Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications

A. Letichevsky*, J. Kapitonova, A. Letichevsky Jr., V. Volkov
Glushkov Institute of Cybernetics, National Academy of Science, Kiev, Ukraine

S. Baranov†, V. Kotlyarov
Motorola, St.Petersburg, Russia

T. Weigert
Motorola, Schaumburg, Illinois, United States

Abstract

Message sequence charts are a widely used notation to express requirements specifications of multi-agent systems. The semantics of message sequence charts can be defined algebraically in the theory of agents and insertion functions. Using this algebra, one can split message sequence chart scenarios into sets of Hoare triples consisting of precondition, the specification of a finite process, and a postcondition. We refer to such triples as "basic protocols." In this paper, we discuss tools to prove properties of systems described as basic protocols, such as the completeness (at each of its stages the system behavior has a possible continuation) and consistency (at each stage the system behavior is deterministic) of the specification, or the correspondence of the specified behavior to given scenarios. Together, these tools constitute a powerful environment for the formal verification of requirements specifications expressed through message sequence charts.

1. Introduction

In [6] requirements capture is defined as an engineering process of determining the artifacts to be produced as the result of a development effort. The process includes the following steps:

- Requirements identification,
- Requirements analysis,
- Requirements representation,
- Requirements communication, and
- Development of acceptance criteria and procedures.

*Email: let@iss.org.ua
†Email: Sergey.Baranov@motorola.com

Requirements are the agreement between the customer and the developer regarding the artifacts to be produced. As agreements, they must be clear to the customer as well as to the developer and their level of formalization depends on the common understanding between these parties and the experience of those involved in the process of requirements identification. Unfortunately, most practices of describing requirements and preliminary designs (be they through natural-language, diagrams, or pseudocode) do not offer mechanized means of establishing their correctness. Therefore, the primary way to deduce properties of the specification and its consequences is through inspections and reviews.

To be amenable to formal analysis, requirements must first be formalized, i.e., rewritten in a formal language. The need of formalization imposes limits on the deployment of traditional formal methods in industrial applications in that it requires familiarity with logical notions and notations these methods use. Such familiarity is not wide-spread among todays software engineers. An additional restriction on the application of deductive tools (e.g., PVS [20]) is that these require the development of a mathematical theory of the domain at a very detailed level to implement even very simple predicates. Mathematical sophistication is usually needed.

Alternatively, instead of relying on formalized specifications, formal methods have been successfully applied to specifications captured in design languages widely accepted in the engineering community, such as MSC [10], SDL [11], or UML [12].

The PTK [1] system implements automated syntax and semantic analysis on MSC diagrams and generates test scripts from such diagrams in various languages including SDL, TTCN, and C. FatCat [19] is aimed at discovering non-determinism in sets of MSC diagrams. Various projects have applied model checking and automated theorem proving to SDL specification, by converting such specifications into appropriate input forms: The IF system from Verimag translates SDL to PROMELA code and then applies the SPIN model checker [5]. At Siemens, verification of part of the GSM protocol was conducted using the BDD-based model checker SVE [21]. An integrated framework for processing SDL specification has been implemented based on the automated theorem prover ACL2 [23]. The project OMEGA (with participation of two tool vendors, Telelogic and iLogix) aims at the development of formal tools for the analysis and verification of design steps based on UML specifications [9].

However, in spite of significant research that has been devoted to the development of formal methods, their application has failed more often than not when confronted with real-life, industrial specifications. Most formal verification methods utilize generic decision procedures which are not effective on large applications.

In this paper we present a new approach based on the notion of basic protocols and the theory of interaction of agents and environments. This approach has been implemented in the system VRS (Verification of Requirement Specifications) and has been piloted in a number of industrial software development projects. The VRS system is based on algebraic and insertion programming [13, 17]. It is an open environment and can be adjusted easily to new subject domains.

Basic protocols represent system requirements in the form of Hoare triples $\alpha \rightarrow < P > \beta$, where P is a process, and α and β are logical formulae (constituting pre and post-conditions of process P). Requirements stated as Hoare triples closely resemble the requirements used in engineering practice. (The main difference to the latter, of course, being the use of formal language instead of natural language.) Software designer typically specify system requirements as a set of possible behavior fragments expressing the system functionalities rather than developing

specifications in the form of complete scenarios.

The representation of requirements as basic protocols is based on the theory of interacting agents and environments [16]. In contrast to the major traditional theories of interaction, including CCS [18], CSP [8], or ACP [3, 4] which are based on an implicit and hence not formalized notion of an environment, the theory of interaction of agents and environments studies agents and environments as objects of different types. An environment may be considered as yet another agent, but sometimes an environment for a given agent is obtained from considering all other agents of the system acting in parallel with the given one. This theory was first introduced in [7], and we will describe its basic notions in Section 2.

Section 3 introduces the formal definition of basic protocols and related concepts. In Section 4, we present verification techniques that our system is able to apply to specifications defined by basic protocols: checking of transition consistency and completeness, checking of time consistency, checking against annotated scenarios, and generating traces and scenarios for testing.

Sections 5 and 6 introduce MSC as a language for expressing requirements and describe how MSC diagrams are mapped to the underlying algebraic framework. An example from the telecommunications domain illustrates our verification techniques. We specify a simple telecommunications protocol by a set of MSC diagrams representing basic protocols and demonstrate how an error can be detected.

We conclude with a short discussion of the implementation of the VRS system and report results of the application of this tool in several large-scale industrial projects.

2. Agents and environments

We formalize system requirements in a language based on process algebra [4] enriched by the model of interaction of agents and environments [16]. *Agents* are labeled transition systems with states considered up to bisimilarity. They interact with each other by performing observable actions. The notion of an agent formalizes such diverse entities as software components, programs, users, clients, servers, or active components of distributed systems.

A state of an agent is defined by its behavior. Therefore, the equivalence of agents can be characterized in terms of the complete and continuous behavior algebra $F(A)$. This is an algebra with two sorts of elements—behaviors $u \in F(A)$, represented as finite or infinite labeled trees, and actions $a \in A$. As in basic process algebra, two operations are defined over $F(A)$: *nondeterministic choice*, which is an associative, commutative, and idempotent binary operation $u+v$, where $u, v \in F(A)$ and *prefixing* $a.u \in F(A)$, where $a \in A, u \in F(A)$. The neutral element of nondeterministic choice is the deadlock element 0 (representing the impossible behavior). The empty behavior Δ performs no actions and denotes successful termination of the behavior of an agent. The algebra $F(A)$ is partially ordered by the approximation relation $\sqsubseteq$ with minimal element $\perp$. Both operations are continuous functions on the set of all behaviors over A. Completeness means that any directed set of behaviors has a least upper bound.

Each element u of a behavior algebra has a canonical representation

$$u = \sum_{i \in I} a_i.u_i + \varepsilon_u$$

defined up to commutativity and associativity of nondeterministic choice. In this representation, all $a_i.u_i$ are different behaviors, I is a finite or infinite set of indices, and $\varepsilon_u \in \{0, \Delta, \perp, \Delta + \perp\}$.

We say that behavior v is *reachable* from u if $u = v + u'$ or (inductively) v is reachable from u_i for some $i \in I$. We call behavior u *divergent* if $\bot$ is reachable from u and *convergent* otherwise.

An *environment* E is an agent over a set of actions C together with an *insertion function* $\mathbf{Ins}(e, u)$ which we denote by $e[u]$. Its first argument e is a behavior of an environment, the second argument is a behavior of an agent over a set A of actions (of the agent) in a given state u. An insertion function is an arbitrary function continuous in both of its arguments. It yields a new behavior of the same environment.

We define an equivalence relation over agents which is, in general, weaker than bisimilarity. Two agents (in given states) u and v are insertion equivalent with respect to an environment E, written $u \sim_E v$, if for all $e \in E$, $e[u] = e[v]$. After the insertion of an agent into an environment, the new environment is ready to accept new agents to be inserted and multiple insertion allows to consider states of the form $e[u_1, u_2 \ldots, u_n]$ (shorthand for $(...((e[u_1])[u_2])...)[u_n]$).

To define an insertion function, one defines labeled transitions on the set of states $e[u]$ of an environment with an inserted agent. We rely on rewrite rules

$$F(x)[G(y)] \to d.F'(z)[G'(z)]$$

$$F(x)[G(y)] \to F'(z)[G'(z)]$$

where $x = (x_1, x_2, \ldots)$, $y = (y_1, y_2, \ldots)$, $z = (x_1, x_2, \ldots, y_1, y_2, \ldots)$, $x_1, x_2, \ldots$, $y_1, y_2, \ldots$ are action or behavior variables, $d \in C$, and F, G, F', G' are expressions in behavior algebra, that is, expressions built by nondeterministic choice and prefixing. The first kind of rule defines transitions of a type

$$F(x)[G(y)] \xrightarrow{d} F'(z)[G'(z)]$$

The second kind of rule defines unlabelled transitions. These are not observable; the definition of environment behavior includes the following rule

$$\frac{e[u] \xrightarrow{*} e'[u'],\ e'[u'] \xrightarrow{d} e''[u'']}{e[u] \xrightarrow{d} e''[u'']}$$

where $\xrightarrow{*}$ denotes the transitive closure of unlabelled transition. Rewrite rules must be left linear with respect to their behavior variables, that is, no behavior variables may occur more than once in the left hand side. Further, rewrite rules for terminal and divergent states must be added to ensure that the insertion function is continuous. Rewrite rules may define nondeterministic transition relations when two different left hand sides can be matched with the same state of an environment $e[u]$ (critical pairs are allowable).

We consider attributed transition systems, that is, systems with a mapping from states to attribute values. For attributed transition systems, the notion of bisimilarity must be slightly modified, and the behavior algebra should be considered together with an attribute mapping from behaviors to the attribute domain.

For more details about a theory of interacting agents and environments see [15, 16, 17].

3. System specification by means of basic protocols

Base language. Basic protocols are functionally definite fragments of system behavior. A system is defined as an attributed transition system, and the states of a system are observed by

means of performed actions as well as attributes and variables defined on states changing their values in time. Properties of states are defined by means of formulae of some logic, which we refer to as the *base language*. Typically, the base language is first order, possibly with typed variables. The formulae of the base language may have attributes as the only free variables. Attributes may belong to functional types; to avoid higher order types, attributes may depend on parameters and functional expressions are restricted to first order expressions. The state of a system consists of the state of the environment which defines the values of attributes and the states of agents inserted into this environment if they are observable after insertion.

The choice of abstraction level of the base language is critical and depends on the problem domain and the state of development. Usually, a low level of abstraction with concrete states is used when the development of a system has been completed and the intention is to generate test cases from the requirements. A higher abstraction level is useful when one attempts to prove properties of the system requirements, where large collections of agents (processors in multi-processor system, mobile phones, etc.) can be replaced by some formula expressing important properties of these collections.

Permutability relation. Before defining transitions, the set of actions C that can be performed by the system and observed by the external world needs to be defined. Actions are functional expressions of the base language and may depend on attributes. To generate the behavior of a system defined by basic protocols, we shall use a binary relation $a \leftrightarrow b$ on the set of actions called permutability relation. We assume that predicate $a \leftrightarrow b$ belongs to the base language, therefore its validity depends on the current state of a system and we can compute either semantic $s \models a \leftrightarrow b$ or syntactic inference $\alpha \vdash a \leftrightarrow b$, where α is a formula (or a set of formulae) of the base language. In the following, we shall assume that permutability relation does not depend on the state of a system; however, all main concepts can be extended to the general case.

We define permutability for the case $u \leftrightarrow b$ where u is a behavior over C and b is an action. This is the minimal relation such that:

1. For all actions b, $\Delta \leftrightarrow b$, $\bot \not\leftrightarrow b$, $0 \not\leftrightarrow b$;
2. $(u + v) \leftrightarrow b \Leftrightarrow u \leftrightarrow b \wedge v \leftrightarrow b$;
3. $a.u \leftrightarrow b \Leftrightarrow a \leftrightarrow b \wedge u \leftrightarrow b$.

From this definition it follows that action b is permutable with behavior u if all actions reachable from u are permutable with b (action a is reachable from u if some v is reachable from u and $v \xrightarrow{a} v'$), u is convergent, and 0 is not reachable from u.

Partial sequential composition of two behaviors. Let

$$u = \sum_{i \in I} a_i.u_i + \varepsilon_u, \;\; v = \sum_{j \in J} b_j.v_j + \varepsilon_v$$

Then

$$u * v = \sum_{u \leftrightarrow b_j, j \in J} b_j.(u * v_j) + \sum_{i \in I} a_i.(u_i * v) + (\varepsilon_u ; \varepsilon_v)$$

where $(\Delta ; \varepsilon) = \varepsilon$, $(\bot ; \varepsilon) = \bot$, $(0 ; \varepsilon) = 0$. Note that partial sequential composition is not continuous in the first argument, but it is continuous in the second one, and it is continuous in both when the first argument is finite and convergent.

If the permutability relation is always false, partial sequential composition coincsides with sequential composition. If all actions are permutable, it coincsides with interleaving parallel composition. Partially sequential composition originates from weak sequential composition introduced by Renier [22] for the definition of the operational semantics of MSC and generalizes it further.

Basic protocols. Each basic protocol is a Hoare triple $\alpha \rightarrow < P > \beta$, where P is a process, α and β are precondition and postcondition of process P, respectively. α and β are represented by logical expressions of the base language and define conditions on the set of states of a system. A process of a basic protocol is a finite convergent process over the set C of environment actions, which may contain the set A of agent actions. We shall use the following notation for arbitrary basic protocols: $\mathbf{pre}(b) = \alpha, \mathbf{post}(b) = \beta$, and the process of B is denoted as P_b.

Each basic protocol defines properties of a system and can be understood as a statement of temporal logic: if the precondition is true then the process of a protocol can start, and after it has successfully terminated, the postcondition must be true.

Predicate transformers. Assume an assertion φ in the form of a formula of the base language means $\forall s(s \models \varphi)$ or $\vdash \varphi$ in a given theory.

A predicate transformer $\mathbf{Tr}(\alpha, \beta)$ is a function defined on formulae of the base language returning a new formula such that $\mathbf{Tr}(\alpha, \beta) \rightarrow \beta$. A predicate transformer strengthens the postcondition of a basic protocol by adding residual properties from the precondition.

Systems specified by basic protocols. A system is specified by its initial state and its properties. Let the initial state be described by a set of properties expressed in the base language, denoted as α_0. We then denote the behavior of a system generated by a set of basic protocols B and initial state satisfying α_0 by $S(B, \alpha_0)$. The system is usually not defined uniquely by the initial state α_0; rather, several protocols may be applicable in the initial state as the initial state α_0 may imply their precondition. Therefore, we can define the behavior of a system as the nondeterministic sum of behaviors starting in the initial state

$$S(B, \alpha_0) = \sum_{\alpha_0 \rightarrow \alpha} S_\alpha$$

The behavior S_α is the partially sequential composition of basic protocols from B. The first protocol is arbitrarily chosen from those basic protocols with a precondition satisfied by α. The set of all such conditions is $B(\alpha) = \{b \in B | \alpha \rightarrow \mathbf{pre}(b)\}$. When the process of a basic protocol has completed, the postcondition of this basic protocol will be true. In fact, a stronger set of conditions may be true, as the postcondition may not take all the aspects of the precondition into account. We consider the stronger condition given by the predicate transformer $\mathbf{Tr}(\alpha, \mathbf{post}(b))$. Consequentially, the behavior S_α is defined as

$$S_\alpha = \sum_{b \in B(\alpha)} P_b * (S_{\mathbf{Tr}(\alpha, \mathbf{post}(b))} + \Delta)$$

The summand Δ is added to generate not only infinite traces, but also finite ones. When the set $B(\alpha)$ is empty, $S_\alpha = 0$. Therefore, if Δ is absent, all finite traces, if any, terminate in the deadlock state 0.

A system is defined up to bisimilarity as a minimal fixed point of the above equations in the behavior algebra.

Scenarios. A system $S(B, \alpha_0)$ represents all possibilities of selecting basic protocols to construct behaviors. At times we are interested in a partial system description which can be obtained by restricting the choice of basic protocols. Behaviors obtained this way are called scenarios generated by basic protocols. To formalize this constuction we consider the set $\mathbf{S}_\alpha$ of scenarios generated by the set of basic protocols B starting from initial condition α. This set is defined as a maximal set satisfying the following condition: If $S \in \mathbf{S}_\alpha$, then there exists $b \in B$ such that $\alpha \rightarrow \mathbf{pre}(b)$ and $S = P_b * S' + S''$ where $S' \in \mathbf{S}_{\mathbf{Tr}(\alpha,\mathbf{post}(b))}$ and $S'' \in \mathbf{S}_\alpha$ or $S = P_b$.

If the set $B(\alpha)$ is not empty, then neither is the set $\mathbf{S}_\alpha$ because it contains the *universal scenario* $S(B, \alpha)$ as well as $B(\alpha)$.

Parameterized basic protocols have the general form

$$\forall x(\alpha(x) \rightarrow < P(x) > \beta(x))$$

where $x = (x_1, \ldots, x_n)$. Parameterized basic protocols are used when there are infinitely many or, at least, a great number of similar basic protocols. Bound variables can be typed if the base language allows types. Substitution of constant (ground) values for x gives us the set $\mathbf{Inst}(B)$ of instantiated basic protocols; this set must be used instead of B in the definitions above.

4. Consistency and completeness of basic protocols and scenarios

Many important properties of requirement specifications can be checked during requirements capture. First of all, requirements characterizing the total behavior of a system may be expressed in terms of temporal modalities (dynamic requirements) including safety and liveness conditions. These requirements must be consequences of static requirements expressed by means of basic protocols. Well-established model checking techniques can be used to check whether these requirements hold in the model defined by a set of basic protocols.

Unfortunately, when initially specified, the set of basic protocols is often inconsistent or incomplete. Different forms of inconsistency and incompleteness occur in practice.

Transition consistency of basic protocols. The behavior of a system defined by basic protocols is characterized by the scenarios it generates. When generating a scenario, at each step an applicable basic protocol must be selected. A basic protocol is applicable at a state if its precondition is true in that state, given the values of its state variables (attributes of environment and agents). In order to construct a deterministic system, the selection of a protocol must depend only on the initial actions that can be performed when a protocol starts. Therefore, each time when some basic protocol can be applied and there is at least one initial action defined, there must be exactly one applicable basic protocol.

A sufficient condition for establishing the transition consistency of basic protocols is as follows: if the preconditions of two basic protocols are intersected, that is, if the negation of their conjunction cannot be proven or can be refuted, then the processes defined by these protocols as well as their postconditions must be equivalent, provided there exists a common initial action (weak consistency) or a common trace (strong consistency). To prove transition consistency, all pairs of basic protocols are considered. For each pair, the consistency condition (the negation of the conjunction of preconditions) is generated, and a proof attempt is initiated. If the proof succeeds (for all symbolic values of state variables and parameters), the pair of protocols is consistent, and these two protocols cannot be applied at the same time. Otherwise (the consistency

condition was not proven or was refuted), the protocols and the induced processes are checked for equivalence. Equivalent processes generate the same traces and have provably equivalent postconditions. Note that this condition is sufficient but not necessary, as it is possible that the intersection of a set of preconditions cannot be refuted, but there are no reachable states that validate this intersection.

Completeness of basic protocols implies that at any moment in time, there must be at least one basic protocol that can be used to continue the scenario at this point unless the scenario has terminated. A sufficient condition for completeness is for the disjunction of all preconditions of all basic protocols to be valid, for a given initial action. Actually, this condition is too strong and can be weakened if an admissibility condition is given for the set of protocols. By admissibility condition we refer to a precondition that is implied by a particular action occurring. In this case, the disjunction of the preconditions of the basic protocols in this set must be valid if the conjunction of the admissibility conditions for this set is valid.

Annotation consistency of scenarios. Each scenario generated by basic protocols can be annotated by assigning formulae of the base language to reachable behaviors.

Each system scenario must be generated by basic protocols. In addition, all annotations must be valid at corresponding states in any admissible basic protocol. If a scenario can be decomposed into a set of basic protocols and all annotations are valid, it is annotation consistent. To check for annotation consistency, we use symbolic simulation of scenarios. We start from initial conditions, determine which of the basic protocols can be applied by checking preconditions, match events found in the scenario against expected events according to the basic protocols generating the scenario, and obtain the conditions at the end of each protocol executed concurrently with others. Each time annotations are encountered, they are verified to be consequences of the conditions that currently hold at that state. The conditions characterizing the internal state of a scenarios may be insufficient for the selection of a basic protocol even if the set of basic protocols is transition consistent. In this case, the alternatives will be determined by control conditions. In case of loops, annotations can be used as loop invariants. If the invariant of a loop is proven, the loop needs to be symbolically evaluated only once.

5. Capturing requirements by basic protocols

Basic protocols resemble the natural language requirements statements found in typical system specifications. Consider the following two examples from industrial practice:

SYRaSRMenu_430 Upon determining that the SETUP_GREETING prompt has been completed AND IF a Voice Recognition Session is active AND MENU LEVEL is "Main Phone Setup" THEN the system shall request the audio input channel AND shall allow the user SESSION_SILENCE_TIMEOUT time to speak a voice command.

SYRaCSTATE_701 While in the NO_PHONE call state AND upon detecting that the Selected Device is set to a valid device AND the Selected Device's call status indicates a call in progress, the system shall assume it is in CIP.

While the stylized natural language still leaves room for interpretation, one can clearly discern pre- and postconditions and a processing section.

Requirements specifications are often presented in two parts. The first, as shown above, is the description of the fundamental system behavior in the form "if some specific conditions are satisfied then a corresponding sequence of actions is performed by each involved system component, and after completing these actions, the new system state will satisfy some new conditions." However, part of requirements specifications is also represented by means of scenarios describing characteristic interactions occurring between components of the system. Scenarios must be consistent with the fundamental system behaviors.

Basic protocols formalize elementary requirements of a system. We rely on two languages to define basic protocols. The first one is the base language used to state pre- and postconditions, the second one is the notation used to define the processes of basic protocols. Basic protocols can be combined into scenarios that describe fragments of system behavior starting from a given system state by means of a composition operation.

Each basic protocol has two meanings. The first meaning is as a behavior and is expressed as an expression of behavior algebra. The second meaning is as a predicate transformer which transforms the precondition into the postcondition. The definition of composition of two basic protocols must cater to both interpretations. Composition of basic protocols viewed as behavior is defined through the permutability relation of actions that captures the nondeterminism in the ordering of actions taken from different basic protocols. When the protocol process starts, not only its precondition will be true but typically some other conditions are also known to be satisfied. Some of these conditions (not mentioned in the precondition) remain true after performing a process, if they do not depend on the postcondition and are consistent with it. Therefore, when viewing basic processes as predicate transformers we strengthen the postcondition by adding these conditions.

Agents and environments. The state of an environment is represented by attributes of the environment of different types. The states of agents are defined on the corresponding level of abstraction by means of symbolic expressions of the base language. Agent states and agent names comprise two special domains used for state descriptions. When there are several types of agents, the set of agent types is also a special domain. Each agent inserted into the environment is uniquely identified by its name.

Processes. Traditional process algebra languages like CCS, CSP, or formalisms based on ACP can be used directly as process languages. However, languages like UML, MSC, or SDL are very popular in modern engineering practice and have a convenient, expressive graphical syntax. State machines, timed automata, wave diagrams etc. serve hardware and real time applications. In any case, behavior or process algebra can be used as a uniform foundation of the latter engineering approaches.

6. Message sequence charts and basic protocols

MSC diagrams are a convenient notation to express both basic protocols and scenarios. Conditions are used as control conditions for selecting alternatives in MSC diagrams, as well as for annotating scenarios, and to express pre- and postconditions.

An MSC diagram consists of a set of instances, or "life lines" which exhibit events that may occur during the execution of that instance. Such events may be the sending or receiving of a message, the occurrence of a local action, the setting or resetting of timers, the occurrence of a timeout, or the creation and stopping of an instance. The events on a life line are strictly

ordered by their occurrence, while no ordering exists between events on different life lines other than that imposed by the causality of events (for example, a message cannot be received before it is sent). Considering an instance of an MSC diagram to be represented by an agent of the behavior algebra, its actions represent the events occurring on an instance. An action is usually permutable with actions occurring at different instances; actions occurring on the same instance are usually not permutable. Therefore, partial sequential composition for agents representing MSC instances coincides with weak sequential composition. More subtle permutability relations take into consideration data dependencies arising from the execution of actions, or can be used to represent constructs such as coregions or general orderings.

An MSC diagram can be used to describe basic protocols in a convenient fashion. Each basic protocol so formalized begins with a global condition, representing the precondition of the basic protocol. The process is expressed as a finite MSC diagram without inline expressions which, therefore, can be represented as a set of transitions

$$P \xrightarrow{a_1} P_1$$
$$P \xrightarrow{a_2} P_2$$
etc.

where $a_1, a_2, \ldots$ represent the initial events in the MSC diagram. The process P is then

$$P = a_1.P_1 + a_2.P_2 + \ldots$$

The semantics of MSC diagrams can then be expressed by the following expression in the algebra of agents and environments:

$$e[a_1.P_1 + a_2.P_2 + \ldots]$$

where each P_i represents a partial sequential composition of actions (events of MSC diagrams) as described, and e is an environment which ensures the correct order of events. By defining the insertion function appropriately, we can capture a notion of behavior that precisely expresses the semantics of MSC diagrams.

Scenarios are simply MSC diagrams. However, for a scenario to be valid with respect to requirements captured as a set of basic protocols, a scenario must be the result of composing a subset of the basic protocols, using partially sequential composition, possibly instantiating their parameters.

Intuitively, composing two basic protocols into a scenario represented as an MSC diagram means drawing the first MSC diagram before the second one. This is straightforward for two (finite) diagrams comprising the same instances: The instances are matched, and all events of the first diagram on each instance precede the events on the second one. However, if the set of instances does not coincide, events on instances of the second diagram not part of the first diagram may occur at any time, as can events on instances of the first diagram not part of the second, subject to constraints imposed by the instances themselves. (This intuition resembles weak sequential composition of processes introduced by Reniers [22].) In the general case, we define a permutability relation on the set of actions permitting the possible first action of the second process to be performed first if it is permutable with all actions of the first process. For MSC diagrams permutability of two events means that they occur on different instances and they are not send or receive events for the same message. A weaker permutability relation could be defined if dependencies between actions exist that should be taken into account.

The postcondition of basic protocols expresses the state of the system after the execution of the process of the basic protocol. When the process starts executing, not only the precondition of the process are known to be true, but also other aspects of the system state that are not affected by this basic protocol. Some of these additional facts will remain true after the execution of the process. Predicate transformers are used to strengthen the postcondition of the basic protocol by these ancilliary (to this basic protocol) pieces of information.

An example from the telecommunications domain [2] will illustrate the reasoning over MSC diagrams expressing basic protocols. In this example, 24 MSC diagrams describe POTS (a basic call bearer service) extended by two supplementary services: Call Waiting (CW) and 3-way connection (3WAY). A subset of the basic protocols for this example[1] are shown in Figure 1. The only type of agent is **phone** and each agent has two symbolic attributes: **cw** (different from 0 when call waiting is in process), and **twc** (different from 0 if three way calling is in process). The states of agents as well as actions (messages exchanged between phones and network) appear in state assertions and processes of basic protocols. Predicate **valid**(n) is defined by the following condition

$$\mathbf{valid}(n) \Leftrightarrow \mathbf{phone}(n, \mathbf{idle}) \vee (\mathbf{phone}\ n.\mathbf{cw} = 0) \wedge \neg(\mathbf{phone}(n, \mathbf{ringing}))$$

Among the instances shown in MSC diagrams of this example, there is always an instance "Network" that corresponds to the environment for all agents. The other instances represent phones participating in basic protocols or scenarios. A set of axioms characterizing state assertions was initially formulated. Other axioms were discovered and added through unsuccessful attempts to prove consistency: when an inconsistency was found, it was analyzed to explain why the given preconditions are not intersected, which yielded additional axioms that allowed the proof to succeed.

Our tool then generated the consistency statements (there are more than 200 of them). Ten of these statements cannot be proven or refuted. These point to inconsistencies due to the interaction between the supplementary services. These inconsistencies can only be removed by redesigning the specification. An example of such inconsistency is revealed when examining the pair of protocols "three_way_teardown2" and "CW_teardown". These two protocols can have the common first action **onhook** z, and their corresponding consistency condition is

$$\begin{aligned}
&\forall(z, n', k', m'', n'')(\\
&\quad \neg(\mathbf{phone}(k', \mathbf{connected}\ z) \wedge \mathbf{phone}(n', \mathbf{cw_wait}\ k')) \vee \\
&\quad \neg(\mathbf{phone}(z, \mathbf{three_wayconnect}(m'' \wedge n'')))
\end{aligned}$$

This condition cannot be proven and, therefore, a warning appears in the verdict generated by the consistency checker. To ascertain whether the failure to prove the consistency condition indeed points to an inconsistency in the specification, one has to prove that a state satisfying the negation of the consistency condition is reachable from the initial state. Such state indeed exists, and our tool constructs a scenario leading to this state. The scenario found by the system is depicted in Figure 2. In this scenario, $k' = n''$; the network ends up in a state where, upon

[1] We use the expression $\mathbf{state}(x, m, s)$ as an atomic formula asserting that the agent with the name m has the type x and is in a state s at the current moment of time (as shorthand we use the notation $x(m, s)$). An attribute r of agent m of a type x is denoted as $x\ m.r$; if this attribute has parameters $t_1, \ldots, t_n$, it will be denoted as $x\ m.r(t_1, \ldots, t_n)$.

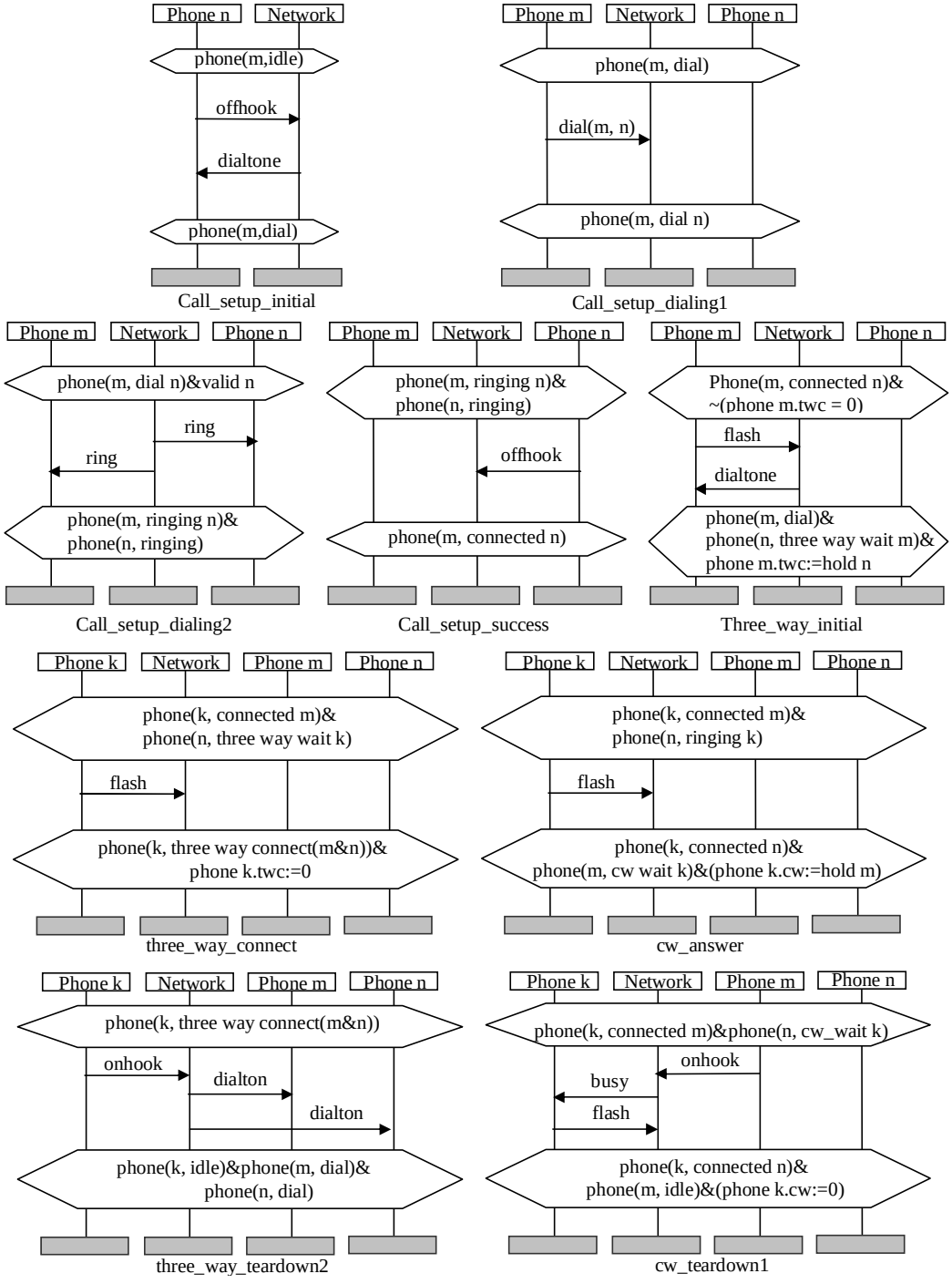

Figure 1: Basic protocols defining POTS augmented by CW and 3WAY.

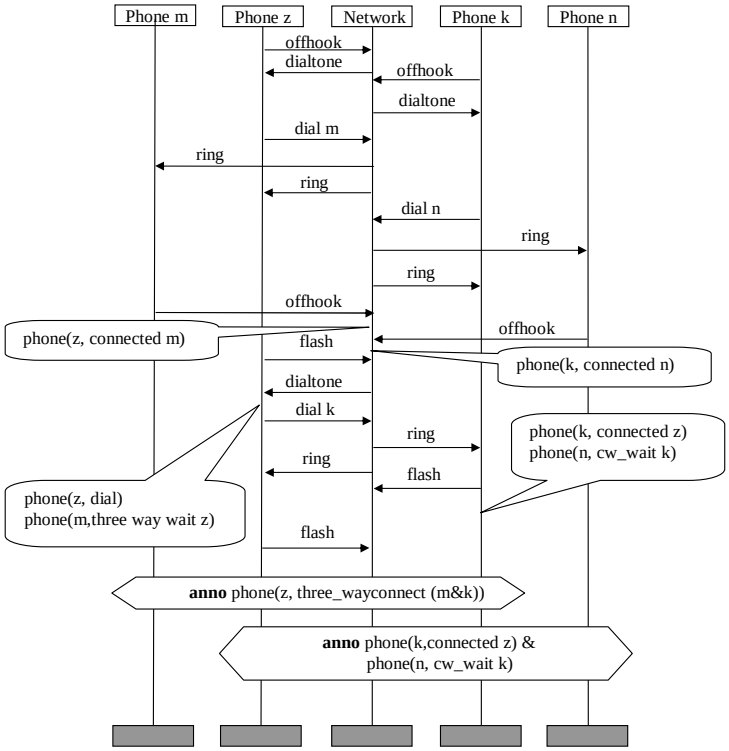

Figure 2: Scenario leading to the detected error.

receiving an onhook message, it would not know how to respond, as the two services demand a different response (it would have to either connect k' with n' or to send a dialtone to phone k').

7. Implementation [1]

VRS (Verification of Requirement Specifications) is based on the APS system [13] developed at the Glushkov Institute of Cybernetics. VRS is comprised of several implementation layers. On top is a level describing an environment specific for the subject domain under development which is tightly connected with the formalization of requirements. The second layer is the Action Language simulator and is the basis for the key functionalities of VRS. This simulator is implemented in APLAN, a language based on rewriting logic. As the lowest layer, the APLAN interpreter and its supporting libraries have been written in C/C++. The following tools have been realized within the VRS system.

- Creation and debugging of a specifications formalized as a set of basic protocols;
- Verification of the formalized requirements;
- Generation of traces and scenarios; and
- Proving of properties defined by annotated MSC and SDL specifications.

These tools are applied during requirements capture and testing: Creation, editing and debugging of basic protocols is supported by the Trace Generator Client. It allows easy creation of basic protocols via a graphical MSC editor, simulation of the system specified by basic protocols, and visualization of behaviors by directing the generation of traces. The description of the environment is created via special forms, supported with pop-up menus and hints. Trace generation is controlled by applying a set of filters and queries.

Verification of the set of basic protocols is performed by the Transition Consistency Checker. It processes the set of basic protocols and creates a verdict regarding the transition consistency and completeness of basic protocols.

The Trace Generator implements the generation of traces and scenarios from the set of basic protocols. Further, it detects deadlocks in the system of basic protocols, checks any safety conditions, and detects the reachability of different states of a system.

Specifications captured in SDL or MSC can be labeled by a set of annotations of the base language. Every annotation expresses a local property of the system. These annotations are checked during symbolic simulation using a special annotation checker and prover. The annotation checker is able to define the validity of local annotations and can construct a counter example where an annotation is not valid (not proved or refuted). In addition, it can construct a set of MSC traces that reach a given annotation.

The VRS system and its supporting tools have been piloted in a number of industrial projects; a sample of these projects is shown in the table below. Most projects are from the telecommunications or telematics domains. For each project, a number of defects has been detected in the requirements, and corresponding sets of traces have been generated. All pilot projects discovered errors in the early stages of software development which eases the demand on subsequent testing. Pilot projects followed the following steps:

Step 1. The informal natural language requirements specifications are developed.

Step 2. The requirements along with the accompanying technical information on the subject domain are studied by the piloting team and formalized specifications of the behavioral properties of the product are developed in the from of MSC diagrams.

Note 1. Studying and rewriting the requirements into formal specifications requires frequent consultations with domain experts.

Note 2. The developed formal specifications are reviewed by the domain experts (customers) for correspondence with their understanding of the original requirements.

Note 3. Typically, about 30% of the inconsistencies and incompletenesses in the original requirements are discovered in the process of formalizing the specifications.

Step 3. The formal requirements are processed by VRS to discover inconsistencies and incompletenesses, if any are present in the specification.

Note 4. VRS also generates a set of traces which guarantee complete coverage of the scenarios induced by the basic protocols. In case a defect is discovered, the respective trace (counter-example) can be converted into a test case which may be executed to put the system into the identified situation to help the developers understand the root cause of the defect.

Note 5. It is still necessary to "manually" test certain requirements, as complete automation of this process is impossible due to the algorithmic undecidability of this general problem. However, manual "residual testing" is restricted to a much smaller portion of the specification, typically by 2 orders of magnitude smaller. The VRS tool identifies those scenarios that have to be verified manually.

Step 4. The formalized requirements specification is corrected and becomes the basis for further development. System tests can be derived from the final requirements specification.

The verdict produced by VRS is in the same graphical language as the input specifications. In this case, the MSC diagrams are augmented with comments which represent test traces. The traces may be then automatically converted into tests to run in the respective test environment or directly on the respective hardware.

The table below summarizes the results of pilot projects. For each pilot project, it indicates the size of initial informal requirements in pages (column "Reqmts") and the corresponding number of formalized basic protocols (column "Protocols"). Column "Coverage" shows the amount of the requirements captured formally as a percentage of the total system specification. Column "Errors" shows the three main types of defects detected: the first type is found during the formalization stages; the second is automatically found while checking basic protocols for transition consistency, and the third one is found with the tests automatically generated from basic protocols. The final column "Effort" indicates the staff weeks required to formalize the requirements and to verify the specification.

Project	**Reqmts**	**Protocols**	**Coverage**	**Errors**	**Effort**
1	400	127	50%	11	5.5
2	200	70	100%	10	5.6
3	730	192	100%	18	20.0
4	624	56	20%	8	5.5
5	323	219	100%	38	3.0
6	116	42	100%	3	0.7

References

[1] P. Baker, P. Bristow, C. Jervis, D. King, and B. Mitchell, Automatic Generation of Conformance Tests From Message Sequence Charts, *Proceedings of 3rd SAM (SDL And MSC) Workshop, Telecommunication and Beyond*, Aberystwyth, UK, 24-26 June 2002, Lecture Notes in Computer Science, 2599, 170–198, 2003.

[2] S. Baranov, C. Jervis, V. Kotlyarov, A. Letichevsky, and T. Weigert, Leveraging UML to deliver correct telecom applications in *Uml for Real: Design of Embedded Real-Time Systems* by L. Lavagno, G. Martin, and B. Selic (editors), 323–342, Kluwer Academic Publishers, 2003.

[3] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1): 109-137, 1984.

[4] J.A. Bergstra, A. Ponse, and S.A. Smolka, Editors. *Handbook of Process Algebra*. Elsevier, 2001.

[5] M. Bozga, J.C. Fernandez, L. Ghirvth S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis "IF: An Intermediate Representation for SDL and its Applications", *Proceeding. of the Ninth SDL Foram*, Montreal, Quebec, Canada, 21-25 June, 1999 pp.423.

[6] J. Brackett, *Software requirements*. Technical Report SEI-CM-19-1.2, Software Engineering Institute, 1990.

[7] D.R. Gilbert and A.A. Letichevsky. A universal interpreter for nondeterministic concurrent programming languages. In M. Gabbrielli (editor), *Fifth Compulog network area meeting on language design and semantic analysis methods*, Sep. 1996

[8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[9] J. Hooman, Towards Formal Support for UML-based Development of Embedded Systems, *Proc. of the 3d PROGRESS Workshop on Embedded Systems*, STW, pages 71–76, 2002.

[10] International Telecommunications Union. Recommendation Z. 120–Message Sequence Chart (MSC), 1999.

[11] International Telecommunications Union. Recommendation Z. 100–Specification and Description Language (SDL), 1999.

[12] Object Management Group. Unified Modeling Language Specification, 2.0, 2003.

[13] J.V. Kapitonova, A.A. Letichevsky, and S.V. Konozenko. Computations in APS. *Theoretical Computer Science*, 119:145–171, 1993.

[14] J.-P. Katoen, L. Lambert. Pomsets for Message Sequence Charts, *1st Workshop of the SDL Forum Society on SDL and MSC*, SAM98, Berlin, June/July 1998.

[15] A.A. Letichevsky and D.R. Gilbert. A general theory of action languages. *Cybernetics and System Analysis*, (1):16–36, Feb. 1998.

[16] A.A. Letichevsky and D.R. Gilbert. A Model for Interaction of Agents and Environments. In: *Selected papers from the 14th International Workshop on Recent Trends in Algebraic Development Techniques*. Lecture Notes in Computer Science, 1827, 311–328, 1999.

[17] A.A. Letichevsky, Y.V. Kapitonova, V.A. Volkov, V.V. Vyshemirsky, and A.A.Letichevsky, Jr. Insertion Programming. *Cybernetics and System Analysis*, Kiev, 1, 2003

[18] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[19] B. Mitchell, R. Thomson, C. Jervis. Phase Automaton for Requirements Scenarios, *Proceedings of Feature Interactions in Telecommunications and Software Systems VII*, 2003, pp. 77-87, IOS Press.

[20] S. Owre, N. Shankar, and J.M. Rushby. User Guide for the PVS Specification and Verification System. *Technical Report*, SRI International, 1996.

[21] F. Regensburger, A. Barnard. Formal verification of SDL systems at the Siemens mobile phone department. *In Tools and Algorithms for the Construction and Analysis of Systems*. 4 th International Conference, ACAS'98, Lecture Notes in Computer Science, 1384, 439-455. Springer Verlag, 1998.

[22] M.A. Reniers. *Message Sequence Chart : Syntax and Semantics*. Eindhoven, Eindhoven, University of Technology, 1998.

[23] 0. Shumsky, L.J. Henschen. Developing a framework for verification, simulation and testing of SDL specifications. In M. Kaufmann and J S. Moore, editors, Proceedings of the ACL2 Workshop 2000. University of Texas at Austin, 2000.