all B_i the statement (2.1) holds, i.e.,
$CNF1/\mathrm{B}_i < INV > CNF2/\mathrm{B}_i$ is a firing of B_i iff
for each l, $1 \leq l \leq k_i$,
$CNF1/\mathrm{PR}^{i,l} < INV > CNF2/\mathrm{PR}^{i,l}$ is a firing of the process $\mathrm{PR}^{i,l}$.

On the other side, if for each i, $1 \leq i \leq k$, $CNF1/\mathrm{PR}^i < INV > CNF2/\mathrm{PR}^i$ is a firing of the process PR^i then, due to the induction assumption (2.1), for all topmost subblocks B_i ($i = 1, ..., m$) of the specification SYS, $CNF1/\mathrm{B}_i < INV > CNF2/\mathrm{B}_i$ is a firing of the subblock B_i. So, the conditions of the composition rule hold, which implies that $CNF1 < INV > CNF2$ is a firing of SYS.

(2.2) Let $CNF1 < EVN > CNF2$ be an active firing of SYS. Then (as SYS is not a process) the composition rule has been applied. Therefore, the condition of the composition rule holds, i.e., for all $i = 1, \ldots, m$ $CNF1/\mathrm{B}_i < EVN/\mathrm{B}_i > CNF2/\mathrm{B}_i$ is a firing of B_i. Due to the definition of the projection, there is no event EVN which could have several active projections. For example, if $EVN = RDN(\mathit{channel}, \mathit{signal}, \mathit{variable})$ or $EVN = CLNIN(\mathit{channel})$, then there is the only process, for which the *channel* is an input one, and if $EVN = EXE(\mathit{program})$, then there is the only process to which the *program* belongs, since the extended names of the variables contain the name of the process, and the "empty" programs (`SKIP`) are indexed by the name of the process. Thus, there is the only n such that $EVN/\mathrm{B}_n = EVN$ and $EVN/\mathrm{B}_i = INV$ ($i \neq n$). Due to the induction assumption for this B_n, there is j, $1 \leq j \leq k_n$, such that $CNF1/\mathrm{PR}^{n,j} < EVN > CNF2/\mathrm{PR}^{n,j}$ is an active firing of the process $\mathrm{PR}^{n,j}$ (i.e., $EVN/\mathrm{PR}^{n,j} = EVN$), and for each l, $1 \leq l \leq k_n$, $l \neq j$ $CNF1/\mathrm{PR}^{n,l} < INV > CNF2/\mathrm{PR}^{n,l}$ is a firing of the process $\mathrm{PR}^{n,l}$. Since each process belongs to at most one subblock at every hierarchical level, then there is the only i such that $CNF1/\mathrm{B}_i < EVN > CNF2/\mathrm{B}_i$.

On the other side, if EVN is an active event and there is j, $1 \leq j \leq k$, such that $CNF1/\mathrm{PR}^j < EVN > CNF2/\mathrm{PR}^j$ is an active firing of the process PR^j, and for each i, $1 \leq i \leq k$, $i \neq j$ $CNF1/\mathrm{PR}^i < INV > CNF2/\mathrm{PR}^i$ is a firing of the process PR^i. Then there is an "active" topmost subblock containing the "active" process. Let the process PR^j belong to the subblock B_n of the topmost level. For this subblock $CNF1/\mathrm{B}_n < EVN > CNF2/\mathrm{B}_n$ and $CNF1/\mathrm{B}_i < INV > CNF2/\mathrm{B}_i$ ($i \neq n$). Let us apply the induction assumption (2.2) and the (already proven) statement (2.1). Then the conditions of the composition rule hold, since $EVN = EVN/\mathrm{B}_j$, $INV = EVN/\mathrm{B}_i$ $(i \neq j)$.

Due to the composition rule, $CNF1 < EVN > CNF2$ is an active firing of SYS. □

Corollary (The semantics of concurrency between the processes)
Let SYS be an executional bREAL specification. The set of all behaviours of SYS coincides with the set of behaviours
CNF_0 ... CNF_n ... ... such that for each $1 \leq i \leq k$, and for each $n \geq 0$ there exists an event EVN^i_n for which
(1) $CNF_n/\mathrm{PR}^i < EVN^i_n > CNF_{n+1}/\mathrm{PR}^i$ is a firing of the process PR^i,
(2) $EVN^i_n \neq INV \Rightarrow EVN^j_n = INV$ for all $j \neq i$.
Proof. A behaviour of an executional specification is a countable sequence of configurations CNF_0 ... CNF_n ... ..., for which there exists a sequence of events EVN_0 ... EVN_n ... ...such that $CNF_0 < EVN_0 > CNF_1$ ... $CNF_n < EVN_n > CNF_{n+1}$ ... ...
is a sequence of firings. Now it is sufficient to refer to the Theorem 2.
□

5. AN EXAMPLE "PASSENGER AND SLOT-MACHINE"

Let us consider the following example: a protocol of serving a good 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.
A good passenger knows a station that he/she needs, has enough money and can:
• press buttons on the keyboard;
• drop coins into the slot;
• see the reading of the indicator;
• get coins from the change tray;
• get a ticket from the booking window.
Informally, the protocol of serving a good passenger is as follows. The seance begins from the passenger pressing a button corresponding to the desired station on the keyboard. The vending-machine having received the station name shows the price on the indicator. Then the following loop begins: the

passenger looks at the indicator and, if he/she sees a non-zero sum, chooses a coin and drops it into the slot, then the vending-machine subtracts the nominal of the coin from the sum to be received from the passenger and shows the new value on the indicator. In general, a passenger can quit this loop at any time by pressing the cancel button, so that the vending-machine must return through the change tray all the coins received so far. A good passenger, however, does not use the option and when reads zero from the indicator he/she presses the ticket request button. The vending-machine receives the request and prints the ticket with the station name, and the passenger takes the ticket. The seance is over for the vending-machine when it returns all sum to a passenger or prints out a ticket. The seance is over for the passenger when he/she takes the money from the change tray (which the good passenger never does) or takes the ticket that he/she needed.

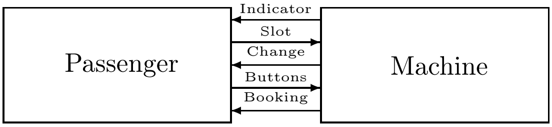

Figure 1. Block:"passenger_and_machine"

This protocol can be specified as a block which consists of two processes. A sketch version of this executional specification is presented below. The diagram of this block `passenger_and_machine` is presented on Fig. 1 on page 23. We would like to avoid some syntactical details related to contexts and scales, so we will present in details the diagram and the processes only. The context of the block consists of inner channels' declarations and coincides with the channels' declarations in the context of a logical specification presented in the next subsection. A sketch of the process `machine` will consists of the diagram (in the graphical form — see Fig. 2 on page 24) and the fairness conditions. We would like to remark that three transitions correspond to the state `getcoin` in the linear form of the diagram: the first corresponds to receiving a coin, the second corresponds to receiving the request ticket command and the last corresponds to receiving the cancel command. The fairness conditions for `machine` are:
`¬AT start ; ¬AT defcount ; ¬AT showcount ; ¬AT add ; ¬AT retcoin ; ¬AT check ; ¬AT give ;`
i.e., a behaviour of the process `machine` is fair if the process can stay forever in a state other than waiting for input signals. A sketch of the process `passenger` is presented by the graphical form of the diagram (see Fig. 3

on the page 25). In the state `continue` “passenger decides” what he/she must do: to choose a coin (`chcoin`) or request a ticket (`request`). So, two transitions correspond to this state in the linear form of the diagram. The fairness conditions of `passenger` are similar to the fairness conditions of `machine`: a behaviour of the process `passenger` is fair if the process can stay forever in a state other than waiting for input signals. The linear forms of both processes are presented in Appendix 3.

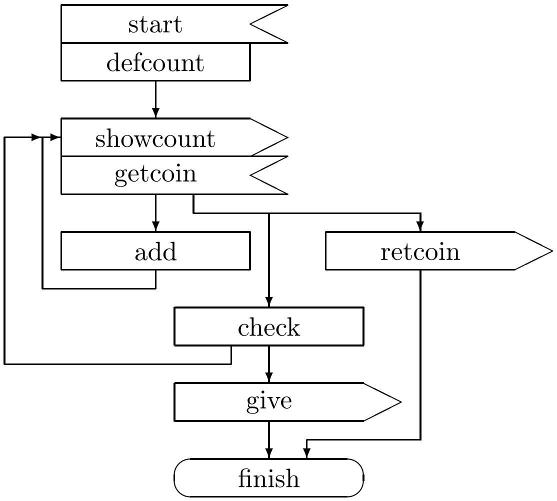

Figure 2. Process: “machine”

The protocol property to be specified is the following: the protocol of serving a good passenger guarantees that the vending-maching gives a ticket to the required station to the passenger. Let us give a logical specification of this property.

property : FORM
INN CHN buttons FOR station WITH PAR name OF ’a’...’z’,
    FOR return, FOR request ;
INN CHN slot FOR coin WITH PAR nominal OF integer ;
INN CHN indicator FOR light WITH PAR sum OF integer ;
INN CHN change FOR change WITH PAR value OF integer ;
INN CHN booking FOR ticket WITH PAR name OF ’a’...’z’ ;
QU VAR passenger.station OF ’a’..’z’ ;
QU VAR machine.expenses OF array [’a’..’z’] of integer ;

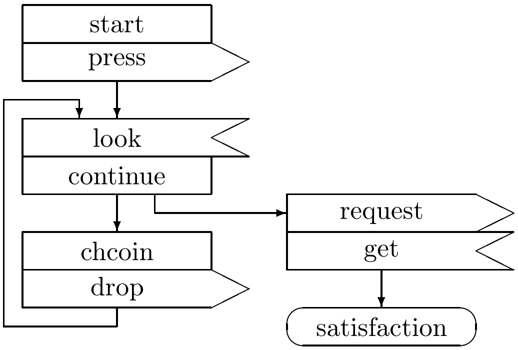

Figure 3. Process: “passenger”

passenger_and_machine
AL machine.expenses .
AL passenger.station .
((start_of_machine & start_of_passenger &
no_commands_on_buttons & no_information_on_indicator &
no_ticket_in_booking)
$\mapsto$
ticket_in_booking & proper_station))

Thus, for all ticket prices, for each station that passenger may need, the validity of the conjunction of the five conditions:
the machine is ready (start_of_machine), the passenger wants to buy a ticket (start_of_passenger), no button is stuck (no_commands_on_buttons), the indicator shows no information (no_information_on_indicator), the booking is empty (no_ticket_in_booking),
leads to the validity of the conjunction of the two conditions: there is a ticket in the booking window (ticket_in_booking) and the station on the ticket is the one that the passenger needs (proper_station).

6. VERIFICATION OF PROGRESS PROPERTIES

6.1. Proof principles

We consider the verification of bREAL specifications as a “proving” of properties presented by logical specifications for associated systems presented by executional specifications. Our approach is close to the one of

[16] and it consists of

— the classification of logical specifications according to their syntactical structure to the problem classes,

— the development of correspondent proof metaprinciples for each problem class,

— the design of proof outlines in terms of proof principles in accordance with the syntactical structure of properties and systems.

It is necessary to note that the proof principles differ from the inference rules, since the inference rules are purely syntactical and they are applied in the framework of a fixed axiomatic theory, while the proof principles are used in the framework of a metatheory which usually includes set theory or arithmetics.

We would like to illustrate our approach by the 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 whose diagrams have the form A $\mapsto$ B, where A and B are subdiagrams. To formulate the proof principles, let us fix the executional specification SYS which is the only element of the system list of the formula. Let SET' and SET'' (with possible subscripts) denote sets of configurations of the system SYS. The semantics of the expression $SET' \mapsto SET''$ is as follows: for each configuration $CNF' \in SET'$ and each fair behaviour of the executional specification SYS, if this behaviour begins from CNF', then it contains the configuration $CNF'' \in SET''$. 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
$A \mapsto B$. When formulating the proof principles, the concept of a *fair firing* is used. A fair firing is a firing that begins a fair behaviour of the system.

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

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

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

4. Transitivity principle.
$SET' \mapsto SET''_1,\ SET''_1 \mapsto SET''$
$\vdash SET' \mapsto SET''$

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 WFS be a well-founded set, i.e., a set with a partial order without infinite descending chains. Let MIN be the set of minimal elements of the set WFS. Let f be a partial function from the set of configurations SET to the well-founded set WFS. Let us denote by f^- a function such that for $C2 \in WFS$ holds $f^-(C2) = \{C1|C1 \in SET$ and $f(C1) = C2\}$. For an arbitrary function $g(x)$ and a set S of its arguments, let $g(S) = \{g(s)|s \in S\}$. Then the principle of mapping to a well-founded set is as follows:
$\forall v \in WFS \setminus MIN.f^-(v) {\mapsto} f^-(\{u|u < v\}) \vdash f^-(WFS) {\mapsto} f^-(MIN)$.

6.2. Verification of the example

Let us, for simplicity in this section, use the abbreviations "p" for "passenger" and "m" for "machine", respectively. Thus, we are proving the following:
`AL m.expenses, p.station`
((`AT p.start`) & (`AT m.start`) &
(`buttons IS EMPTY`) & (`indicator IS EMPTY`) & (`booking IS EMPTY`) $\mapsto$
$\mapsto$ ((`ticket IN booking`) & `booking.ticket.name = p.station`))
With respect to a standard inference rule GEN(eralization), $A \models \forall x.A$, it is sufficient to prove the following progress property: $P \mapsto Q$, where P and Q are obvious from the above. We would like to point out that it is a property of *a finite* but *parametrised* system.

For a finite set S of states, let `AT` $S = \vee_{s \in S}$`AT` s. Let us introduce the following denotations.
$S0 =$ {`p.look`, `p.continue`, `p.chcoin`, `p.drop`},
$S1 =$ {`m.showcount`, `m.getcoin`, `m.add`},
P1: `AT` $S0$ & `AT` $S1$ & `m.station = p.station` & `m.sum` $\leq$`p.sum` & `buttons IS EMPTY` & `booking IS EMPTY` & (`AT m.showcount` $\vee$
`m.sum` $\leq$`indicator.light.sum` $\leq$`p.sum`),
P2: `AT p.continue` & `AT` $S1$ & `m.station` $=$ `p.station` & `p.sum` ≤ 0 & `m.sum` ≤ 0 & `buttons IS EMPTY` & `booking IS EMPTY`,
P3: `AT p.get` & `m.station` $=$ `p.station` & `buttons IS EMPTY` & `ticket IN booking` & `booking.ticket.name` $=$ `p.station`.

According to the transitivity principle, to prove this progress property, it is sufficent to prove the correctness of all "local" progress properties. $P {\mapsto} P1 {\mapsto} P2 {\mapsto} P3 {\mapsto} Q$. But the property $P3 {\mapsto} Q$ is obvious, since it is a particular case of the subset principle. The properties $P {\mapsto} P1$ и $P2 {\mapsto} P3$ are easy to prove using the proof principles, or model-checking[4]. But the step $P1 \mapsto P2$ is essentially inductive, since it has an uninterpreted parameter, viz., the price of the required ticket.

Let us consider the proof of the progress property $P1 \mapsto P2$, using the principle of mapping to a well-founded set. 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 a1≤a2 and b1<b2, or a1<a2 and b1≤b2. Then $MIN = \{(0,0)\}$. Let SET be the set of configurations such that $CNF \models P2$. Let POS mean the operation of getting the positive part of a number, i.e., $POS(c) = c$, if > 0, and 0 otherwise. Let the partial function $f : SET \to WFS$ be the following: $f(CNF) = (POS(\texttt{m.sum}), POS(\texttt{p.sum}))$, if $CNF \models P2$, and undefined otherwise. Let us prove that $f^{-}(WFS) \mapsto f^{-}(MIN)$.

Let us choose $v = (a, b) \in WFS \setminus MIN$ and a fair firing $CNF' < EVN > CNF''$ such that $f(CNF') = v$. Since $CNF' \models P2$, then $CNF' \models$ `AT` $S0$ and $CNF' \models$ `AT` $S1$. Thus, in the configuration CNF' the following events are possible: RDN(indicator, light, `p.sum`), EXE(`IF p.sum` $\leq$`0 THEN SKIP ELSE ABORT`), EXE(`IF p.sum >0 THEN SKIP ELSE ABORT`), EXE("choosing the value of p.nominal"), WRT(slot, coin, p.nominal), WRT(indicator, light, `m.sum`), RDN(slot, coin, m.nominal), EXE("decrementing the value of `m.sum`"). Since $CNF' \models P2$, in the configuration CNF' holds `m.sum` $\leq$`p.sum` and

`AT m.showcount` $\vee$`m.sum` $\leq$`indicator.light.sum` $\leq$`p.sum`.

Let c be the value of m.nominal in CNF', and d, the value of `indicator.light.sum` in CNF'. Then

$$f(CNF'') \in \{(a, b), (a, d), (a - c, b)\},$$

i.e., $f(CNF'') \leq (a, b)$. According to the rules of the structural operational semantics we have: $\{CNF^0 | f(CNF^0) = v\} \mapsto SET1 = \{CNF^1 | \ f(CNF^1) \leq v$ & $CNF^1 \models$ `AT` *m.getcoin*$\}$. Let $SET2 = \{CNF^2 | \ f(CNF^2) \leq v$& $CNF^2 \models$ `AT` *m.getcoin* & $CNF^2 \models$ *slot* `IS FULL`$\}$, $SET3 = \{CNF^3 | f(CNF^3) \leq v$ & $CNF^3 \models$ `AT` *m.getcoin* & $CNF^3 \models$ *slot* `IS EMPTY`$\}$,
$SET4 = \{CNF^4 \mid f(CNF^4) \leq v$ & $CNF^4 \models$ `AT` *m.add*$\}$. Then $SET1 = SET2 \bigcup SET3$ and $SET2 \mapsto SET4$. We have
$SET3 \mapsto \{CNF^5 | f(CNF^5) \leq v$ & $CNF^5 \models$ (*slot* `IS EMPTY` & `AT` *m.getcoin* & `AT p.drop`)$\}$
$\mapsto SET2$. Therefore, $SET1 \mapsto SET4$. But $SET4 \mapsto \{CNF^6 | f(CNF^6) < v\}$. Therefore, for each fair behaviour CNF_0 ... CNF_i ..., if $f(CNF_0) = v$, then $\exists\ i > 0$ such that $f(CNF_i) < v$. Thus, according to the principle of partial

mapping to a well-founded set, $f^-(WFS) \mapsto f^-(MIN)$. Now it is sufficient to apply the transitivity principle:
$P1 \mapsto f^-(WFS), f^-(WFS) \mapsto f^-(MIN),\ f^-(MIN) \mapsto\ P2 \vdash P1 \mapsto P2$.

7. CONCLUSION

In distinction to other specification languages, bREAL is a combined language which allows us to represent both distributed real-time systems and their properties.

The distinctive features of bREAL are

- a simple syntax allowing the graphic presentation of executable specifications in SDL style;
- a complete structural operational semantics in a close to Plotkin style [26] allowing important semantical properties of executable specifications to be proved;
- a new time concept based on uninterpreted time units which extends expressiveness of the language;
- the logical specification language which is an extension of CTL with time intervals and first-order dynamic logic constructs.

When compared to SDL, bREAL has the following advantages:

- timed intervals related to transitions allow us to overcome the lacks of the timer concept in SDL [6];
- nondeterministic transitions are widely used;
- local interactions are done by means of bounded/unbounded channels with a variety of data structures which are defined like abstract data structures and can represent queues, stacks, and bags.

The project REAL is rapidly progressing. It includes CASE systems for translation from a SDL subset to bREAL, and for modelling of the executable bREAL specifications, as well as a model-checker oriented to interactive verification of properties expressed by logical bREAL specifications. We intend to extend our verification method to logical bREAL specifications with nondegenerated timed intervals.

A challenging Object-Oriented advance of Formal Description Techniques in 90-ies (ex., SDL-92) and a recent emergence of industrial quasi-standard languages for specification, visualization, design and documentation of artifacts of software systems without a well-defined observable sound formal semantics (the Unified Modelling Language (UML) [28] and Object Constraint Language (OCL) [23]) lead to an urgent necessity of further development of specification and verification languages with well-defined formal semantics and proof-search methodology toward capturing basic features of Object-Oriented Programming, i.e. encapsulation, inheritance, and polymorphism. We consider an extension of formal syntax and semantics of **Basic** level toward capturing several of these OO features, as well as an extension of bREAL toward the compatibility with UML and OCL as a perspective of further development of **REAL**.

REFERENCES

1. Alur R., Henzinger T.A. Logics and Models of Real Time: A Survey. Lecture Notes in Computer Science, 1992, 600, 74–106.
2. Alur R., Henzinger T.A. Real-time logics: complexity and expressiveness. - Information and Computation, 1993, v.104, N1, 35–77.
3. Bodin E.V. Approaches to the verification of Basic-REAL specifications. Problems of specifications and verifications of concurrent systems. Novosibirsk, Inst. of Inf. Syst., 1995. (in Russian)
4. 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, p. 1–18.
5. Broy M. (1991) Towards a formal foundation of the specification and description language SDL. Formal Aspects of Computing, **3, n.1**, 21–57.
6. Broy M., Grosu R. Klein C. Reconciling real-time with asynchronous message passing.-Lect. Notes in Computer Sci., 1997, 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. (1988) Parallel program design, Addison-Wesley.
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. Clarke E.M., Grumberg O., Long D.E. Model checking and abstraction. - ACM Trans. Progr. Languages & Systems, 1994, V.16, N5, 1512–1542.
11. Definition of the Temporal Ordering Specification Language LOTOS. ISO/TC 97/SC 16/WG 1 n.229, 1984, 50 p.
12. A Formal Description Technique Based on an Extended State Transition Model. ISO/TC 97/SC 21 177, 1983, 120 p.
13. Gammelgaard A., Kristensen J.E. A correctness proof of a translation from SDL to CRL, Proc. of the 6th SDL Forum, 1993, 205–219.
14. Gibson P., Mery D. Telephone feature verification: translating SDL to TLA+. - Report CRIN, Nancy, Dec. 1996.
15. Harel D. First-order dynamic logic. Lecture Notes in Computer Science, 1979, **68**.
16. Henzinger A., Manna Z., Pnueli A. Temporal proof metodologies for real-time systems. Proc. of Symp. on POPL, 1991, 353–366.
17. Lamport L. Verification and specification of concurrent programs. - Lect. Notes in

Comp. Sci., 1994, v.803, 347–374.
18. 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.
19. Mery D., Mokkedem A. CROCOS: An integrated environment for interactive verification of SDL specifications. Lecture Notes in Computer Science, 1993, **663**, 343–356.
20. Nepomniaschy V.A., Shilov N.V. A specification language for systems and properties of real-time communicating processes. Methods of theoretical and system programming. Novosibirsk, 1991, 32–45. (in Russian)
21. Nepomniaschy V.A., Shilov N.V. Real92: A combined specification language for systems and properties of real-time communicating processes. "Programmirovanie", 1993, N6, P. 64–80. (in Russian)
22. Nepomniaschy V.A., Shilov N.V., Bodin E.V. A concurrent systems specification language based on SDL & CTL. Proc. of Workshop on Concurrency, Specifications & Programming, Berlin, 1994, Humboldt University, Informatik-Bericht Nr.36, 1994, 15–26.
23. The Object Constraint Language. http://www.software.ibm.com/ad/ocl/
24. Orava F. Formal semantics of SDL specifications. Proc. of 8-th IFIP Intern. Symp. on Protocol Spec. Test., and Verif., 1988, 143–157.
25. Ostroff J.S. Automated verification of timed transition models. Lecture Notes in Computer Science, 1990, **407**, 247–256.
26. Plotkin G.D. A structure approach to operational semantics. 1981, Technical report FN-19, Aarhus University, DAIMI, Denmark.
27. Specification and Description Language. - CCITT, Recommendation Z.100, 1988.
28. Unified Modelling Language Resource Center. http://www.rational.com/uml/index.jtmpl

APPENDIX 1. BREAL SYNTAX

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. BREAL SEMANTICS

Conditions

Let $CNF1 = (T1, V1, C1, S1 = (ACT1, DEL1))$ and
$CNF2 = (T2, V2, C2, S2 = (ACT2, DEL2))$ be a pair of configurations.

TIME.CONST $T1 = T2$
TIME.STEP $T1 < T2$

VAR.CONST $\forall x.V1(x) = V2(x)$.
VAR.STEP(x) $\forall y \neq x.V1(y) = V2(y)$.
VAR.IO($prog$) $(V1, V2) \in IO(prog)$.

CHAN.CONST $\forall chan.C1(chan) = C2(chan)$.
CHAN.STEP($chan$) $\forall$ $chan' \neq chan$: $C1(chan') = C2(chan')$.
CHAN.HEAD($chan, sig, x$) $GET(C1(chan)) = (C2(chan), sig, V2(x))$.
CHAN.PUT($chan$) $\exists sig, val.PUT(C1(chan), sig, val) = C2(chan)$.
CHAN.GET($chan$) $\exists sig, val.GET(C1(chan)) = (C2(chan), sig, val)$.
CHAN.WR($chan, sig, x$) $PUT(C1(chan), sig, V1(x)) = C2(chan)$.

CHAN.CLEAN($chan$) `EMP`$C2(chan)$.

CHAN.INPUT($chan$) $chan$ is input.

CHAN.OUTPUT($chan$) $chan$ is output.

DEL.CONST $\forall state.DEL1(state) = DEL2(state)$.
DEL.ZER $\forall state.DEL2(state) = 0$.
DEL.OUT $\forall state, transition$ if $state \in ACT1$ and $state$ marks the *transition*, then $DEL1(state)$ exceeds the right bound of the $interval$ of the $transition$.
DEL.NOT-OUT
$\forall state$. if $state \in ACT1$, then there exists a transition marked by $state$ in the process such that $DEL1(state)$ does not exceed the right bound of the transition $interval$.
DEL.IN($state, interval$) $DEL1(state) \in interval$.
DEL.PROGR $\forall state$. if $state \in ACT1$, то $DEL2(state) = DEL1(state) + T2 - T1$, иначе $DEL2(state) = 0$.

ACT.CONST $\forall state.state \in ACT1 \Leftrightarrow state \in ACT2$.
ACT.ACT($state$) $state \in ACT1$.
ACT.NEXT($next$) $next \in ACT2$.
STATE.FIN $\forall state$. if $state \in ACT1$, then in the process there is no transition marked by the $state$.

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

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

CTR(*state, chan, interval, next*)
the process diagram contains a transition *state* `CLEAN` *chan* `JUMP` *Set* where *Set* is a set of states such that $next \in Set$.

PTR($state, prog, interval, next$)

the process diagram contains a transition *state* `EXE` *prog interval* `JUMP` *Set* where *Set* is a set of states such that $next \in Set$.

Rules

RULE 1 (Stuttering)
TIME.CONST, VAR.CONST, CHAN.CONST, ACT.CONST, DEL.PROGR, DEL.NOT-OUT $\models CNF1 <$ INV $> CNF2$
RULE 2 (Stabilization)
TIME.STEP, VAR.CONST, CHAN.CONST, ACT.CONST, DEL.PROGR, STATE.FIN $\models CNF1 <$ INV $> CNF2$
RULE 3 (Reading of a signal)
TIME.CONST, DEL.ZER, $\exists$ *state*, $sig, x, chan$, *interval*, *next*: VAR.STEP(x), CHAN.HEAD($chan, sig, x$), CHAN.STEP($chan$), ACT.ACT($state$), ACT.NEXT($next$), DEL.IN($state, interval$), RTR(*state, sig, x, chan, interval, next*) $\models CNF1 <$ RDN($chan, sig, x$) $> CNF2$
RULE 4 (Writing of a signal)
TIME.CONST, DEL.ZER, VAR.CONST, $\exists$ *state*, *sig*, *x*, *chan*, *interval*, *next* : CHAN.WR(*chan*, *sig*, *x*), CHAN.STEP($chan$), ACT.ACT($state$), ACT.NEXT($next$), DEL.IN($state, interval$), WTR(*state, sig, x, chan, interval, next)* $\models CNF1 <$ WRT($chan, sig, x$) $> CNF2$
RULE 5 (Appearance of a signal)
TIME.CONST, VAR.CONST, ACT.CONST, DEL.CONST, $\exists$ *state*, sig, x, *chan*, *interval*, *next* : CHAN.PUT($chan$), CHAN.STEP($chan$), RTR(*state, sig, x, chan, interval, next*) $\models CNF1 <$ INV $> CNF2$
RULE 6 (Disappearance of a signal)
TIME.CONST, DEL.ZER, VAR.CONST $\exists$ *state, sig, chan, interval, next* : CHAN.GET($chan$), CHAN.STEP($chan$), ACT.ACT($state$), ACT.NEXT($next$), DEL.IN($state, interval$), WTR(*state, sig, x, chan, interval, next*) $\models CNF1 <$ INV $> CNF2$
RULE 7 (Cleaning input channel)
TIME.CONST, DEL.ZER, VAR.CONST, $\exists$ *state, chan, interval, next* : CHAN.INPUT($chan$), CHAN.CLEAN($chan$), CHAN.STEP($chan$), ACT.ACT($state$), ACT.NEXT($next$), DEL.IN($state, interval$), CTR(*state, chan, int erval, next*) $\models CNF1 <$ CLNIN $> chanCNF2$
RULE 8 (Cleaning output channel)
TIME.CONST, DEL.ZER, VAR.CONST, $\exists$ *state, chan, interval, next* : CHAN.OUTPUT($chan$), CHAN.CLEAN($chan$), CHAN.STEP($chan$), ACT.ACT($state$), ACT.NEXT($next$), DEL.IN($state, interval$), CTR(*state,*

chan, interval, next) $\models CNF1 <$ CLNOUT $>$ *chan*$CNF2$
RULE 9 (Program Execution)
TIME.CONST, CHAN.CONST, DEL.ZER, $\exists$ *state, prog, interval, next* : VAR.IO(*prog*), ACT.ACT(*state*), ACT.NEXT(*next*), DEL.IN(*state, interval*), PTR(*state, prog, interval, next*) $\models CNF1 < EXE(prog) > CNF2$
RULE 10 (Clock)
TIME.STEP, VAR.CONST, CHAN.CONST, ACT.CONST, DEL.PROGR, DEL.NOT-OUT $\models CNF1 <$ INV $> CNF2$
RULE 11 (Starvation)
TIME.STEP, VAR.CONST, CHAN.CONST, ACT.CONST, DEL.PROGR, DEL.OUT $\models CNF1 <$ INV $> CNF2$

APPENDIX 3. SPECIFICATION OF THE PROCESSES AND PREDICATES OF THE SYSTEM "PASSENGER AND MACHINE"

Specification of the process slot-machine

```
   start
READ station(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 change(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 finish
```

{**Comment.** In the specifcation, the following construction was used:

```
   check
EXE (sum <= 0)?
JUMP give
   check
EXE (sum > 0)?
JUMP showcount
```

Further, the similar “forks” will be denoted by the macro:

```
   check
IF (sum <= 0)
THEN JUMP give
ELSE JUMP showcount  }
```

Specification of the process “passenger”

```
      passenger :  PROCESS
   OUT CHN buttons FOR station WITH PAR name OF 'a'...'z',
       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 change WITH PAR value OF integer ;
   INP CHN booking FOR ticket WITH PAR name OF 'a'...'z' ;
   PR VAR sum, nominal OF integer ;
   PR VAR decision OF 'a'..'z' ;
   PR VAR gottenstation OF 'a'..'z' ;
   QU VAR station OF 'a'..'z' ;
   ¬AT start ; ¬AT press ; ¬AT continue ;
   ¬AT request ; ¬AT chcoin ; ¬AT drop ;
```

{the diagram of the process passenger in the graphical form is given in Fig. 3, page 25) }

```
   start
EXE decision := station JUMP press
   press
WRITE station(decision) INTO buttons JUMP look
   look
READ light(sum) FROM indicator JUMP continue
   continue
IF (sum <= 0)
THEN JUMP request
ELSE JUMP chcoin
   chcoin
EXE
nominal := 1 ;
LOOP nominal := nominal + 1 POOL ; JUMP drop
   drop
WRITE coin(nominal) INTO slot JUMP look
   request
WRITE request INTO buttons JUMP get
   get
READ ticket(gottenstation) FROM booking JUMP satisfaction
```

Specification of predicates

{specification of the predicate start_of_machine}

```
   start_of_machine :  PRED
   {locator} AT machine.start
   {end of specification of the predicate start_of_machine}

   start_of_passenger :  PRED
   AT passenger.start

   no_commands_on_buttons:  PRED
   {controller of emptiness} buttons IS EMPTY

   no_information_on_indicator:  PRED
```

```
indicator IS EMPTY

no_ticket_in_booking:  PRED
booking IS EMPTY

ticket_in_booking:  PRED
{checker of presence}  ticket IN booking

proper_station:  PRED
{relation}  booking.ticket.name = passenger.station
```

В. А. Непомнящий, Н. В. Шилов, Е. В. Бодин

НОВЫЙ ЯЗЫК BASIC-REAL ДЛЯ СПЕЦИФИКАЦИИ И ВЕРИФИКАЦИИ МОДЕЛЕЙ РАСПРЕДЕЛЕННЫХ СИСТЕМ

Препринт
65

Рукопись поступила в редакцию 12.05.1999

Рецензент Ф. А. Мурзин

Редактор А. А. Шелухина

Подписано в печать 28.08.1999

Формат бумаги 60×84 1/16 Объем 2,3 уч.-изд.л., 2,4 п.л.

Тираж 50 экз.

ЗАО РИЦ "Прайс-курьер" 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6