View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

UDC 623/518.3/517.5

A.A. Letichevsky, J.V. Kapitonova , V.P. Kotlyarov, A.A. Letichevsky Jr., N.S.Nikitchenko, V.A. Volkov, and T.Weigert

INSERTION MODELING IN DISTRIBUTED SYSTEM DESIGN

The paper describes insertion modeling methodology, its implementation and applications. Insertion modeling is a methodology of model driven distributed system design. It is based on the model of interaction of agents and environments [1-2] and use Basic Protocol Specification Language (BPSL) for the representation of requirement specifications of distributed systems [3-6]. The central notion of this language is the notion of basic protocol – a sequencing diagram with pre- and postconditions, logic formulas interpreted by environment description. Semantics of BPSL allows concrete and abstract models on different levels of abstraction. Models defined by Basic Protocol Specifications (BPS) can be used for verification of requirement specifications as well as for generation of test cases for testing products, developed on the basis of BPS.

Insertion modeling is supported by the system VRS (Verification of Requirement Specifications), developed for Motorola by Kiev VRS group in cooperation with Motorola GSG Russia. The system provides static requirement checking on the base of automatic theorem proving, symbolic and deductive model checking, and generation of traces for testing with different coverage criteria. All tools have been developed on a base of formal semantics of BPSL constructed according to insertion modeling methodology.

The VRS has been successfully applied to a number of industrial projects from different domains including Telecommunications, Telematics and real time applications.

Introduction

Insertion modeling is the technology of system design founded on the theory of interaction of agents and environments. This theory has been developed in [1–2]. It is based on process algebra and is intended for the unification of different models of interaction and computation (such as CCS, CSP, π-calculus, mobile ambients etc.). In the last years this approach has been successfully applied to the problems of the verification of requirement specifications [3–5] for distributed concurrent systems from different subject domains including Telecommunications, Telematics, distributed computing and others. These applications are supported by the system VRS developed for Motorola by Kiev VRS group. In combination with TAT system developed in Motorola Software Group Russia it supports also the generation of test cases from requirement specifications.

We use the basic protocol specifications [4–6] to formalize requirement specifications for distributed concurrent systems. Basic protocols are parameterized MSCs (Message Sequence Charts) with pre- and postconditions interpreted on the states of an environment with inserted agents. Semantically basic protocol can be considered as a statement $\forall x(\alpha \to < u > \beta)$ of some kind of dynamic logic. In this statement x is a (typed) list of parameters, α and β are precondition and postcondition, correspondingly, and u is a process defined by the MSC diagram. Preconditions and postconditions are formulas of first order multisorted language called the *Basic Language*. This language is used to describe the properties of the states of a system represented as a composition of environment and agents inserted into this environment. The evolving part of a system is represented by distinguished functional and predicate symbols from the signature of the basic language called the *attributes* of an environment. The process u describes finite behavior of an environment with inserted agents. When parameters of a basic protocol are fixed, then we can speak about instantiated basic protocol.

The purpose of this paper is to represent the formal definitions of the main concepts of the Basic Protocols Specification Language (BPSL), define the notion of implementation of Basic Protocols Specifications (BPS) and to give the high level description of the tools of the VRS system. Each

© A.A. Letichevsky, J.V. Kapitonova, V.P. Kotlyarov, A.A. Letichevsky Jr., N.S.Nikitchenko, V.A. Volkov, and T.Weigert, 2008

BPS consists of two parts: the environment description and the set of basic protocols. Environment description determines the signature of basic language and the restrictions on possible interpretations of this signature (some part of a signature can be interpreted at the very beginning, for example, numerical functions and predicates, or constructors for the states of agents). The signature can also include some constructors for actions of agents inserted into environment. The set of basic protocols defines the requirements to the behavior of a system and implicitly defines the insertion function for the given environment. The requirements informally can be expressed in the following way: *If the precondition of some instantiated protocol is valid and the process of this protocol started then after successful termination of this process the postcondition is valid.*

The semantics of BPS is defined by the variety of possible implementations of BPS that satisfy the informal property above. On abstract level an implementation is represented as an *attributed transition system*, that is a labeled transition system with transitions labeled by actions and states labeled by attribute labels.

We distinguish among *concrete* and *abstract* implementations. A concrete implementation assumes the concrete interpretation of a signature of basic language, and the states of concrete implementations are labeled by the attribute valuations that are the partial mappings from constant attribute expressions (expressions of a type $f(a_1, a_2, ...)$ where *f* is an attribute symbol and $a_1, a_2, ...$ are constant terms of corresponding types) to their values. Each closed (no free variables) formula of basic language has the value on each state of a concrete implementation. Basic protocols can be also considered as formulas, but formulas of dynamic logic that express the main behavioral requirements. These formulas must be valid on any state of a concrete implementation and arbitrary values of parameters. Also we would like to specify concrete implementations for a case of concurrent performance of several basic protocols. To catch the situation the *permutability relation* for attributed actions is added to BPS and is used for the definition of so called partially sequential composition of processes. For empty permutability relation this composition coincides with sequential composition and for the case when all actions are permutable it is a parallel composition.

For abstract implementations we do not use concrete interpretation of a basic language mentioned above. An abstract implementation of BPS is defined as an attributed transition system with validity relation between the attribute labels and the formulas of the basic language. Abstract and concrete implementations are partially ordered by two abstraction relations: direct and inverse. These relations were studied in [6]. They generalize many abstractions referred to in [8-9] and were used for the definition of two abstract implementations of a system of basic protocols that cover concrete implementations. The first one is used for the verification, the second one – for generating of tests.

The main tools of the VRS system are divided into two groups: static tools and dynamic tools. Static tools include checkers for consistency and completeness of preconditions, safety checker, time checker and annotation checker. All these tools are based on deductive system, which contains the universal prover for the first order predicate calculus and special provers for linear numeric arithmetic (over real numbers and integers), enumerated and symbolic data types. Deductive system is capable for extension by integrating new specialized provers and solvers for special theories.

Dynamic tools include concrete and symbolic trace generators (CTG and STG). Both are assigned for simulating the behavior of models of a system defined by BPS by generating their traces in the system state space. For CTG the state space is generated in a traditional way by the valuation of attributes. The generation of traces is controlled by the goal state condition, safety conditions checked along traces, and some other means and heuristics that bound the search space. The states for STG are symbolic. Like as in symbolic model checking they are defined by logic formulas and symbolic computations in combination with deduction are used for computing transitions. The BPS used for concrete trace generation can be used for symbolic trace generation as well.

The paper is structured as follows. First we give the general introduction to insertion modeling based on the model of interaction of agents and environments and describe the environments for MSC. Then the main features of the Basic Protocol Specification Language are described. The semantics of this language for concrete and abstract implementations are defined and used then for the definition of requirements for CTG and its high level description. Then we describe the main algorithms used for the static requirements checking. After the high level description of STG we define the notion of abstraction and prove the theorem about the connection between concrete and abstract implementations. We also describe some tools for tests generation. The last section contains the conclusions and the comparison with related approaches.

Insertion modeling

Insertion modeling is the development and investigation of distributed concurrent systems by means of representing them as a composition of interacting agents and environments. Both agents and environments are attributed transition systems, considered up to bisimilarity, but environments are additionally provided with insertion function used for the composition and characterizing the behavior of environment with inserted agents. Attributed transition systems are labeled transition systems such that besides the labels of transitions called actions, they have states labeled by *attribute labels*. If *s* is a state of a system, then its attributed label will be denoted as **al**(*s*). A transition system can be also *enriched* by distinguishing in its set of states *S* the set of *initial states* $S_0 \subseteq S$ and the set of *terminal states* $S_\Delta \subseteq S$. For attributed transition system we use the following notation. $\alpha : s \xrightarrow{a} \alpha' : s'$ means that there is a transition from the state *s* with attributed label $\alpha \in L$ to the state *s*′ labeled by attributed label $\alpha' \in L$, and this transition is labeled by action $a \in A$. Therefore an enriched attributed system *S* can be considered as a tuple

$$< S, A, L, S_0, S_\Delta, T \subseteq S \times A \times S, \mathrm{al} : S \to L >$$

A pair $< A, L >$ of actions and attributed labels is called the signature of a system *S*. We also distinguish a hidden action τ and hidden attributed label 1. In the difference from other actions and attributed labels these hidden labels are not observable.

Behaviors. Each state of a transition system is characterized up to bisimilarity by its behavior represented as an element of behavior algebra (a special kind of a process algebra). The behavior of a syster⊑ı a given state for the ordinary (labeled, but not attributed) systems is specified as an element of a complete algebra of behaviors *F*(*A*) (with prefixing *a.u*, non-deterministic choice *u*+*v*, constants 0, Δ, $\perp$, the approximation relation , and the lowest upper bounds of directed sets of behaviors) [2]. In the sequel we shall use the term process as a synonym of behavior.

For attributed systems *attributed behaviors* should be considered as invariants of bisimilarity. The algebra $< U, A, L >$ of attributed behaviors is constructed as a three sorted algebra. The main set is a set *U* of attributed behaviors, *A* is a set of actions, *L* is a set of attribute labels. Prefixing and non-deterministic choice are defined as usual (nondeterministic choice is associative, commutative, and idempotent). Besides the usual behavior constants 0 (deadlock), Δ (successful termination) and $\perp$ (undefined behavior), the empty action τ is also introduced with the identity

$$\tau.u = u$$

The operation $(\alpha : u) \in U$ of labeling the behavior $u \in U$ with an attribute label $\alpha \in L$ is added. The empty attribute label 1 is introduced with the identity

$$1 : u = u$$

The approximation is extended to labeled behaviors so that

$$(\alpha : u) < (\beta : v) \Leftrightarrow \alpha = \beta \wedge u < v .$$

Constructing a complete algebra *F*(*A*,*L*) of labeled behaviors is similar to the constructing the algebra *F*(*A*). Each behavior *u* in this algebra has a canonical form:

$$u = \sum_{i \in I} \alpha_i : u_i + \sum_{j \in J} a_j . u_j + \varepsilon_u ,$$

where $\alpha_i \neq 1, a_j \neq \tau$, ε_u is a termination constant ($0, \Delta, \perp, \Delta + \perp$), all summands are different and behaviors u_i and u_j are in the same canonical form.

Behaviors, i.e., elements of the algebra *F*(*A*,*L*) can be considered as the states of an attributed transition system. The transition relation of this system is defined as follows:

$$\begin{array}{l} a.u + v \xrightarrow{a} u \\ \alpha : u + v = 1 : (\alpha : u + v) \xrightarrow{\tau} \alpha : u \\ \alpha : a.u \xrightarrow{a} u \\ \alpha : \beta : u \xrightarrow{\tau} \beta : u \end{array} .$$

A set *E* of behaviors is called *transition closed* if $u \in E, u \xrightarrow{a} u' \Rightarrow u' \in E$.

Ordinary labeled transition systems are considered as a special case of attributed ones with the set of attribute labels equal to {1}, and the algebra *F*(*A*) is identified with *F*(*A*,{1}).

Insertion function. Environment $< E, C, L, A, M, \varphi >$ is defined as a transition closed set of behaviors $E \subseteq F(C, L)$ with insertion function $\varphi : E \times F(A, M) \to E$. The only requirement for insertion function is that it must be continuous w.r.t. approximation relations defined on *E* and *F*(*A*,*M*). Usually the behaviors of environment are represented by the states of a transition system considering them up to bisimilarity. The state $\varphi(e, u)$ of an environment resulting after agent insertion (identified with the corresponding behavior) is denoted as $e[u]$ or $e_\varphi[u]$ to mention insertion function explicitly, and the iteration of insertion function as $e[u_1, u_2, ..., u_m] = (...(e[u_1])[u_2]...)[u_m]$. Environments can be considered as agents and therefore can be inserted into a higher level environments with another insertion functions, so the state of multilevel environments can be described for example by the following expression: $e_\varphi[e_\psi^1[u_1^1, u_1^2, ...], e_\psi^2[u_2^1, u_2^2, ...], ...]$. The most of insertion functions considered in this paper are one-step or head insertion functions. Typical rules for the definition of insertion function are the following (one-step insertion):

$$\frac{e \xrightarrow{a} e', u \xrightarrow{a} u'}{e[u] \xrightarrow{c} e'[u']} , \tag{1}$$

$$\frac{e \xrightarrow{c} e'}{e[u] \xrightarrow{c} e'[u]} . \tag{2}$$

The first rule can be treated as follows. Agent *u* ask for permission to perform an action *a*, and if there exist an *a*-transition from the state *e* the performance of *a* is allowed and both agent and environment come to the next state with observable action *c* of environment. The second rule describes the move of environment with suspended move of an agent. The additivity conditions usually are used:

$$\begin{array}{c} e[u + v] = e[u] + e[v] , \\ (e + f)[u] = e[u] + f[u] . \end{array}$$

The rules (1–2) can be also written in the form of rewriting rules:

$$\begin{array}{c} (a.e')[a.u'] = c.e'[u'] + f , \\ (c.e')[u] = c.e'[u] + g . \end{array}$$

MSC environment. The standard semantics of MSC diagrams is defined as process algebra semantics [7–8], so that the meaning of MSC is the set of traces of a process over the set of events used in MSC. The process algebra proposed by M.A.Reniers for this purpose was very complicated and far from implementations. An alternative approach has been developed in [9–10] where insertion semantics of MSC has been defined. The main difference from the Reniers semantics is that we consider branching time instead of linear and synchronizing treatment of conditions and references. We shall use this semantics as intermediate semantics for basic protocols so let us consider it here.

We use in basic protocols only simple MSC, that is MSC with the following constructs: instances, conditions, messages, coregions, and local actions. Before inserting MSC *B* to the MSC-environment it is converted to the process $proc(B) = p_1 \| \ldots \| p_n$ where $p_1, \ldots, p_n$ are sequential compositions of events corresponding to the instances of a diagram (parallel composition must be used for coregions) in the same order as on instances. Events are considered as actions and these actions are message actions, local actions, instance actions and local conditions used instead of ordinary conditions sharing several instances. Parallel composition is defined by means of interleaving. We use the notations for events as in [9] (fig.1).

send message	(*i*: *m* to *j*)
receive message	(*i*: *m* from *j*)
local action	(*i*: action *b*)
instance start	(*i*: instance)
instance stop	(*i*: stop)
condition	(*J*: condition *y*)
local condition	(*i*: cond *y*(*J*))

Fig. 1

The process over the actions listed in this figure is called MSC-process and MSC-process without local conditions is called reduced MSC process.

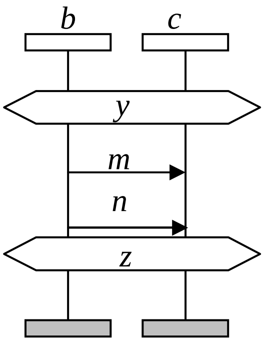


Fig.2

For example, if *B* is a diagram of fig.2, then *proc*(*B*)=((*b*:instance).(*b*:cond *y*{*b*,*c*}).(*b*:*m* to *c*). (*b*:*n* to *c*).(*b*:cond *z*{*b*,*c*}).(*b*:stop) || (*c*:instance).(*c*:cond *y*{*b*,*c*}).(*c*:*m* from *b*). (*c*:*n* from *b*).(*c*:cond *z*{*b*,*c*}).(*c*:stop)).

The state of MSC-environment is a triple of functions *e* = (O, Σ, Y) (empty environment, no inserted agents) or the expression $e[u_1, u_2, \ldots, u_m]$, where *e* is an empty environment and agents $u_1, u_2, \ldots, u_m$ are MSC-processes. The states of MSC environment are unlabeled. The function O is a partial function of three arguments *m*, *i*, *j*, where *m* is a message expression, and *i* and *j* are instances. This function yields values in the set of non-negative integers. Equation O(*m*, *i*, *j*) = *k* means that earlier *k* message events (*i* : *m* **to** *j*) occurred for which there are no corresponding receiving message events pending. Function Σ is a partial function of two arguments *y* and *J*. The first argument is a condition expression, the second is a set of instances. The value Σ(*y*, *J*) represents a nonempty subset of the set *J*. Σ(*y*, *J*) = *I* means that earlier a control event *i* : **cond** *y*(*J*) had been executed, for all instances $i \in J$. The condition event is attached to all instances in *J*, and Σ(*y*, *J*) is the set of all instances which have already been synchronized by the condition *y*. The last component Y of a triple is a set of all instances that had already started. This explanation results in the following definition of environment transitions. For empty environment *e* a transition

$$e \xrightarrow{a} e'$$

is possible in the following cases:

1. *a* = (*i*: *m* to *j*),
   $i \in \mathrm{Y}, e'.\mathrm{O}(m,i,j) = e.\mathrm{O}(m,i,j) + 1$
2. *a* = (*i*: *m* from *j*),
   $i \in \mathrm{Y}, e'.\mathrm{O}(m,i,j) = e.\mathrm{O}(m,i,j) - 1$
3. *a* = (*i*: action *b*), $e' = e$
4. *a* = (*i*: instance), $e'.\mathrm{Y} = e.\mathrm{Y} \cup \{i\}$
5. *a* = (*i*: stop), $e'.\mathrm{Y} = e.\mathrm{Y} \setminus \{i\}$

6. $a = (i\text{: cond } y(J))$,
$J \subseteq \Upsilon, e'.\Sigma(i:\text{cond } y, J) = \Phi(J, e.\Sigma(i:\text{cond } y, J), i)$

where

$$\Phi(J, \perp, i) = \{i\}$$
$$\Phi(J, J \setminus \{i\}, i) = \perp$$
$$\Phi(J, I, i) = I \cup \{i\}$$

and a rule is applied if previous one is not applied. In all these rules only the components of environment state that change their value are shown. All other components of environment left unchanged. Now we define the rule (1) for empty MSC-environment so that in the cases 1-3 $c = a$, in cases 4-5 $c = \tau$, and in the case 6, $c = (J\text{: condition } y)$ if $e'.\Sigma(i:\text{cond } y, J) = \perp$ and $c = \tau$ otherwise.

Initial state of empty MSC environment is $e_0 = (\perp, \perp, \varnothing)$ where the first two components are nowhere defined functions and the last component is the empty set of instances. For the process *B* on fig. 2 the representation of the process $e_0[B]$ is given on fig. 3.

To define insertion function for nonempty environment let us define some auxiliary notions. MSC-process is called belonging to instance *i* if all its actions belong to the instance *i*. MSC-process is called *decomposable* if it can be represented as a parallel composition of processes belonging to different instances. The set of all decomposable processes is transition closed (follows directly from the definition).

The process *proc*(*B*) is equivalent to the instance oriented textual representation of MSCs and we can define the composition *u***v* of decomposable MSC-processes which in the case of MSCs is equivalent to their vertical product. This composition is defined by the following identities:

$$(p_1 \parallel \ldots \parallel p_m \parallel Q) * (q_1 \parallel \ldots \parallel q_m \parallel S) =$$
$$= ((p_1; q_1) \parallel \ldots \parallel (p_m; q_m) \parallel Q \parallel S),$$

where $p_1, \ldots, p_m$ belong to the same instances as $q_1, \ldots, q_m$, correspondingly, *Q* and *S* have no common instances. Note that joining two instances we can use the identity ((*i*: stop).(*i*: instance)) = Δ, because these two behaviors are insertion equivalent.

For nonempty environment *e*[*u*] and decomposable processes *u* and *v* define $e[u, v] = e[u * v]$. This definition can be extended to arbitrary number of decomposable processes:

```
e0[B] = (
            ((b:instance). (c:instance)+(c:instance). (b:instance)); {b,c}:condition y);
            (b:m to c). e1 [
                                (b:n to c). (b:cond z{b,c}).(b:stop) ||
                      (c:m from b). (c:n from b).(c:cond z{b,c}).(c:stop)
            ]
      ) = (
            ((b:instance). (c:instance)+(c:instance). (b:instance));
            ({b,c}:condition y). (b:m to c).(
                      ((b:n to c). (c:m from b)+(c:m from b). (b:n to c));
                      (c:n from b).({b,c}:condition z);
                      ((b:stop). (c:stop) +(c:stop).(b:stop));
                      e0[Δ]
            )
      )
```

Fig. 3

$$e[u_1,...,u_m]=e[u_1 * ... * u_m].$$

To use this definition for arbitrary (not only decomposable) MSC-processes one must extend the notion of the composition of MSC-processes.

Partially sequential composition of behaviors. Let us consider attributed behaviors of the algebra *F*(*A*,*L*). Let $L:A=\{\alpha:a\,|\,\alpha\in L, a\in A\}$ be the set of attributed actions (unlabeled actions are 1:*a*, attribute labels are $\alpha:\tau$). First we define the permutability relation over the set *L*:*A*. This is an arbitrary symmetric and reflexive binary relation denoted as $a\leftrightarrow b$. Intuitively this relation means that $(a*b)\sim_E(b*a)$ for environment *E* where the agents from *F*(*A*,*L*) will be inserted into. We say that an attributed action $\alpha:a$ is reachable from behavior *u* if there exists behavior *v* such that $\alpha:v$ is reachable from *u* and $\alpha:v\xrightarrow{a}v'$. Let us expand the permutability relation of attributed actions to attributed behaviors. We say that behaviors *u* and *v* are permutable ($u\leftrightarrow v$) if 0 and $\perp$ are not reachable from *u* and *v*, and for each attributed action *a* reachable from *u* and attributed action *b* reachable from *v* $a\leftrightarrow b$.

Now we can define the partially sequential composition $u*v$ of two behaviors. Let *u* and *v* are two attributed behaviors, represented in the canonical form:

$$u=\sum_{i\in I}\alpha_i:u_i+\sum_{j\in J}a_j.u_j+\varepsilon_u, v=$$

$$=\sum_{k\in K}\beta_k:v_k+\sum_{l\in L}b_l.v_l+\varepsilon_v$$

Then

$$u*v=\sum_{i\in I}\alpha_i:(u_i*v)+\sum_{j\in J}a_j.(u_i*v)+$$

$$+\sum_{u\leftrightarrow\beta_k,k\in K}\beta_k:(u*v_k)+\sum_{u\leftrightarrow b_l,l\in L}b_l.(u*v_l)+(\varepsilon_u;\varepsilon_v)$$

A sequential composition of termination constants is defined with the following relations:

$$(\Delta;\varepsilon)=\varepsilon,\ (\perp;\varepsilon)=\perp,\ (0;\varepsilon)=0.$$

Note that partially sequential composition is not continuous with respect to the first argument; however it is continuous with respect to the second one. It is also continuous with respect to both arguments, if the first argument is finite and totally defined. Let now *u* and *v* be completely unlabelled. Then if all actions are permutable, then a partially sequential composition coincides with a parallel composition, and if no actions are permutable, then we obtain a sequential composition of behaviors. The notion of partially sequential composition originates from the notion of weak sequential composition introduced by Renier for describing the semantics of MSC diagrams and is a generalization of the latter (for not delayed nondeterministic choice).

Permutability for MSC-environment. The states of MSC have no attribute labels (the empty label is omitted). Two actions are permutable if they belong to different instances, or in the case of conditions (sharing a set of instances) have no common instances. Now we can define partially sequential composition of arbitrary (not necessarily decomposable) MSC-processes and define for them insertion function as above but changing vertical product to partially sequential composition that coincides with vertical product for decomposable processes.

For MSCs *B* and *C* we have *proc*(*B*)**proc*(*C*)=*proc*(*B***C*) where the product of MSCs is their vertical product. If e_0 is the initial state of empty MSC-environment and *B* is an MSC diagram then [*B*] denotes the behavior of a system $e_0[proc(B)]$.

Interpreted MSC. MSC environment regulates the correct ordering of MSC-actions in MSC-processes. The actions of interpreted MSCs can also define the data transformation in higher level data environment *D*. Transitions $d\xrightarrow{a}d'$ in *D* for MSC action *a* define the corresponding transformation. The insertion function for data environment can be defined in the same way as for MSC environment using partially sequential composition, but with another notion of permutability. Below several kinds of data environments will be considered. For MSC diagram *B* we can construct MSC-process [*B*] and insert this process into data environment obtaining two level environment *d*[[*B*]]. As it was mentioned before *d*[[*B*],[*C*]]= *d*[[*B*]*[*C*]] = *d*[[*B***C*]].

Basic protocol specifications

Let us start with a very simple and well known example: readers and writers

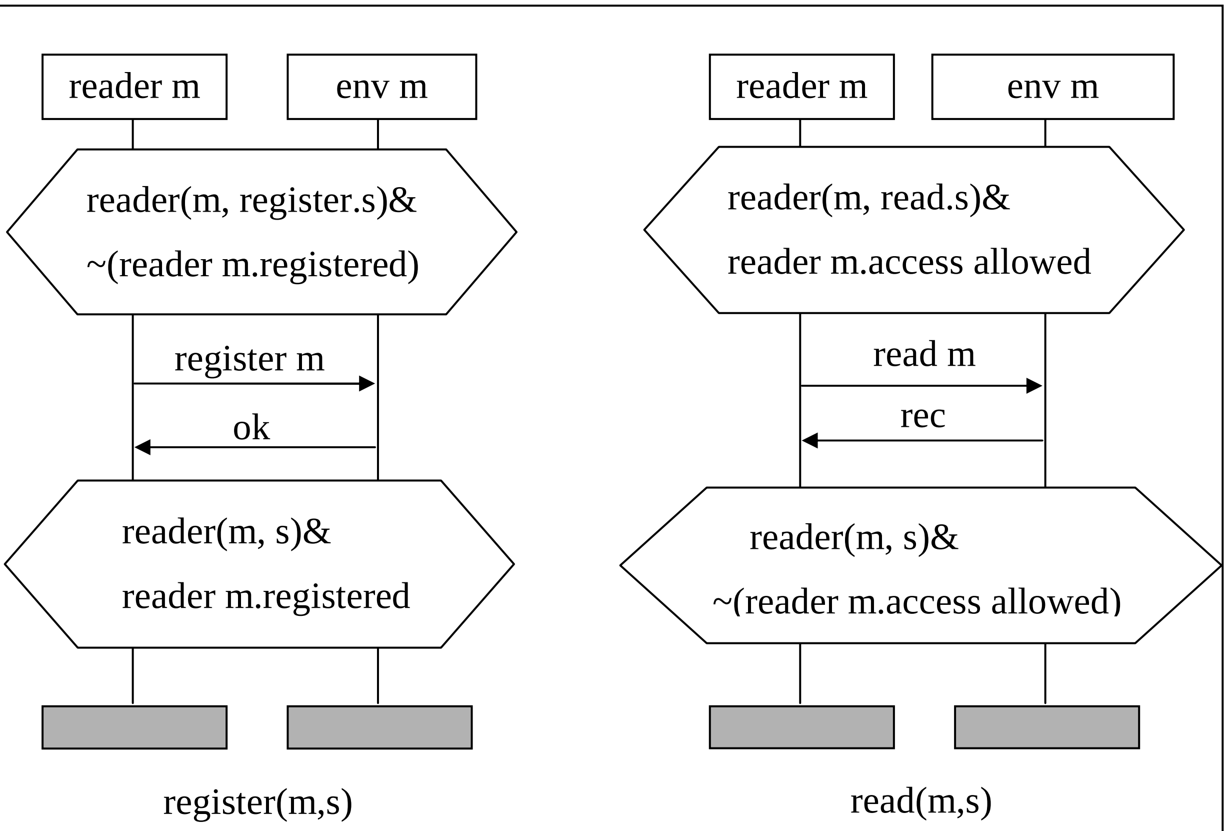
(R&W). The environment keeps shared record that can be sent to readers and updated by writers. Readers and writers are two types of agents that can be inserted into this environment. Basic Protocol Specification (BPS) of a system consists of two parts: environment description and the set of basic protocols. Environment description has a text representation and basic protocols are MSC diagrams with pre- and postconditions. The description of R&W environment is presented in fig.4. It shows that there are two environment attributes **rec** of symbolic type and **queue**, the list of symbolic type data. Each reader has two Boolean attributes **registered** and **access allowed**. The agents of a type writer have no attributes in the environment. Safety condition asserts that access (to the **rec**) is allowed only for registered readers. This is a dynamic requirement which must be satisfied in any environment state. The last assumption states that initially there are no registered readers (and therefore access is not allowed for all of them). Other sections of environment description will be discussed later.

The set of basic protocols represents the set of local requirements to the system. Each protocol is a simple MSC with a special text block with parameters. Only two conditions are allowed in Basic Protocol. The first one is at the beginning of MSC. It is shared by all instances of the protocol and contains a precondition formula as a condition text. The second condition is at the end of a protocol, it is shared by all instances and contains a postcondition as a text. Text block contains a list of typed parameters used for instantiation of a protocol.

Basic protocols for R&W are represented in figures 5-7. Each diagram is accompanied by its name and a list of parameters. First three protocols describe the local behavior of the system inspired by reader activity. The protocol write(m,s,x) describes the activity of the writer, and the last protocol update(x,t) describes the transition of the environment (without agents participation). The first four protocols have an expression of a type $\tau(m,s)$ where τ is a type of agent and s is the state expression. It is a *state assumption* and means that the agent m of the type τ is in a state s. Agents are represented by their unique names (ids), states are represented by means of behavior (process) algebra expressions.

```
environment(
        agent types:(
                reader: (
                        registered: Bool,
                        access allowed: Bool
                ),
                writer: Nil
        );
        attributes:(
                rec: symb,
                queue: list of symb
        );
        safety condition: Forall(m:reader)(
                m.access allowed->m.registered
        );
        initial condition:(
                Forall(m:reader)(
                        ~(reader m.registered)&
                        ~(reader m.access allowed)
                )
        )
);
```

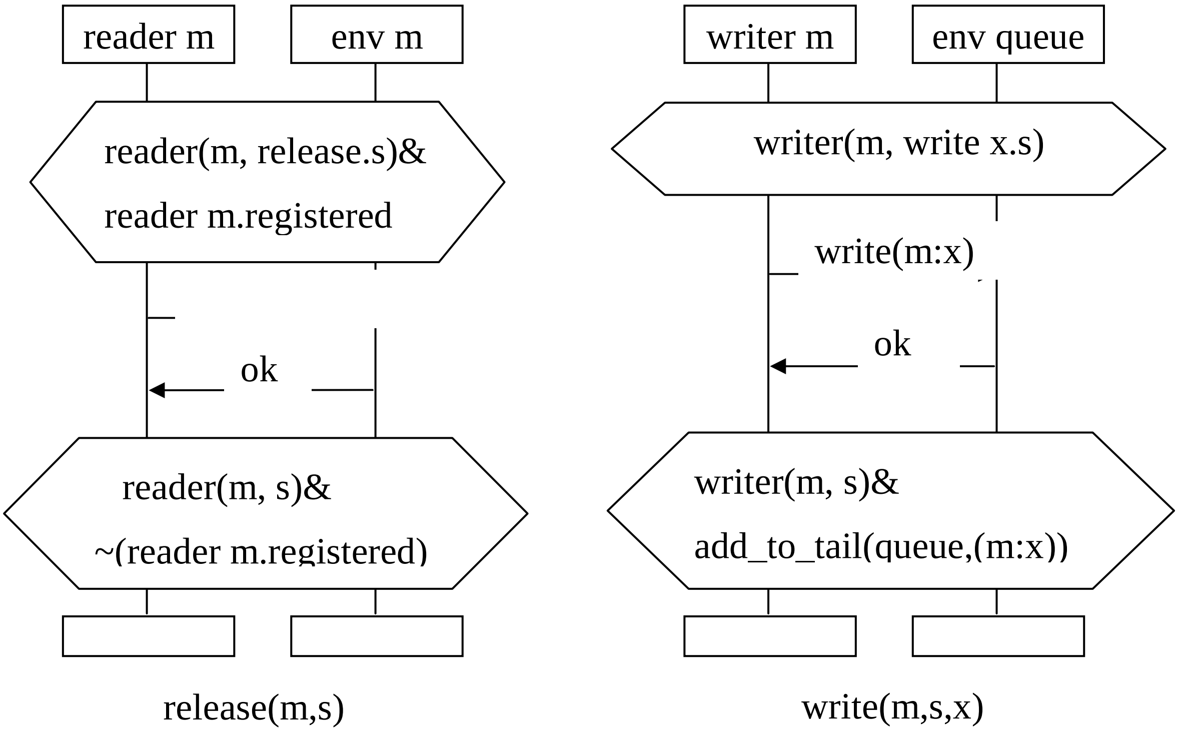
Fig. 4



register(m,s)

read(m,s)

Fig. 5



release(m,s)

write(m,s,x)

Fig. 6

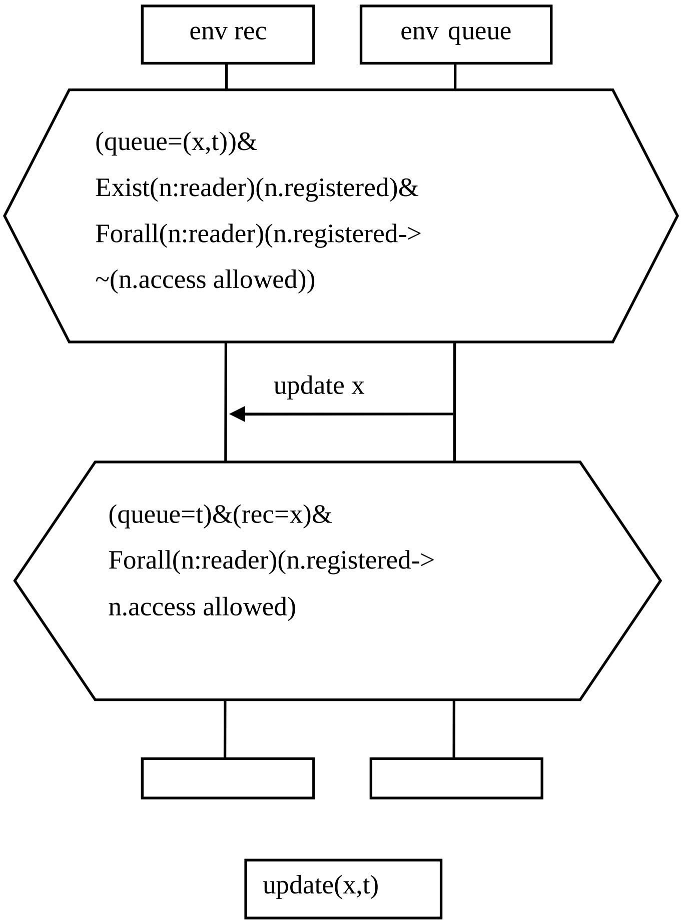


Fig. 7

Therefore the symbols **register**, **read**, and **release** are reader actions, and **write** is a writer action. The actions of environment are send and receive message events or local actions presented by MSC. Agent attributes are accessed by the expressions of the type $\tau\, m.x$, where τ is the type of agent m and x is the name of attribute (type of agent can be omitted in some cases). The instances are named by expressions containing the names of agents or the name of environment **env** possibly attached by the list of attributes or the names of agents.

In symbolic notation a basic protocol will be represented by expressions of the form

$$\forall x(\alpha \to < u > \beta)$$

where x is a list of parameters, α and β are precondition and postcondition, correspondingly, and u is an MSC process (usually inserted to MSC environment).

Basic language used for pre- and postconditions is the first order language of multi-sorted predicate calculus with equality. The signature of this language has predefined part and the part defined by an environment description. Predefined part contains simple numeric types **int** and **real,** with arithmetical operations and inequalities, Boolean type **Bool**, type **agent state** with operations of process algebra (prefixing and nondeterministic choice), symbolic type **symb** with some predefined constructors (such as (():()), ((),()) etc.), and composite types: **arrays** and **lists** of simple type elements and functional types $(\tau_1,...,\tau_m) \to \tau$ where $\tau_1,...,\tau_m$ are already defined types and τ is a simple type. However there are some restrictions on the use of functional types for different tools. For example, concrete trace generator allows only enumerated types for $\tau_1,...,\tau_m$.

Predefined part of basic language signature contains also the agent state function symbol $state_\tau$ for each agent type τ defined in environment description. This is a unary symbol for the function of a type $\text{agent name} \to \text{agent state}$. These symbols are used implicitly in state assumption expressions $\tau(m,s)$ equivalent to formulas $state_\tau(m) = s$. Some constant symbols for agent states can also be included into the signature of basic language an can be defined by behavior algebra equations. These equations can be represented in **reductions** section of environment description. For example, possible equations for the behaviors of readers and writers in R&W example are:

state reduction: rs(x)(
reader init = register.(loop read;release),
writer init = (loop write;release),
loop x = (x;(loop x+Delta))
).

This is a correct behavior of readers and writers. More free behavior including incorrect activity can be described by another reductions:

incorrect state reduction: rs()(
reader init = register state + read state +
+ release state,
register state = register. reader init,
read state = read. reader init,
release state = release. reader init,
writer init = write state + writer release,
write state = write. writer init,
writer release = release. writer init
).

In environment description one can define several enumerated types in the section **types** (for example **types**: (colour: (red, green, yellow), size: (small, big),…)). One can also define agent types with associated agent attributes that are considered as functional symbols. Functional and array attributes have arity more then 0, attributes of simple types and lists (simple attributes) are considered as functional symbols of arity 0. They are equivalent to the variables that change their values when a system moves from one state to another (but they cannot be bounded by quantifiers). Other attributes can also change their values during the evolution of a system, but this changing may refer only to the change of the parts of a value.

Universal and existential quantifiers with variables typed by simple types can be used in formulas. All terms in formulas must have simple types.

The difference of precondition and postcondition is that in postcondition one can use the imperative elements like assignments. In this case assignment (*x*:=*y*) is equivalent to the statement: new value of attribute *x* (after performing a basic protocol) is equal to the previous value of the expression y (before performing of a protocol). In protocol write(m,s,x) the operator add_to_tail (queue,(m:x)) is used. This operator adds new element to a list and is interpreted as statements so that new value of list queue is the result of performing of the operator add_to_tail applied to the previous value of the second parameter.

The signature of basic language may include also abstract data types defined in the sections **types**, **functions**, **axioms**, and **reductions**. Section **types** contains symbols for simple abstract data types, section functions contains specification of functional symbols of type $(\tau_1,...,\tau_m) \to \tau$, axioms are first order formulas without attributes, and reductions are systems of rewriting rules that must define canonical forms.

This information is used by deductive system, which contains a universal procedure for the first order theory, and procedures for eliminating quantifiers and solving equations in special theories (numeric, symbolic and enumerated) and some distinguished data types theories.

Semantics of BPS

In [4] semantics of basic protocols has been presented in a very abstract and general form. In [6] there were introduced two kinds of semantics, direct and inverse, and a theorem about connection between abstract and concrete implementations of a system of basic protocols was formulated with a sketch of proof. Here we shall consider more concrete definitions close to processes represented by MSCs and the implementation in VRS system. Moreover we use here the independent notion of implementation of BPS and the constructions of [4] and [6] can be now proved to be implementations.

As in [4] and [6] we associate a transition systems with each BPS as its possible semantic values and distinguish between concrete and abstract implementations. In concrete implementations all functional and predicate symbols are interpreted on corresponding domains and the state of environment is the partial valuation defined on all attributes. In abstract implementations the class of possible interpretations for the signature is fixed, and the states of environment are formulas of basic language.

Implementation of BPS. First we should answer the question: what is the implementation of a basic protocol specification. Then the class of all possible implementations will be declared as a semantic value of BPS. It must be an attributed transition system with validity relation $s \models \alpha$ where *s* is a state and α is a closed statement (no free variables) of basic language. The transitions of this system must be labeled by environment actions that are the events of basic protocols considered as MSCs.

The validity relation |= must satisfy the following condition. Let **T** be the set of formulas of basic language that includes all axioms of the environment description (including reductions) and all true statements of the specialized theories for interpreted part of a signature. Then the set **C**(*s*) of all statements α valid in *s*, that is statements for which $s \models \alpha$ is valid, must be closed relative to the inference in the theory **T**, that is for any statement $\alpha \in \mathbf{C}(s)$ all statements β such that $\mathbf{T}, \alpha \mid - \beta$ are also in **C**(*s*). The set **C**(*s*) relates to the observable part of a system and must depend only on the attribute label of a state *s*. In abstract form of a system model the attribute label can be identified with this set, but in practice more constructive representations are used.

Note that basic language includes all information about the types and independent behaviors of agents to be inserted into their environment. But they interact with environment only via basic protocols (state assumptions) and can be present implicitly. Some information about the variety of agents can be represented by means of validity relation which can include more statements except of deduced in the theory **T**.

If a system *S* is the implementation of BPS it must satisfy the following requirement. Let *B* is an instantiated basic protocol (a protocol with substituted parameters) with precondition α and postcondition β. Let $s \in S$. Then

$$s \models \alpha, s \xrightarrow{[B]} s' \Rightarrow s' \models \beta.$$

This requirement is not enough because it does not take into account the possibility of concurrent performing of several protocols. To make this possible we use corresponding permutability relation and partially sequential composition. The modified requirement can now be expressed in the following form:

$$s \models \alpha, s \xrightarrow{[B]*[C]} s' \Rightarrow s' \models \beta \quad (3)$$

for arbitrary MSC *C* such that $[B] \leftrightarrow [C]$. Therefore to define the notion of implementation we must define the permutability relation for basic protocol specifications.

A system *S* can be considered as a data environment for reduced MSC processes defined by equation (1), additivity conditions, and equation

$$s[u,v] = s[u*v].$$

Let us consider some assumptions about interpretation of MSC actions on *S*, that is a transition relation for *S*. We assume that message and local actions as well as instance actions do not change the set **C**(*s*). It means that if $s \xrightarrow{a} s'$ and *a* is not a condition, then **C**(*s*) = **C**(*s*′). We introduce two kinds of conditions: condition **when** and condition **set** to distinguish the performance of pre- and postconditions. Condition *a* = (*J*: **condition when** *y*) is a guard, it does not change the attribute labels of the states of *S*, but allows the transition $s \xrightarrow{a} s'$ only if *s*|=*y*. Condition **set** changes not only the state of *S* but also its validity relation. If *a* = (*J*: :**condition set** *y*) and $s \xrightarrow{a} s'$ then $s' \models y$. Assume that in basic protocols preconditions are changed to **when** condition and postconditions are changed to **set** condition. The set of MSC actions modified in this way will be called the modified MSC action set. A BPS is called *consistent* if a theory **T** defined by environment description is consistent and the conjunction of all initial conditions and safety conditions is satisfiable.

Summarizing discussion above we give the following definition of implementation of BPS. An attributed transition system *S* over the set of modified MSC actions is called an *implementation* of a given consistent BPS if

1. Validity relation |= depending only on attribute labels and closed over the theory **T**, defined by environment description, is defined on *S*.
2. A permutability relation is defined on the set of modified attributed MSC actions so that condition (3) held for all basic protocols.

Trivial implementation consisting with only one deadlock state always exists. So usually we are interested in the existence of nontrivial implementations, for example free of dead locks or satisfying safety (integrity) conditions or satisfying other kinds of dynamic properties.

Direct and inverse implementations. The difference between two kinds of implementations is in the applicability condition used in the formula (3). In this formula the applicability condition is $s \models \alpha$ for instantiated precondition α. This is a direct implementation. For inverse implementation a dual condition is used: $\neg(s \models \neg\alpha)$. This condition means that precondition α is consistent with current state and is weaker than direct condition. For concrete implementations these conditions coincide (consistency and completeness of concrete implementation), so this is interesting first of all for abstract implementations. Considered form of representation of inverse applicability condition is difficult for computations so simpler conditions are used in practice. For example, let attributed labels are formulas of basic language. Then $s \models \alpha \Leftrightarrow \mathbf{T}, \mathbf{al}(s) \models \alpha$. Let $\mathbf{al}(s) = \gamma(r_1, r_2, ...), \alpha = \alpha(r_1, r_2, ...)$, where $r_1, r_2, ...$ attributes occurring in two formulas and they are simple attributes. Then sufficient condition for inverse applicability can be expressed in the form

$$s \models \exists (x_1, x_2, ...)(\gamma(x_1, x_2, ...) \wedge \alpha(x_1, x_2, ...)).$$

Concrete and abstract implementations

Concrete implementations. Concrete implementations of BPS are characterized first of all that a concrete interpretation of all symbols from the signature of the base language are defined for them, except for attributes. For them only the value domains and the ranges of the arguments are defined. Another characteristic of concrete implementations is an explicit representation of the system in the form of a composition of the environment and agents inserted into it. Finally, a concrete implementation should be deterministic with respect to the agents' behavior. In other words, an arbitrary change of the state of the agents and their attributes should lead to a uniquely determined change of other attributes of the environment. Here a certain class of concrete implementations will be described.

BPS assumed to be consistent for a finite number of agents. That is the set of axioms is consistent and there exists an initial state of environment with finite number of agents satisfying initial and safety conditions. State *s* of empty environment is the partial valuation of the set of all attributes. To achieve the determinism some additional hidden attributes can be added to the attributes described in environment description together with rules or algorithms for their change. Let **A** be the set of all attributes (including hidden ones), that is a set of all constant attribute expressions $f(a_1, a_2, ...)$ where *f* is an attribute symbol and $a_1, a_2, ...$ are constant terms of corresponding types. If *f* is a symbol of agent attribute it has the form (*m.x*) where *m* is the name of agent and *x* is the name of attribute. In this case we write $m.x(a_1, a_2, ...)$. We add an undefined value $\perp$ to all domains and consider the state *s* as a totally defined mapping $s : \mathbf{A} \to \mathbf{D}$ where **D** is the union of all domains extended by undefined value. Let $\mathbf{A}_0$ be the set of all attributes defined by environment description (they are observable). Define an attribute label for *s* as a narrowing of *s* to $\mathbf{A}_0$, that is $\mathbf{al}(s) : \mathbf{A}_0 \to \mathbf{D}, \mathbf{al}(s)(x) = s(x), x \in \mathbf{A}_0$. Let $\mathbf{Eq}(s)$ is the conjunction of all equalities $x = s(x), x \in \mathbf{A}_0$. Define validity relation so that $s \models \alpha \Leftrightarrow \mathbf{T}, \mathbf{Eq}(s) \models \alpha$. In the most of applications when quantifiers are restricted by finite sets of values (agent names or enumerated types) and *s* is defined only on the finite set of attributes the validity of the formulas of basic language is computed without any deduction simply by substituting the values.

The next step of constructing the concrete implementation is the definition of transitions for the empty environment. Define $s \xrightarrow{a} s'$ so that

1. If a is not a condition, then $\mathbf{al}(s) = \mathbf{al}(s')$;
2. If a = (J: **condition when** y), $\mathbf{al}(s) = \mathbf{al}(s')$ and $s \models y$;
3. If a = (J: **condition set** y), then $s' \models y$ and only those attributes which are the left hand sides of assignments or occur in logic formulas can change their values in the new state.

Now define the states of entire system as

$$s[q_1 * \ldots * q_m][v_1 : u_1, \ldots, v_n : u_n]$$

where $q = q_1 * \ldots * q_m$ is the partially sequential composition of MSCs with modified action sets and inserted into MSC environment, $u_1, \ldots, u_n$ are the states of agents over the action set A, $v_1, \ldots, v_n$ are their names (all different) considered as the attribute labels of agents. Transitions of named agents are defined as follows:

$$u \xrightarrow{a} v \Rightarrow m : u \xrightarrow{a} m : v$$

Initial state

$$s[\Delta][\mu_1 : w_1, \ldots, \mu_k : w_k]$$

is such that all initial and safety conditions are valid on s. So the set S of empty environment states is considered as an environment for interpreted MSC agents and the set $s[q]$ is considered as an environment for agents over A with insertion function defined below. We call this environment as the main one. We assume that the set of named agents that can be inserted into the main environment is fixed and constitutes only the agents in the initial state. This assumption will be used each time the protocol is instantiated or the value of quantifier formula with variables bounded to the agent states or names is computed.

The permutability relation for MSC agents is defined in the following way.

1. Unconditional actions are permutable if they belong to different instances;
2. Conditional actions a and b are permutable if their instance sets do not intersect and for all states s if $s \xrightarrow{a.b} s'$ and $s \xrightarrow{b.a} s''$ then $\mathbf{al}(s') = \mathbf{al}(s'')$.

From this definition it follows that for two MSCs A and B such that $[A] \leftrightarrow [B]$ from $s \xrightarrow{[A]*[B]} s'$ and $s \xrightarrow{[B]*[A]} s''$ it follows that $\mathbf{al}(s') = \mathbf{al}(s'')$.

The insertion function for the entire system is assumed to be additive and satisfy the commutativity condition: $s[q][u,v] = s[q][v,u]$. Also we assume that $s[q] \xrightarrow{a} s'[q'] \Rightarrow$ $s[q][u] \xrightarrow{a} s'[q'][u]$ and $s[q][m:\Delta, v] = s[q][v]$. We say that an agent m participate in q if a formula $\tau(m,u)$ occur in q. We also assume that the type of an agent is uniquely defined by its name m. Now we can define the insertion function by means of the following transition rules.

Changing the state of a protocol:
where action a is supposed to be unconditional action.

$$\frac{s \xrightarrow{a} s', q \xrightarrow{a} q'}{s[q] \xrightarrow{a} s'[q']}, \qquad (4)$$

Activating a protocol b:

$$\frac{s \models \alpha, \mathbf{[}b\mathbf{]} = a.p, q \leftrightarrow a, s \xrightarrow{a} s'}{s[q][m_1 : u_1, \ldots, m_k : u_k] \xrightarrow{a} s'[q * p]}, \qquad (5)$$

where $a = (J : \text{condition when } \alpha)$, $m_1 : u_1, \ldots, m_k : u_k$ are agents involved into the protocol b (their state assumptions occur in the condition α). The protocol b is assumed to be instantiated by constant values of parameters.

Terminating a protocol:

$$\frac{s \xrightarrow{a} s', q \xrightarrow{a} q'}{s[q] \xrightarrow{a} s'[q'][m_1 : u_1, \ldots, m_k : u_k]}. \qquad (6)$$

Here q is a partially sequential composition of MSC agents, a is a condition **set** action (therefore a modified postcondition), $m_1, \ldots, m_k$ are the names of agents which participate in q, but do not participate in q', $u_1, \ldots, u_k$ are their states. The new states of agents are defined syntactically by state assumptions from postcondition.

The described implementation is direct one. The inverse implementation can work when some attributes needed for computing validity relation have no values. This is normal for abstract implementations used for proving properties but is not normal for concrete implementations for which the undefined value usually is the indication of error.

Denote the class of concrete implementations of BPS *P* described in this section as Concr(*P*)

Theorem 1. Each transition system *S* from the class Concr(*P*) is a direct and inverse implementation of *P*.

The first property of implementation follows from the definition of validity relation for *S*, the second property follows from the definition of permutability relation. For concrete implementation we have a concrete interpretation of the theory T. Therefore for each state *s* each formula α is valid ($s \models \alpha$) or not valid ($s \models \neg\alpha$). Therefore $s \models \alpha \Leftrightarrow \neg(s \models \neg\alpha)$ and the each direct implementation from Concr(*P*) is at the same time the inverse implementation and wise versa.

Abstract implementations. For abstract implementations the interpretation of the signature of a basic language is not strictly defined and the validity relation is derived from attributed labels of states and a theory **T** defined by environment description (including predefined interpretation). We assume that the attributed labels of the states $s \in S$ of empty environment are formulas of the basic language and the validity set $\mathbf{C}(s) = \{\alpha \,|\, \mathbf{T}, \mathbf{al}(s) \,|{-}\, \alpha\}$. The environment for abstract implementation is constructed in the same way as a concrete two level environment. The low level is the environment for basic protocols and the high level is the level for named agents over the action set *A*. So the general state of a system has the form

$$s[q_1 * ... * q_m][v_1 : u_1, ..., v_n : u_n],$$

but in the difference from concrete implementations the agents occurring in the state do not constitute a complete set of agents in the system. New agents can appear and disappear during the performance of environment and their states can be defined by means of symbolic expressions.

Instantiation of basic protocols in the case of concrete and abstract implementations are different, however in the both cases the solution of the precondition must be found, that is the list of values of parameters such that after substitution the precondition is valid. In the case of concrete implementation the solution must be concrete, and in the case of abstract implementation the solution can use symbolic values in the extended signature of basic language. For example, let precondition be the formula $x \le a$ where *x* is an integer parameter and *a* is a simple integer attribute. If the value of *a* is defined then concrete implementation can select arbitrary integer less or equal to *a* for *x* (which one depends on hidden computations with hidden attributes. Otherwise the protocol is not applicable. In the case of direct abstract implementation a new symbolic value *z* for *x* can be introduced as a new constant symbol and a condition $(z \le a)$ will be added to the attribute label of a current state if $a \ne \perp$ is valid. The permutability relation is defined in the same way as for concrete interpretation.

The insertion function for abstract implementations is defined by means of the same rules (4–6) as for concrete implementations with slight change and another restrictions on the constituents of these rules. The rule (4) is used without any changes because the unconditional actions never change the attributed labels. The rule (5) for activating protocol is slightly changed. To explain this change let us consider the symbolic representation of a protocol *b*:

$$\forall x(\alpha(x) \to < u(x) > \beta(x))$$

Applicability of this protocol in the current state $\gamma : s$ is equivalent to the condition $\mathbf{T}, \gamma \models \exists x \alpha(x)$. If this condition is valid, we can found a general solution $x_1 = t_1(z), x_2 = t_2(z), ...$ for the problem $\mathbf{T}, \gamma \models \alpha(x)$, where $z = (z_1, z_2, ...)$ is the list of symbolic parameters needed to express the general solution. Their symbols must be different from all other symbols already used in BPS and have corresponding types. In the worst case if there are no good solution procedure, the list *x* can be renamed to new parameters list *z* and $\alpha(z)$ added to the attributed label. Let

$a(x) = (J : \text{condition when}\, \alpha(x)), t = (t_1, t_2, ...)$.

Then the modified rule (5) can be represented as follows:

$$\frac{(\gamma : s) \models \exists x \alpha(x), [b] = a(x).p(x), q \leftrightarrow a(t), s \xrightarrow{a(t)} (\gamma \wedge \alpha(t)) : s'}{s[q][m_1 : u_1,...,m_k : u_k] \xrightarrow{a(t)} ((\gamma \wedge \alpha(t)) : s')[q * p(t)]} \tag{5a}$$

The process of instantiation of a basic protocol which could be hidden in the case of concrete basic protocols because there were no symbolic parameters, must be defined explicitly together with the rule of activation of a protocol. Another peculiarity of symbolic case is that not all agents participating in the residual part *p*(*t*) of a protocol are taken from the list of inserted in the current moment agents but some can be generated in the moment of instantiation. For example, some of parameters from the list *z* can be the names of new agents, others can be parameters of their state expressions in the state assumptions. All this depends on the states of a system and algorithms used for the implementation. One more peculiarity of symbolic implementations is that it can implement inverse implementation model. In this case the activation rule must looks as follows.

$$\frac{\neg((\gamma : s) \models \neg\exists x \alpha(x)), [b] = a(x).p(x), q \leftrightarrow a(t), s \xrightarrow{a(t)} (\gamma \wedge \alpha(t)) : s'}{s[q][m_1 : u_1,...,m_k : u_k] \xrightarrow{a(t)} ((\gamma \wedge \alpha(t)) : s')[q * p(t)]} \tag{5b}$$

The termination rule is the same as for concrete implementation, but let us present it with the explicit change of attributed labels:

$$\frac{(\gamma : s) \xrightarrow{a} (\gamma' : s'), q \xrightarrow{a} q'}{s[q] \xrightarrow{a} s'[q'][m_1 : u_1,...,m_k : u_k]} \tag{6a}$$

Here $a = (J : \text{condition set}\, \beta)$, and the attributed label γ' must satisfy the condition common for all condition **set** actions: the postcondition must be valid on $\gamma' : s'$ that is $\mathbf{T} | - \gamma' \to \beta$. The function $\gamma' = \mathbf{pt}_s(\gamma, \beta)$ is called a *predicate transformer* in the state *s*.

As for concrete implementation $m_1,...,m_k$ are the names of agents which participate in *q*, but do not participate in q', $u_1,...,u_k$ are their states. Again the new states of agents are defined syntactically by state assumptions from postcondition, but the list of agents can be not complete, some agents can disappear or even all of them disappear (*k* = 0).

Predicate transformers. Two classes of implementations are described in a very general way. They contains a big undefined components that depend on subject domain, the not formalized requirements, and concrete circumstances of the development process. A source BPS model can be very concrete and remains no choices for implementation details or too abstract and provides a big freedom in making decisions. The definition of termination rule has the most indefinite component. Any condition γ' consistent with the theory **T** and strengthening β can be chosen for attributed label of new state of environment. We can make the choice of predicate transformer more definite if some additional requirements are added to it. For example, we can claim that in arbitrary concrete implementation covered by the given abstract implementation the values of attributes can change only if they occur in postcondition. In this case if $\gamma = \gamma_1 \wedge \delta_1 \vee \gamma_2 \wedge \delta_2 \vee ...$ and $\delta_1, \delta_2,...$ have no occurrences of the attributes occurring in β, we can set $\mathbf{pt}(\gamma, \beta) = (\delta_1 \vee \delta_2 \vee ...) \wedge \beta$.

Denote the class of direct (inverse) abstract implementations of BPS *P* described in this section as Adir(*P*) (Ainv(*P*)).

Theorem 2. Each system from the class Adir(*P*) (Ainv(*P*)) is a direct (inverse) implementation of *P*.

The validity relation is closed by definition. The second property of implementation follows from the definition of permutability.

Abstractions. As it is clear from above that the implementations can have different levels of *abstraction*. This notion can be defined exactly in terms of attributed transition systems with validity relation considered up to bisimilarity. The definition below is close to the definition of [6] but does not use the labeled actions. First we must introduce some simplification of transition systems connected with elimination of hidden transitions, that is transitions labeled by τ.

Hidden transitions of the form $1{:}\,s\xrightarrow{\tau}1{:}\,s'$ are called *redundant*. All redundant transitions may be eliminated in the following way: first, add transitions according to the rules:

$$\frac{1{:}\,s\xrightarrow{\tau}1{:}\,s',s'\xrightarrow{a}s''}{s\xrightarrow{a}s''},$$

$$\frac{1{:}\,s\xrightarrow{\tau}1{:}\,s',s'\xrightarrow{\tau}\alpha:s''}{s\xrightarrow{\tau}\alpha:s''},$$

where $\alpha\neq 1$, until it is possible (an infinite number of steps may be required, at each step all rules are applied to all states simultaneously), then all redundant transitions are eliminated. After such transformation at least one state in each hidden transition will be labeled by attributed label different of 1. The system with no redundant transitions is called reduced. Below only reduced systems will be considered.

Let S and S' be two attributed (not necessarily different) systems with the same signatures, validity relations and the same basic language denoted as **BL**. Define the abstraction relation $\mathbf{Abs}\subseteq S\times S'$ in such a way that

$$(s,s')\in\mathbf{Abs}\Leftrightarrow\forall(\alpha\in\mathbf{BL})((s\models\alpha)\Rightarrow(s'\models\alpha)).$$

To denote abstraction relations (the state s is an *abstraction* of the state s') the notation $s\lhd s'$ will be used instead of $(s,s')\in\mathbf{Abs}$. If $s\lhd s'$ and $s'\lhd s$, then the states s and s' are called deductively equivalent.

The system S is called a *direct abstraction* (or a direct abstract model) of the system S' and the system S' is called a *concretization* of the system S if there exists a relation $\varphi\subseteq\mathbf{Abs}^{-1}$, which is a relation of modeling (simulation). It means, that the following statement holds:

$$\forall(s\in S,s'\in S')((s',s)\in\varphi\wedge s\xrightarrow{a}t\Rightarrow$$
$$\Rightarrow\exists(t'\in S')(s'\xrightarrow{a}t'\wedge(t',t)\in\varphi)).$$

The system S is called an *inverse abstraction* (or an inverse abstract model) of the system S' and the system S' is called an *inverse concretization* of the system S if there exists a relation $\varphi\subseteq\mathbf{Abs}^{-1}$, which is a relation of inverse modeling. It means, that the following statement holds:

$$\forall(s\in S,s'\in S')((s',s)\in\varphi\wedge s'\xrightarrow{a}t'\Rightarrow$$
$$\Rightarrow\exists(t\in S)(s\xrightarrow{a}t\wedge(t',t)\in\varphi)).$$

For enriched systems the requirement of preserving the initial and terminal states is added in both cases: if $s'\in S'_0(S'_\Delta)$ and $(s',s)\in\varphi$, then $s\in S_0(S_\Delta)$.

The system and its abstraction are related in the following way. If from an abstraction of some initial system state a certain property is reachable in its direct abstraction, then it is also reachable from the initial system state. For inverse abstraction the inverse property hold: if from initial state of a system some property is reachable then it is reachable from some initial state of its inverse abstraction.

Let us consider as a famous example a Kripke system S'; i.e., an attributed system with binary attributes $x_1,...,x_n$. The states of this system are labelled with Boolean functions of the variables $x_1,...,x_n$. Let $f(x_1,...,x_n)$ be one of such functions. Let us decompose it with respect to the first k variables; i.e., present it in the form:

$$f(x_1,...,x_n)=$$
$$=\bigvee x_1^{\alpha_1}...x_k^{\alpha_k}f(\alpha_1,...,\alpha_k,x_{k+1},...,x_n),$$

where the disjunction is over all binary vectors $(\alpha_1,...,\alpha_k)$ such that $f(\alpha_1,...,\alpha_k,x_{k+1},...,x_n)$ is different from 0. Let us define the transformation $P_{x_1,...,x_k}(f(x_1,...,x_n))=g(x_1,...,x_k)$ of the function $f(x_1,...,x_n)$, by:

$$g(x_1,...,x_k)=\bigvee x_1^{\alpha_1}...x_k^{\alpha_k}.$$

Let us modify the system S' by substituting the function **al** for its labeling function **al**′ with $\mathrm{al}(s)=P_{x_1,...,x_k}(\mathrm{al}'(s))$. Denoting thus modified system as S, we shall receive and abstract model of the system S'. Similar models are used in model checking to reduce the state space generated for checking.

The importance of the notion of abstraction is explained by the following theorem.

Theorem 3. Each abstract implementation of the class Adir(*P*) (Ainv(*P*)) is an abstraction of some concrete implementation from the class Concr(*P*). This follows from the well known logic fact that each consistent axiom system has a consistent complete interpretation.

Theorem 4. There exist a direct and an inverse abstract implementations which are direct and inverse abstractions of all implementations from Concr(*P*).

Corresponding constructions were defined in the paper [6]. Complete proof need some more mathematics and we are going to this in the next papers.

Concrete trace generator

Purpose. CTG module of VRS system provide checking the reachability of properties, detecting deadlocks, non-determinisms, safety violations, unreachable requirements, usage of uninitialized attributes and admitted range attribute overflow on a base of a concrete model of formal requirement specification in the form of BPS. The problems above are solved by means of generating traces reachable from the initial state of a model. The generated traces can be also used for test generation. Results are represented in .MSC/PR and .txt formats.

In general, the CTG module input is the set of the following files:

- Environment description;
- Filters description;
- Events description;
- Set of Basic Protocols.

Environment. Environment description is restricted to the following elements: list of enumerated types in type description, list of environment attributes, list of agent types, list of instances, initial values of listed above elements and agent states. Abstract syntax of environment description is as follows:

```
<environment descr>::= environment(
        types: <enumerated types declaration>;
        attributes: <list of attr descr>;
        agent_types: <list of agent descr>;
        agents:  <list of typed agent ids>;
        instances: <list of instances>;

        initial: env(
          (
                attributes: <list of environment attributes values>;
                agent_parameters: <list of agent attributes values>
          ),
          <agent states>
        )
).
```

Enumerated type declaration defines the symbol of enumerated type and the set of its values. Attribute description defines the symbol of environment attribute and its type. Agent type description defines the name of an agent type and the list of agent attributes for this type together with the types of these attributes. The names of agents inserted into environment are represented in the list of typed agents ids. The list of instances represents the set of instances which can be used in MSC diagrams representing the interaction activity of agents. There is no explicit correspondence between agents and instances. Usually each instance corresponds to one agent, but there can be another kinds of correspondence (one instance – several agents or one agent – several instances). Some of the instances can be used to represent the activity of environment. This correspondence does not explicitly expressed in the model and is in the mind of a user.

Every environment/agent attribute has one of the following types: integer, real, enumerated, symbolic type, list, array, functional type (parameterized attributes) with only enumerated parameters and values.

Environment description represents the structure of environment and contains some information about agents inserted into this environment. It also defines the initial state of environment. In this definition each agent and environment attribute can be assigned a value. Default is the undefined value. The initial states of all agents are represented in <agent states>.

Filters. Filters are used for the restriction of the set of traces generated by CTG. The following filters are available in **CTG:**

goal state,
restricted state ,
break state,
safety conditions.

All filters are defined by formulas checked during the trace generation.

Goal state defines the successful termination of generating procedure. It is used to check the reachability of goal state condition or generating testing sequences for given coverage criteria.

Restricted states cut the set of states used for the search of goal states. When the restricted state is reached the system returns back to the nearest choice point.

Break states switch the trace generator to interactive mode. In this mode the direction of further development of traces is selected by user. After several steps of interactive move a user can return to automatic generation of traces.

Safety conditions must be true at any states so it is checked each time the system comes into the new state. If some of the safety conditions is violated the generation is stopped and the corresponding trace is fixed.

There are also some other restrictions like the maximal trace length or switching the control of visited states.

There are also some predefined conditions such as deadlock condition, non-determinisms, unreachable requirements, usage of uninitialized attributes and admitted range attribute overflow which are checked and result terminating the generation of a trace when these conditions are valid.

Events description contains the information about messages used in MSC diagrams: message names, parameters of messages, parameter values.

Basic protocols. To be efficient the CTG allows a very restricted class of basic protocols. Each protocol has the following abstract syntax form in symbolic notation:

$$\mathrm{Forall}(x,y)(L(x,y)\,\&\,u(x)\,\&\,F(x,y)\rightarrow$$

$$\rightarrow< B(x,y) > v(x,y)\,\&\,A(x,y))$$

In this formula $x=(x_1,x_2,...)$ is a list of parameters called key agent parameters. They are used in the conjunction of state assumptions $u(x)$. The first state assumption in $u(x)$ is called a key agent assumption and has the form $\tau(m,s(z_1,z_2,...))$ where m is the name of a key agent and can be one of the key agent parameters, s is the (constant) name of an agent state, and $z_1,z_2,...$ are all other key agent parameters. All other agent state assumptions do not contain parameters and are constructed with symbolic constants, agent parameters, and attributes. The list $y=(y_1,y_2,...)$ contains data parameters that obtain their values from the definitions $L(x,y)=(y_1=t_1\,\&\,y_2=t_2\,\&...)$, the right hand sides of these definitions can depend on key agent parameters, but not on data ones. The formula $F(x,y)$ is the formula of basic language. It can contain quantifiers but only for finite sets represented by enumerated data types. The expression $v(x,y)$ is a conjunction of agent state assumptions for agents introduced by precondition and $A(x,y)$ is the conjunction of imperative statements: assignments, list processing statements (add_to_head, add_to_tail etc.). $B(x,y)$ is MSC representation of this basic protocol.

The restrictions on basic protocols structure show that the performance of basic protocols can be done by computation on the current state of a system under modeling. The state expression in key agent assumption is matched with the states of all agents to get the values of parameters x and to decide which of the agents can be used as a key agent in the performance of the protocol. Data parameters obtain their values after the choice of a key agent and are used for instantiating the protocol by means of substitution.

The main part of precondition is a logic formula $F(x,y)$ which is computed on the current state if it is possible. The failure can appear if the values of some of the attributes needed for computation are undefined. So if precondition is valid the protocol can be applied by performing the imperative statements of postcondition in parallel (first compute the right hand sides of assignments then assign). The state assumptions in postcondition give the new states of agents participating in the protocol. Expressions in pre- and postconditions use various functions defined on data types. Some other imperative constructions in addition to assignments are admissible in the postconditions.

Generating traces. Ordinary trace for a transition system is a sequence

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} ... \xrightarrow{a_{n-1}} s_n$$

of conjugated transitions. If restrict only to observable part of a trace it is a sequence $a_1, a_2, ..., a_{n-1}$, and for attributed systems an observable trace is

$$\mathrm{al}(s_1) \xrightarrow{a_1} \mathrm{al}(s_2) \xrightarrow{a_2} ... \xrightarrow{a_{n-1}} \mathrm{al}(s_n).$$

The set of traces (ordinary or observable) for the behavior [*B*] of an MSC *B* can be very huge, but it is well represented by the MSC itself. So for the user representation it is not necessary to come from MSC to its traces. CTG generates traces by sequential activating of basic protocols, so it obtains first the set of so called BP traces, that is a sequences

$$s_1 \xrightarrow{B_1} s_2 \xrightarrow{B_2} ... \xrightarrow{B_{n-1}} s_n$$

where $B_1, B_2, ..., B_{n-1}$ are instantiated basic protocols, and $s_1, s_2, ..., s_n$ are intermediate states. According to the semantics of concrete implementation the resulting set of traces is the set of traces of the system $s_1[[B_1]*[B_2]*...*[B_{n-1}]]$ where * means the partially sequential composition of interpreted MSCs. We can take for observation partially sequential composition $[B_1]*[B_2]*...*[B_{n-1}]$ of uninterpreted MSCs, that can be well represented by vertical composition $[B_1 * B_2 * ... * B_{n-1}]$ of MSCs, but it may contain more observable traces than needed. Indeed, two conditional actions can be permutable as uninterpreted actions because they have no common instances, but they are not permutable because they have common attributes. To avoid the necessity of considering all observable traces CTG assumes the following *correctness condition* for MSCs: *if two condition actions are not permutable as interpreted actions, they must have at least one common instances*.

As a consequence, the CTG looks through basic protocols entirely, but not actions and checks the permutability of basic protocols using this easy criteria – no common instances. If two protocols are not permutable, CTG consider both cases in their order, otherwise only one order will be considered.

So the CTG generates the tree of BP traces and after performance of the next basic protocol it checks the filters. If one of the filters triggered the generating of the current BP-trace is interrupted, this trace is written if it is necessary to the set of generated traces, and CTG returns to the nearest choice point to continue the generation of the next trace.

Output. The set of generated BP-traces with the explanations of reasons of terminating. Each trace is contained in the separate verdict file. CTG also produces statistic of the generation process.

Static requirements checking

Static requirement checking (SRC) tools allow to solve verification problems without generating traces and exploring the state space. For this purpose SRC uses deductive system. The system is based on the universal prover for the first order predicate calculi with equality extended by some special provers and solvers. Universal prover is based on Glushkov evidence algorithm [11], insertion representation of this algorithm is presented in [2]. The specialized part of deductive system supports proving and solving linear numerical constraints for integers (Presburger algorithm) and for reals (Furiet-Motskin algorithm), proving and solving formulas for enumerated and symbolic data types. So the environment description for SRC is much richer than for CTG. Especially the use of axioms and rewriting rules are allowed. Of course the use of arbitrary axiomatic or equational theories can lead to insolvability or inefficiency, so some preliminary adjustment may be needed. Another requirement for usage of special theories is the possibility of separation of subformulas belonging to different theories if there are no procedures for their combination.

The input of deductive system is a closed formula (no free variables) of basic language with all needed axioms and reductions which can be invoked preliminarily. Deductive system gives one of four possible answers: proved, not proved, refuted and unknown. If the statement is proved, the proof can be printed by request. The answer not proved means that the statement cannot be proved. If the statement is refuted, the refutation can be printed by request, and the answer unknown means that the process failed for lack of resources or for lack of knowledge (on the combination of theories or equational problems etc.).

The statement to be proved can contain the occurrences of attributes. Such a statement must be checked for arbitrary values of attributes. In this case all attributes are substituted by variables which are bind by universal quantifiers. For functional attributes which occur with different arguments the substitution must be more complicated then in the simple case.

The query for solving has the form $\text{solve}(x_1, x_2,...)P(x_1, x_2,...)$. It is assumed that the formula $\exists(x_1, x_2,...)P(x_1, x_2,...)$ has been already proved. If the formula *P* has quantifiers they are eliminated (only theories which allow quantifier elimination are considered) and then the formula *P* is simplified to obtain the explicit solution $x_1 = t_1, x_2 = t_2,...$. If the explicit solution is not possible, than the simplified formula is used as an implicit solution. Generally a mixed solution can be obtained in the form of conjunction of the explicit part (conjunction of solved equalities) and the implicit part (just a formula).

The problems that can be solved by SRC are:

Transition consistency of preconditions;

Completeness of preconditions;

Proving safety conditions.

Transition consistency. Let $\forall x(\alpha \rightarrow< u > \beta)$ and $\forall x'(\alpha' \rightarrow< u' > \beta')$ are two basic protocols. Their preconditions are called to be consistent if they cannot be valid at the same time. Formally it can be written by the consistency formula $\forall x \forall x' \neg(\alpha \wedge \alpha')$ or $\neg\exists x \exists x'(\alpha \wedge \alpha')$ (the lists x and x' must have no common variables otherwise use renaming). Consistency means determinism, only one of two protocols can be applied at the same time. If the consistency formula cannot be proved or is refuted, we have indeterminism that can be undesirable and it must be announced to user who can answer if the discovered indeterminism is desired or not.

Completeness. The applicability condition for a protocol $\forall x(\alpha \rightarrow< u > \beta)$ is the validity of the formula $\exists x \alpha$. The set of basic protocols is called complete if the disjunction of all applicability conditions for them is always valid. So completeness means that in any situation there must be at least one applicable protocol.

Both properties are only partially recognizable because their violation is only necessary condition for inconsistency or incompleteness. Indeed, sometimes inconsistency can be proved but the state that satisfies the inconsistency may be unreachable. Therefore we can set the inconsistency condition $\exists x \exists x'(\alpha \wedge \alpha')$ as a goal state and prove its reachability by means of CTG or STG.

Proving safety. Safety condition is an invariant property for the system and in some cases can be proved inductively. That is it must be valid on any initial state and if it is valid on some reachable state, then it must be valid on the state obtained after the application of a basic protocol. If it is difficult to characterize the set of reachable states by logic formula the tool consider arbitrary state such that safety condition γ is valid and for arbitrary basic protocol applicable to this state applies this protocol formulates and calls deductive system to prove the corresponding statement. For the protocol $\forall x(\alpha \rightarrow< u > \beta)$ and safety condition γ the corresponding formula is

$$\forall x(\text{pt}(\gamma \wedge \alpha, \beta) \rightarrow \gamma)$$

where **pt** is the predicate transformer described at the end of the fifth section.

Symbolic trace generator

All input files for symbolic trace generator (STG) are the same as for CTG. The difference is that an abstract model of a system specified by a set of basic protocols is considered instead of a concrete one. The state of a model has the form

$$\gamma[u_1,...,u_n]$$

where γ is a formula over attributes which coincides with the attributed label of the environment state and $u_1,...,u_n$ are agents inserted into environment. The deductive system is used for checking the applicability of basic protocol and predicate transformer described in the section 5 is used to obtain the next state.

There are several modes of generating traces controlled by the following properties;

1. Complete or incomplete set of agents. Complete set of agents means that there are no other agents except of those existing in the initial state. Incomplete agent set allows introducing and inserting new agents form the virtual high level environment in the time of generating traces.

2. Direct or inverse implementation. Defines which formula is used for checking applicability of basic protocol.

Basic protocols. Each protocol has the following abstract syntax form in symbolic notation:

$$\text{Forall}(x)(u(x)\,\&\,F(x)\rightarrow$$
$$\rightarrow< B(x)>v(x)\,\&\,A(x)\,\&\,w(x))$$

The list of parameters is not separated to agent and data parameters. The conjunction of state assumptions *u*(*x*) is the same as in CTG. As for CTG the first state assumption in *u*(*x*) is a key agent assumption and has the form $\tau(m,s(z_1,z_2,...))$ where *m* is the name of a key agent and can be one of parameters, *s* is the (constant) name of an agent state, and $z_1,z_2,...$ are parameters or constants. The formula *F*(*x*) is the formula of basic language without any restrictions. The expression *v*(*x*) is a conjunction of agent state assumptions for agents introduced by precondition, *A*(*x*) is the conjunction of imperative statements, *w*(*x*) is a postcondition formula (no restrictions). *B*(*x*,*y*) is MSC representation of this basic protocol.

The application of a protocol to the current state of a system is performed according to the rules (5a) in the case of direct implementation or (5b) in the case of inverse one with the following refinement for STG. The state expression in key agent assumption is matched with the states of all agents to get the values of those parameters which occur in the first state assumption and to decide which of the agents can be used as a key agent in the performance of the protocol. If a key agent cannot be found in the set of agents inserted into environment then in the case of complete agent set the checking is failed and in the case of incomplete agent set a new agent can be generated and inserted into environment with the state matched with key agent state assumption. After making decision about a key agent, and checking state assumptions for other agents some of parameters obtain values. Moreover some of attributes can obtain the values because for not key agents the unification, not matching is applied considering the attributes having no definite value as variables. After that the parameters which obtained values are deleted from the list of parameters and their values are substituted to the protocol. Now the main applicability condition is checked:

$$\mathbf{T}\models\forall z(\gamma(z)\rightarrow\exists xR(x,z))$$

for direct implementation and

$$\mathbf{T}\models\exists z(\gamma(z)\wedge\exists xR(x,z))$$

for inverse one. Here $\gamma(r)$ is an environment state formula, $R(x,r)$ is the main part of precondition with the explicit dependence on attributes and $r=(r_1,r_2,...)$ is a list of attributes occurring in formulas. These formulas are valid for simple attributes and at this time only this simple case is implemented in CTG. Now the protocol must be instantiated for the remained part of parameters. It is realized as was described in the section 5 by solving the problem $\text{solve}(x)(\gamma(r)\rightarrow R(x,r))$ for direct implementation or the problem $\text{solve}(x)(\gamma(r)\wedge R(x,r))$ for inverse one. STG generates BP-protocols as CTG, Therefore the termination of a protocol must be done just after checking its applicability. This is performing according to (6a) using the same predicate transformer as in SRC.

Conclusions and related approaches

In this paper we have described insertion modeling methodology, its foundations, implementation and applications. Insertion modeling was presented as a technology of model driven distributed system design. This means that the following development scheme is adopted. First, requirements specifications are defined, then an executable model is constructed, and at last an implementation is built based on results of requirement and model verification.

Table

Project	Reqs & related docs in pages	Number of BPs in formaliza tion	Coverag e of original reqs	Defects found	Generated traces with counter-examples	Effort in staff-weeks
Telecommunication 1	400	127	50 %	11	0	5.5
Telematics 1	200	70	100 %	10	3	5.6
Telecommunication 2	730	192	100 %	18	7	20
Telecommunication 3	~1500	56	50 %	8	5	5.5
Telematics 2	323	219	60 %	38	8	3
Telematics 3	116	42	100 %	3	1	0.7
Telematics 4	~1500	3005	100 %	129	7	22,5
Telecommunication 4	~2000	2311	100 %	223	17	21,3

This scheme was successfully piloted in a number of industrial projects of Motorola. Main domains of application are telecommunication and telematics. The table below gives a short statistics about piloting. Project size is characterized by number of pages in initial requirements documentation and numbers of Basic Protocols in corresponding formalized specification. Fourth column demonstrates what part of initial documentation was formalized. column contains total number of defects found with all VRS tools and methods but sixth gives results of traces generators that could be used in further testing. The last column presents summary of engineering efforts required for piloting.

In the proposed insertion modeling approach an attributed transition system is considered as a system model, requirement specifications are defined as basic protocol specifications. Process part of such specifications is presented as an MSC-diagram. For verification of specifications algorithmic (model checking) and deductive (proofs) methods are used.

The proposed system development and verification schemes are quite common for a number of other approaches.

Traditional mathematical models for specifications of concurrent systems usually are based on process algebras (CSP, CCS, Lotos, ACP, μCRL, π-calculus, etc.), temporal and dynamic logics (LPTL, LTL, CTL, CTL*, PDL), and automata models (timed Büchi and Muller automata, abstract state machines).

Classical process algebras (CSP [12], CCS [13]) are very abstract. The states of processes are terms; transitions specify transformations of such terms. Further development of these process algebras (ACP, μCRL, π-calculus) extends classical models with data, named channels and new process compositions. The theory of interaction of agents and environments goes further in concretization of process models. First, it distinguish environment as a special agent and introduces new composition (insertion function) of agents into environment. It also introduces data of various types and develops a special action language to present processes in abstract or concrete forms.

The main emphasis is paid to Basic Protocol Specifications, which relate insertion modeling with temporal logics.

Temporal logic is a formal specification language for the description of behavioral properties of non-terminating and interacting (reactive) systems. Among such properties traditionally are distinguished *safety* (“something bad never happens”), *liveness* (“something good will eventually happen”), and various *fairness* properties (a property holds infinitely often under certain conditions).

For example, Lamport’s TLA (Temporal Logic of Actions) [14, 15], is oriented on description of such properties and is based on Pnueli’s temporal logic [16] with assignment, enriched signature, and module specifications. It supports types (strings, numbers, sets, records, tuples, functions) and syntactic constructs (IF, CASE, LET, etc.) taken from programming languages to ease maintenance of large-sized specifications. *Behaviors* are considered as sequences of *states*. States themselves are assignments of values to variables. A system satisfies a formula iff that formula is true for all behaviors of this system. Transition relation is presented by formulae where the arguments are only the old and new states. Such formulae present *actions*.

Many temporal logics are decidable and corresponding decision procedures exist for linear and branching time logics [17], propositional modal logic [18], and some variants of CTL^* [19]. In such decision procedures techniques from automata theory, semantic tableaux, or binary decision diagrams (BDD) [20] may be used.

Typically, a system to be verified is modeled as a (finite) state transition graph, and the properties are formulated in an appropriate temporal logic. An efficient search procedure is then used to determine whether the state transition graph satisfies the temporal formulae or not. This technique was first developed in the 1980's by Clarke and Emerson [21], and by Quielle and Sifakis [22] and extended later by Burch *et.al.* [23].

In the current version of VRS system temporal formulas are presented implicitly being realized with checking algorithms. Still, future versions of the system are planned to be extended with explicit presentation of such formulas.

Among different verification tools we distinguishes here the SCR toolset (Software Cost Reduction) [24] and Action Language Verifier (ALV) [25] which are rather close to the VRS system.

The SCR toolset aims to verify the requirement specifications of applied systems. SCR is based on a user-friendly table notation of finite state machines. There are tables for the description of types, constants, variables. Also initial restrictions, transitions and behavioral invariants of a specified system are presented with tables. System variables are divided into input (or monitored), output (or controlled), and internal variables. Static analysis permits to check the system model for presence of non-determined transitions and non-specified states. Dynamic (behavioral) analysis is oriented on checking safety, fault tolerance, and different temporal properties. All these properties are interpreted as a safety property for a specific formula class. Model checker is used to find counter-example traces. The following verification tools are used: TAME, specialized interface to PVS prover, and SALSA, which is a specialized solver. This solver is oriented at propositional (quantifier-free) formulas with linear equalities and inequalities.

The SCR toolset have much in common with VRS. Terminology is a little different: for model description VRS uses basic protocols and scenarios, and SCR uses tabular notations, in VRS: environment and tabular description, in SCR: dictionaries (of constants, types, mode classes, variables); in VRS: axioms for subject domains, in SCR: environment assumptions dictionary. VRS transition consistency, completeness and safety checkers have SCR disjoints, coverage and property checkers respectively.

In comparison with SCR the VRS system for static checking uses more powerful formulas, allowing linear equalities and inequalities (both in the arithmetic of integers and in the arithmetic of real numbers), equalities for finitely enumerated types and variables with indexes (attributes with parameters).

In the framework of SCR project, to resolve the problems of behavioral properties checking, groups of inter-supplementing tools oriented both at inconsistency search and at verification of specifications were created. The same approach to the tools development was exercised in the VRS project, where were developed tools to cover such functionalities, especially simulators (trace generators) for searching of inconsistent traces at a level of concrete and symbolic models, as well as checkers that perform verification were developed.

Another similar tool is Action Language Verifier (ALV) which is an infinite state model checker. It uses linear arithmetic constraints on integer variables for system specification which is presented in Action Language. Such specifications involve integer, Boolean and enumerated variables, parameterized integer constants and a set of modules and actions composed with synchronous and asynchronous compositions. Statecharts and SCR specifications can be easily translated to Action Language. Like VRS, ALV uses symbolic model checking techniques to verify or falsify behavioral properties of the input specifications. Current version of ALV uses two different symbolic representations for integer variables: polyhedral and automata representation; for bounded integers BDD representation is used. These techniques permit to construct rather powerful symbolic model checker.

The VRS approach uses MSC as a language for presenting system requirements. Actually, an extended MSC is realized in VRS checkers. This decision to orientation on engineering languages is in good accordance with other approaches to requirements verification. Let us mention such MSC extensions as Live Sequence Charts (LSC [26]), Triggered Message Sequence Charts (TMSC [27]), Object Message Sequence Charts (OMSC [28]) and others.

Future VRS development is connected with extension of BPS Language, and new algorithms for symbolic checking and invariant construction.

1. *Letichevsky and D. Gilbert*. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer, Springer, 1999, Science 1827,.
2. *Letichevsky A.* Algebra of behavior transformations and its applications, in V.B.Kudryavtsev and I.G.Rosenberg eds. Structural theory of Automata, Semigroups, and Universal Algebra, NATO Science Series II. Mathematics, Physics and Chemistry – Springer 2005. – Vol. 207, P. 241–272.
3. *Baranov S., Jervis C., Kotlyarov V., Letichevsky A., and Weigert T.* Leveraging UML to Deliver Correct Telecom Applications. In L. Lavagno, G. Martin, and B.Selic, editors. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, Amsterdam, 2003.
4. *Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V.Kotlyarov, T. Weigert*. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks. – 2005, 47. – P. 662–675.
5. *J. Kapitonova, A. Letichevsky, V. Volkov, and T. Weigert*. Validation of Embedded Systems. In R. Zurawski, editor. The Embedded Systems Handbook. CRC Press, Miami, 2005.
6. *A.A.Letichevsky, J.V.Kapitonova, V.A.Volkov, A.A.Letichevsky, jr., S.N.Baranov, V.P.Kotlyarov, T.Weigert*. System Specification with Basic Protocols, Cybernetics and System Analyses, 4, 2005.
7. *International* Telecommunications Union. Recommendation Z.120 Annex B: Formal semantics of Message Sequence Charts, 1998.
8. *Reniers M.A.* Message Sequence Chart: Syntax and Semantics. Eindhoven, University of Technology, 1998.
9. *Letichevsky A.A., Kapitonova J.V., Kotlyarov V.P., Volkov V.A., Letichevsky A.A. Jr., and Weigert T.* Semantics of Message Sequence Charts, SDL Forum, 2005.
10. *Letichevsky A.A, Kapitonova J.V., Kotlyarov V.P., Letichevsky A.A. Jr, Volkov V.A.* Semantics of timed MSC language, Kibernetika and System Analysis, 2002.
11. *Degtyarev, J. Kapitonova, A. Letichevsky, A. Lyaletsky, and M. Morokhovets.* Evidence algorithm and problems of representation and processing of computer mathematical knowledge. Kibernetika and System Analysis, (6):9–17, 1999.
12. C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.
13. R. Milner. Communication and Concurrency. Prentice Hall, 1989.
14. L. Lamport. Introduction to TLA. SRC Technical Note 1994-001, 1994.
15. L. Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, May 1994. – P. 872-923,
16. Pnueli. The temporal logic of programs // Proc. of the 18th Annual Symp. on the Foundations of Comp. Sci. – Nov. 1977. – P. 46–52,
17. E. Emerson and J. Halpern. Decision procedures and expressiveness in the temporal logic of branching time // J. of Comp. and System Sci., 1985. – 30(1):1–24.
18. M.J. Fisher and R. E. Ladner. Propositional modal logic of programs // Proc. 9th ACM Ann. Symp. on Theory of Comp. – Boulder, Col., May 1977. – P. 286–294.
19. *Emerson E.* Temporal and modal logic. In: J. van Leeuwen editor: Handbook of Theoretical Computer Science, Elsevier, (B):997–1072, 1990.
20. *Bryant R.* Graph-based algorithms for Boolean function manipulation // IEEE Trans. on Computers, 35 (8). – 1986. – P. 677–691.

21. *Clarke E. and Emerson E.* Synthesis of synchronization skeletons for branching time temporal logic // The Workshop on Logic of Programs, 128–143, Lecture Notes in Computer Sci., 131. Springer Verlag, 1981.
22. J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In: Proc. 5th Intern. Symposium on Programming , 142–158, 1981.
23. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^20 states and beyond. Information and Computation, 98 (2): 142–170, 1992.
24. Constance Heitmeyer, Myla Archer, Ramesh Bharadwaj and Ralph Jeffords/ Tools for constructing requirements specifications: The SCR toolset at the age of ten, Computer Systems Science & Engineering, 1: 19-35, 2005.
25. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In Proc. of ASE 2001, 382–386, November 2001.
26. David Harel, Rami Marelly. Come, Let's Play: Scenario-Based programming Using LSCs and the Play-Engine. Springer 2003.
27. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. In Proceedings of SIGSOFT 2002/FSE-10, 167–176, ACM Press, 2002
28. Frank Buschmann, Regine Meunier Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture-A System of Patterns. Wiley & Sons, New York, 1996.

Date received 07.07.2008

About the authors:

Letychevsky Alexander,
PhD, member-correspondent of National Academy of Sciences of Ukraine, Chair of department

Kapitonova Julia,
PhD, Professor, Chair of department

Letychevskyi Oleksandr,
PhD, researcher

Kotlyarov Vsevolod,
PhD, leading technical specialist of Motorola

Nikitchenko Nikolaj,
PhD, Professor, Chair of department, faculty of cybernetics

Volkov Vladyslav,
PhD, researcher

Thomas Weigert,
PhD, Professor of Computer Science, Department of Computer
Science

Author's affiliation:

Glushkov Institute of Cybernetics, National Academy of Sciences of Ukraine, Glushkov ave., N40,
phone: 38 044 5260058,
fax: 38 044 525 1558,
let@cyfra.net

phone: 38 044 5260058,
fax: 38 044 525 1558,
kap@d100.icyb.kiev.ua

phone: 38 044 5260058,
fax: 38 044 525,
lit@iss.org.ua

St.Petersburg software development center, Sedova str., N14, St.Petersburg, Russia,
phone: 7 812 329 19,
fax: 7 812 329 19 12,
Vsevolod.Kotlyarov@motorola.com,

Shevchenko National Kiev University, Glushkova ave., N2, building 6,
phone: 2590519,
fax: 521 33 45,
nikitchenko@unicyb.kiev.ua

Glushkov Institute of Cybernetics, National Academy of Sciences of Ukraine, Glushkov ave., N40,
phone: 38 044 5260058,
fax: 38 044 525,
vlad_volkov_98@yahoo.com

Missouri University of Science and Technology, 500 West 15th
Street 325 Computer Science Bldg. Rolla, MO 65409, USA (573) 341-4491,
weigert@mst.edu